pandagg Documentation

Release 0.1

Léonard Binet

Sep 23, 2021

Contents

1 Principles **1**
- 1.1 Elasticsearch tree structures . . . 1
- 1.2 Interactive usage . . . 1

2 User Guide **3**
- 2.1 Search . . . 3
  - 2.1.1 Query part . . . 4
  - 2.1.2 Aggregations part . . . 5
  - 2.1.3 Other search request parameters . . . 6
  - 2.1.4 Request execution . . . 6
- 2.2 Query . . . 6
  - 2.2.1 Declaration . . . 7
    - 2.2.1.1 From native “dict” query . . . 7
    - 2.2.1.2 With DSL classes . . . 8
    - 2.2.1.3 With flattened syntax . . . 8
  - 2.2.2 Query enrichment . . . 8
    - 2.2.2.1 query() method . . . 9
    - 2.2.2.2 Compound clauses specific methods . . . 9
    - 2.2.2.3 Inserted clause location . . . 10
- 2.3 Aggregation . . . 11
  - 2.3.1 Declaration . . . 11
    - 2.3.1.1 From native “dict” query . . . 11
    - 2.3.1.2 With DSL classes . . . 12
    - 2.3.1.3 With flattened syntax . . . 12
  - 2.3.2 Aggregations enrichment . . . 13
- 2.4 Response . . . 13
  - 2.4.1 Hits . . . 14
  - 2.4.2 Aggregations . . . 15
    - 2.4.2.1 Tree serialization . . . 16
    - 2.4.2.2 Tabular serialization . . . 17
- 2.5 Interactive features . . . 18
  - 2.5.1 Cluster indices discovery . . . 18
  - 2.5.2 Navigable mapping . . . 19
  - 2.5.3 Navigable aggregation response . . . 20

3 IMDB dataset **25**
- 3.1 Query requirements . . . 25

3.2 Data source . . . 25
3.3 Index mappings . . . 26
3.3.1 Overview . . . 26
3.3.2 Which fields require nesting? . . . 26
3.3.3 Text or keyword fields? . . . 26
3.3.4 Mappings . . . 26
3.4 Steps to start playing with your index . . . 27
3.4.1 Dump tables . . . 27
3.4.2 Clone pandagg and setup environment . . . 27
3.4.3 Serialize movie documents and insert them . . . 28
3.4.4 Explore pandagg notebooks . . . 28

4 pandagg package **29**
4.1 Subpackages . . . 29
4.1.1 pandagg.interactive package . . . 29
4.1.1.1 Submodules . . . 29
4.1.1.2 Module contents . . . 29
4.1.2 pandagg.node package . . . 29
4.1.2.1 Subpackages . . . 29
4.1.2.2 Submodules . . . 54
4.1.2.3 Module contents . . . 54
4.1.3 pandagg.tree package . . . 54
4.1.3.1 Submodules . . . 54
4.1.3.2 Module contents . . . 61
4.2 Submodules . . . 61
4.2.1 pandagg.aggs module . . . 61
4.2.2 pandagg.discovery module . . . 72
4.2.3 pandagg.exceptions module . . . 72
4.2.4 pandagg.mappings module . . . 73
4.2.5 pandagg.query module . . . 79
4.2.6 pandagg.response module . . . 86
4.2.7 pandagg.search module . . . 87
4.2.8 pandagg.types module . . . 94
4.2.9 pandagg.utils module . . . 95
4.3 Module contents . . . 95

5 Contributing to Pandagg **97**
5.1 Our Development Process . . . 97
5.2 Pull Requests . . . 97
5.3 Any contributions you make will be under the MIT Software License . . . 97
5.4 Issues . . . 98
5.5 Report bugs using Github's issues . . . 98
5.6 Write bug reports with detail, background, and sample code . . . 98
5.7 License . . . 98
5.8 References . . . 98

6 Installing **99**

7 Usage **101**

8 License **103**

9 Contributing **105**

Python Module Index **107**

Index 109

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: pandagg.tree.mappings.Mappings*, *client: Optional[elasticsearch.client.Elasticsearch] = None*, *index: Optional[List[str]] = None*, *depth: int = 1*, *root_path: Optional[str] = None*, *initial_tree: Optional[pandagg.tree.mappings.Mappings] = 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

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: str, type_or_agg: Union[str, Dict[str, Dict[str, Any]], pandagg.node.aggs.abstract.AggClause, None] = None, **body*) → pandagg.node.aggs.abstract.AggClause

Accept multiple syntaxes, return a AggNode instance.

Parameters

- **name** – aggregation clause name
- **type_or_agg** –
- **body** –

Returns AggNode

class pandagg.node.aggs.abstract.**AggClause**(*meta: Optional[Dict[str, Any]] = None, identifier: Optional[str] = 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.

classmethod extract_bucket_value(*response: Union[pandagg.types.BucketsWrapperDict, Dict[str, Any]], value_as_dict: bool = False*) → Any

extract_buckets(*response_value: Union[pandagg.types.BucketsWrapperDict, Dict[str, Any]]*) → Iterator[Tuple[Union[None, str, float, Dict[str, Union[str, float, None]]], Dict[str, Any]]]

is_convertible_to_composite_source() → bool

line_repr(*depth: int, **kwargs*) → Tuple[str, str]

Control how node is displayed in tree representation. First returned string is how node is represented on left, second string is how node is represented on right.

MyTree ├── one OneEnd │ └── two twoEnd └── three threeEnd

to_dict() → Dict[str, Dict[str, Any]]

ElasticSearch aggregation queries follow this formatting:

```
{
    "<aggregation_name>" : {
        "<aggregation_type>" : {
            <aggregation_body>
        }
        [,"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: str*) → bool

class pandagg.node.aggs.abstract.**BucketAggClause**(***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') >>> } >>>)

extract_buckets(*response_value: Union[pandagg.types.BucketsWrapperDict, Dict[str, Any]]*) → Iterator[Tuple[Union[None, str, float, Dict[str, Union[str, float, None]]], Dict[str, Any]]]

class pandagg.node.aggs.abstract.**FieldOrScriptMetricAgg**(*field: Optional[str] = None*, *script: Optional[pandagg.types.Script] = None*, ***body*)

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

Metric aggregation based on single field.

class pandagg.node.aggs.abstract.**MetricAgg**(*meta: Optional[Dict[str, Any]] = None*, *identifier: Optional[str] = None*, ***body*)

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

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

extract_buckets(*response_value: Union[pandagg.types.BucketsWrapperDict, Dict[str, Any]]*) → Iterator[Tuple[Union[None, str, float, Dict[str, Union[str, float, None]]], Dict[str, Any]]]

class pandagg.node.aggs.abstract.**MultipleBucketAgg**(*keyed: bool = False*, *key_as_string: bool = False*, ***body*)

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

IMPLICIT_KEYED = False

extract_buckets(*response_value: Union[pandagg.types.BucketsWrapperDict, Dict[str, Any]]*) → Iterator[Tuple[Union[None, str, float, Dict[str, Union[str, float, None]]], Dict[str, Any]]]

class pandagg.node.aggs.abstract.**Pipeline**(*buckets_path: str*, *gap_policy: Optional[typing_extensions.Literal['skip', 'insert_zeros', 'keep_values']][skip, insert_zeros, keep_values]] = None*, ***body*)

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

class pandagg.node.aggs.abstract.**Root**(*meta: Optional[Dict[str, Any]] = None*, *identifier: Optional[str] = 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: Union[pandagg.types.BucketsWrapperDict, Dict[str, Any]]*, *value_as_dict: bool = False*) → Any

extract_buckets(*response_value: Union[pandagg.types.BucketsWrapperDict, Dict[str, Any]]*) → Iterator[Tuple[Union[None, str, float, Dict[str, Union[str, float, None]]], Dict[str, Any]]]

line_repr(*depth: int*, ***kwargs*) → Tuple[str, str]

Control how node is displayed in tree representation. First returned string is how node is represented on left, second string is how node is represented on right.

MyTree ├── one OneEnd │ └── two twoEnd └── three threeEnd

class pandagg.node.aggs.abstract.**ScriptPipeline**(*script: pandagg.types.Script*, *buckets_path: str*, *gap_policy: Optional[typing_extensions.Literal['skip', 'insert_zeros', 'keep_values']][skip, insert_zeros, keep_values]] = None*, ***body*)

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

VALUE_ATTRS = ['value']

class pandagg.node.aggs.abstract.**UniqueBucketAgg**(***body*)

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

Aggregations providing a single bucket.

extract_buckets(*response_value: Union[pandagg.types.BucketsWrapperDict, Dict[str, Any]]*) → Iterator[Tuple[Union[None, str, float, Dict[str, Union[str, float, None]]], Dict[str, Any]]]

pandagg.node.aggs.bucket module

class pandagg.node.aggs.bucket.**AdjacencyMatrix**(*filters: Dict[str, Dict[str, Dict[str, Any]]]*, *separator: Optional[str] = None*, ***body*)

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

KEY = 'adjacency_matrix'

VALUE_ATTRS = ['doc_count']

class pandagg.node.aggs.bucket.**AutoDateHistogram**(*field: str*, *buckets: Optional[int] = None*, *format: Optional[str] = None*, *time_zone: Optional[str] = None*, *minimum_interval: Optional[str] = None*, *missing: Optional[str] = None*, *key_as_string: bool = True*, ***body*)

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

KEY = 'auto_date_histogram'

VALUE_ATTRS = ['doc_count']

class pandagg.node.aggs.bucket.**Children**(*type: str*, ***body*)

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

KEY = 'children'

VALUE_ATTRS = ['doc_count']

class pandagg.node.aggs.bucket.**DateHistogram**(*field: str*, *interval: str = None*, *calendar_interval: str = None*, *fixed_interval: str = None*, *key_as_string: bool = True*, ***body*)

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

KEY = 'date_histogram'

VALUE_ATTRS = ['doc_count']

WHITELISTED_MAPPING_TYPES = ['date']

is_convertible_to_composite_source() → bool

class pandagg.node.aggs.bucket.**DateRange**(*field: str, ranges: List[pandagg.types.RangeDict], keyed: bool = False, **body*)

Bases: *pandagg.node.aggs.bucket.Range*

KEY = 'date_range'

VALUE_ATTRS = ['doc_count']

WHITELISTED_MAPPING_TYPES = ['date']

class pandagg.node.aggs.bucket.**DiversifiedSampler**(*field: str, shard_size: Optional[int], max_docs_per_value: Optional[int] = None, execution_hint: Optional[typing_extensions.Literal['map', 'global_ordinals', 'bytes_hash']][map, global_ordinals, bytes_hash]] = None, **body*)

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

KEY = 'diversified_sampler'

VALUE_ATTRS = ['doc_count']

class pandagg.node.aggs.bucket.**Filter**(*filter: Optional[Dict[str, Dict[str, Any]]] = None*, *meta: Optional[Dict[str, Any]] = None*, ***body*)

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

KEY = 'filter'

VALUE_ATTRS = ['doc_count']

class pandagg.node.aggs.bucket.**Filters**(*filters: Dict[str, Dict[str, Dict[str, Any]]]*, *other_bucket: bool = False*, *other_bucket_key: Optional[str] = None*, ***body*)

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

DEFAULT_OTHER_KEY = '_other_'

IMPLICIT_KEYED = True

KEY = 'filters'

VALUE_ATTRS = ['doc_count']

class pandagg.node.aggs.bucket.**GeoDistance**(*field: str, origin: str, ranges: List[pandagg.types.RangeDict], unit: Optional[str] = None, distance_type: Optional[typing_extensions.Literal['arc', 'plane']][arc, plane]] = None, keyed: bool = False, **body*)

Bases: *pandagg.node.aggs.bucket.Range*

KEY = 'geo_distance'

VALUE_ATTRS = ['doc_count']

WHITELISTED_MAPPING_TYPES = ['geo_point']

class pandagg.node.aggs.bucket.**GeoHashGrid**(*field: str, precision: Optional[int] = None, bounds: Optional[Dict[KT, VT]] = None, size: Optional[int] = None, shard_size: Optional[int] = None, **body*)

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

KEY = 'geohash_grid'

VALUE_ATTRS = ['doc_count']

WHITELISTED_MAPPING_TYPES = ['geo_point', 'geo_shape']

class pandagg.node.aggs.bucket.**GeoTileGrid**(*field: str, precision: Optional[int] = None, bounds: Optional[Dict[KT, VT]] = None, size: Optional[int] = None, shard_size: Optional[int] = None, **body*)

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

KEY = 'geotile_grid'

VALUE_ATTRS = ['doc_count']

WHITELISTED_MAPPING_TYPES = ['geo_point', 'geo_shape']

class pandagg.node.aggs.bucket.**Global**(***body*)

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

KEY = 'global'

VALUE_ATTRS = ['doc_count']

class pandagg.node.aggs.bucket.**Histogram**(*field: str, interval: int, **body*)

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

KEY = 'histogram'

VALUE_ATTRS = ['doc_count']

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

is_convertible_to_composite_source() → bool

class pandagg.node.aggs.bucket.**IPRange**(*field: str, ranges: List[pandagg.types.RangeDict], keyed: bool = False, **body*)

Bases: *pandagg.node.aggs.bucket.Range*

KEY = 'ip_range'

VALUE_ATTRS = ['doc_count']

WHITELISTED_MAPPING_TYPES = ['ip']

class pandagg.node.aggs.bucket.**MatchAll**(***body*)
Bases: *pandagg.node.aggs.bucket.Filter*

class pandagg.node.aggs.bucket.**Missing**(*field: str*, ***body*)
Bases: *pandagg.node.aggs.abstract.UniqueBucketAgg*

KEY = 'missing'

VALUE_ATTRS = ['doc_count']

class pandagg.node.aggs.bucket.**MultiTerms**(*terms: List[Dict[KT, VT]]*, ***body*)
Bases: *pandagg.node.aggs.abstract.MultipleBucketAgg*

KEY = 'multi_terms'

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

class pandagg.node.aggs.bucket.**Nested**(*path: str*, ***body*)
Bases: *pandagg.node.aggs.abstract.UniqueBucketAgg*

KEY = 'nested'

VALUE_ATTRS = ['doc_count']

WHITELISTED_MAPPING_TYPES = ['nested']

class pandagg.node.aggs.bucket.**Parent**(*type: str*, ***body*)
Bases: *pandagg.node.aggs.abstract.UniqueBucketAgg*

KEY = 'parent'

VALUE_ATTRS = ['doc_count']

class pandagg.node.aggs.bucket.**Range**(*field: str, ranges: List[pandagg.types.RangeDict], keyed: bool = False, **body*)
Bases: *pandagg.node.aggs.abstract.MultipleBucketAgg*

KEY = 'range'

VALUE_ATTRS = ['doc_count']

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

class pandagg.node.aggs.bucket.**RareTerms**(*field: str, max_doc_count: Optional[int] = None, precision: Optional[float] = None, include: Union[str, List[str], None] = None, exclude: Union[str, List[str], None] = None, missing: Optional[Any] = None, **body*)
Bases: *pandagg.node.aggs.abstract.MultipleBucketAgg*

KEY = 'rare_terms'

VALUE_ATTRS = ['doc_count']

class pandagg.node.aggs.bucket.**ReverseNested**(*path: Optional[str] = None*, ***body*)
Bases: *pandagg.node.aggs.abstract.UniqueBucketAgg*

KEY = 'reverse_nested'

VALUE_ATTRS = ['doc_count']

WHITELISTED_MAPPING_TYPES = ['nested']

class pandagg.node.aggs.bucket.**Sampler**(*shard_size: Optional[int] = None*, ***body*)
Bases: *pandagg.node.aggs.abstract.UniqueBucketAgg*

KEY = 'sampler'

VALUE_ATTRS = ['doc_count']

class pandagg.node.aggs.bucket.**SignificantTerms**(*field: str*, ***body*)
Bases: *pandagg.node.aggs.abstract.MultipleBucketAgg*

KEY = 'significant_terms'

VALUE_ATTRS = ['doc_count', 'score', 'bg_count']

class pandagg.node.aggs.bucket.**SignificantText**(*field: str*, ***body*)
Bases: *pandagg.node.aggs.abstract.MultipleBucketAgg*

KEY = 'significant_text'

VALUE_ATTRS = ['doc_count', 'score', 'bg_count']

WHITELISTED_MAPPING_TYPES = ['text']

class pandagg.node.aggs.bucket.**Terms**(*field: str*, *missing: Union[str, int, None] = None*, *size: Optional[int] = None*, ***body*)
Bases: *pandagg.node.aggs.abstract.MultipleBucketAgg*

Terms aggregation.

KEY = 'terms'

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

is_convertible_to_composite_source() → bool

class pandagg.node.aggs.bucket.**VariableWidthHistogram**(*field: str*, *buckets: int*, ***body*)
Bases: *pandagg.node.aggs.abstract.MultipleBucketAgg*

KEY = 'variable_width_histogram'

VALUE_ATTRS = ['doc_count', 'min', 'max']

pandagg.node.aggs.composite module

class pandagg.node.aggs.composite.**Composite**(*sources: List[Dict[str, Dict[str, Dict[str, Any]]]]*, *size: Optional[int] = None*, *after: Optional[Dict[str, Any]] = None*, ***body*)
Bases: *pandagg.node.aggs.abstract.BucketAggClause*

KEY = 'composite'

VALUE_ATTRS = ['doc_count']

after

extract_buckets(*response_value: Union[pandagg.types.BucketsWrapperDict, Dict[str, Any]]*) → Iterator[Tuple[Dict[str, Union[str, float, None]], Dict[str, Any]]]

size

source_names

sources

pandagg.node.aggs.metric module

class pandagg.node.aggs.metric.**Avg**(*field: Optional[str] = None, script: Optional[pandagg.types.Script] = 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: Optional[str] = None, script: Optional[pandagg.types.Script] = None, **body*)

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

KEY = 'cardinality'

VALUE_ATTRS = ['value']

class pandagg.node.aggs.metric.**ExtendedStats**(*field: Optional[str] = None, script: Optional[pandagg.types.Script] = 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: Optional[str] = None, script: Optional[pandagg.types.Script] = 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: Optional[str] = None, script: Optional[pandagg.types.Script] = 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: Optional[str] = None, script: Optional[pandagg.types.Script] = 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: Optional[str] = None, script: Optional[pandagg.types.Script] = 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: str, values: List[float], **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: Optional[str] = None, script: Optional[pandagg.types.Script] = 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: Optional[str] = None, script: Optional[pandagg.types.Script] = 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: Optional[str] = None, script: Optional[pandagg.types.Script] = 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: Optional[Dict[str, Any]] = None, identifier: Optional[str] = None, **body*)

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

KEY = 'top_hits'

VALUE_ATTRS = ['hits']

class pandagg.node.aggs.metric.**ValueCount**(*field: Optional[str] = None, script: Optional[pandagg.types.Script] = None, **body*)

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

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: str, gap_policy: Optional[typing_extensions.Literal['skip', 'insert_zeros', 'keep_values']][skip, insert_zeros, keep_values]] = None, **body*)

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

KEY = 'avg_bucket'

VALUE_ATTRS = ['value']

class pandagg.node.aggs.pipeline.**BucketScript**(*script: pandagg.types.Script, buckets_path: str, gap_policy: Optional[typing_extensions.Literal['skip', 'insert_zeros', 'keep_values']][skip, insert_zeros, keep_values]] = None, **body*)

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

KEY = 'bucket_script'

VALUE_ATTRS = ['value']

class pandagg.node.aggs.pipeline.**BucketSelector**(*script: pandagg.types.Script, buckets_path: str, gap_policy: Optional[typing_extensions.Literal['skip', 'insert_zeros', 'keep_values']][skip, insert_zeros, keep_values]] = None, **body*)

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

KEY = 'bucket_selector'

VALUE_ATTRS = []

class pandagg.node.aggs.pipeline.**BucketSort**(*script: pandagg.types.Script, buckets_path: str, gap_policy: Optional[typing_extensions.Literal['skip', 'insert_zeros', 'keep_values']][skip, insert_zeros, keep_values]] = None, **body*)

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

KEY = 'bucket_sort'

VALUE_ATTRS = []

class pandagg.node.aggs.pipeline.**CumulativeSum**(*buckets_path: str, gap_policy: Optional[typing_extensions.Literal['skip', 'insert_zeros', 'keep_values']][skip, insert_zeros, keep_values]] = None, **body*)

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

KEY = 'cumulative_sum'

VALUE_ATTRS = ['value']

class pandagg.node.aggs.pipeline.**Derivative**(*buckets_path: str, gap_policy: Optional[typing_extensions.Literal['skip', 'insert_zeros', 'keep_values']][skip, insert_zeros, keep_values]] = None, **body*)

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

KEY = 'derivative'

VALUE_ATTRS = ['value']

class pandagg.node.aggs.pipeline.**ExtendedStatsBucket**(*buckets_path: str, gap_policy: Optional[typing_extensions.Literal['skip', 'insert_zeros', 'keep_values']][skip, insert_zeros, keep_values]] = 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: str, gap_policy: Optional[typing_extensions.Literal['skip', 'insert_zeros', 'keep_values']][skip, insert_zeros, keep_values]] = None, **body*)

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

KEY = 'max_bucket'

VALUE_ATTRS = ['value']

class pandagg.node.aggs.pipeline.**MinBucket**(*buckets_path: str, gap_policy: Optional[typing_extensions.Literal['skip', 'insert_zeros', 'keep_values']][skip, insert_zeros, keep_values]] = None, **body*)

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

KEY = 'min_bucket'

VALUE_ATTRS = ['value']

class pandagg.node.aggs.pipeline.**MovingAvg**(*buckets_path: str, gap_policy: Optional[typing_extensions.Literal['skip', 'insert_zeros', 'keep_values']][skip, insert_zeros, keep_values]] = None, **body*)

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

KEY = 'moving_avg'

VALUE_ATTRS = ['value']

class pandagg.node.aggs.pipeline.**PercentilesBucket**(*buckets_path: str, gap_policy: Optional[typing_extensions.Literal['skip', 'insert_zeros', 'keep_values']][skip, insert_zeros, keep_values]] = None, **body*)

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

KEY = 'percentiles_bucket'

VALUE_ATTRS = ['values']

class pandagg.node.aggs.pipeline.**SerialDiff**(*buckets_path: str, gap_policy: Optional[typing_extensions.Literal['skip', 'insert_zeros', 'keep_values']][skip, insert_zeros, keep_values]] = None, **body*)

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

KEY = 'serial_diff'

VALUE_ATTRS = ['value']

class pandagg.node.aggs.pipeline.**StatsBucket**(*buckets_path: str, gap_policy: Optional[typing_extensions.Literal['skip', 'insert_zeros', 'keep_values']][skip, insert_zeros, keep_values]] = 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: str, gap_policy: Optional[typing_extensions.Literal['skip', 'insert_zeros', 'keep_values']][skip, insert_zeros, keep_values]] = 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**(*properties: Optional[Union[Dict, Type[DocumentSource]]] = None, **body*)

Bases: *pandagg.node.mappings.abstract.Field*

is_valid_value(*v: Any*) → bool

class pandagg.node.mappings.abstract.**Field**(*, *multiple: Optional[bool] = None, required: bool = False, **body*)

Bases: pandagg.node._node.Node

is_valid_value(*v: Any*) → bool

line_repr(*depth: int, **kwargs*) → Tuple[str, str]

Control how node is displayed in tree representation. First returned string is how node is represented on left, second string is how node is represented on right.

MyTree ├── one OneEnd │ └── two twoEnd └── three threeEnd

to_dict() → Dict[str, Any]

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

is_valid_value(*v: Any*) → bool

class pandagg.node.mappings.abstract.**Root**(*, *multiple: Optional[bool] = None*, *required: bool = False*, **body)
Bases: *pandagg.node.mappings.abstract.Field*

KEY = ''

line_repr(*depth: int*, ***kwargs*) → Tuple[str, str]
Control how node is displayed in tree representation. First returned string is how node is represented on left, second string is how node is represented on right.

MyTree ├── one OneEnd │ └── two twoEnd └── three threeEnd

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

KEY = 'ip_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**(*properties: Optional[Union[Dict, Type[DocumentSource]]] = None*, ***body*)
Bases: *pandagg.node.mappings.abstract.ComplexField*

KEY = 'nested'

class pandagg.node.mappings.field_datatypes.**Object**(*properties: Optional[Union[Dict, Type[DocumentSource]]] = None*, ***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: Optional[bool] = None*, *required: bool = False*, **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: Optional[bool] = None*, *required: bool = False*, **body*)
Bases: *pandagg.node.mappings.abstract.Field*

The document's ID.

KEY = '_id'

class pandagg.node.mappings.meta_fields.**Ignored**(*, *multiple: Optional[bool] = None*, *required: bool = False*, **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: Optional[bool] = None*, *required: bool = False*, **body*)
Bases: *pandagg.node.mappings.abstract.Field*

The index to which the document belongs.

KEY = '_index'

class pandagg.node.mappings.meta_fields.**Meta**(*, *multiple: Optional[bool] = None*, *required: bool = False*, **body*)
Bases: *pandagg.node.mappings.abstract.Field*

Application specific metadata.

KEY = '_meta'

class pandagg.node.mappings.meta_fields.**Routing**(*, *multiple: Optional[bool] = None*, *required: bool = False*, **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: Optional[bool] = None*, *required: bool = False*, **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: Optional[bool] = None*, *required: bool = False*, ***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: Optional[bool] = None*, *required: bool = False*, ***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: str*, *_name: Optional[str] = None*, ***body*)

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

class pandagg.node.query.abstract.**FlatFieldQueryClause**(*field: str*, *_name: Optional[str] = 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: Optional[str] = None*, *_name: Optional[str] = None*, *_expand__to_dot: bool = 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: int*, ***kwargs*) → Tuple[str, str]

Control how node is displayed in tree representation. First returned string is how node is represented on left, second string is how node is represented on right.

MyTree ├── one OneEnd │ └── two twoEnd └── three threeEnd

class pandagg.node.query.abstract.**LeafQueryClause**(*_name: Optional[str] = None*, ***body*)

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

class pandagg.node.query.abstract.**MultiFieldsQueryClause**(*fields: List[str]*, *_name: Optional[str] = None*, ***body*)

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

line_repr(*depth: int*, ***kwargs*) → Tuple[str, str]

Control how node is displayed in tree representation. First returned string is how node is represented on left, second string is how node is represented on right.

MyTree ├── one OneEnd │ └── two twoEnd └── three threeEnd

class pandagg.node.query.abstract.**ParentParameterClause**

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

line_repr(*depth: int*, ***kwargs*) → Tuple[str, str]

Control how node is displayed in tree representation. First returned string is how node is represented on left, second string is how node is represented on right.

MyTree ├── one OneEnd │ └── two twoEnd └── three threeEnd

pandagg.node.query.abstract.**Q**(*type_or_query: Union[str, Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause, None] = None*, ***body*) → pandagg.node.query.abstract.QueryClause

Accept multiple syntaxes, return a QueryClause node.

Parameters

- **type_or_query** –
- **body** –

Returns QueryClause

class pandagg.node.query.abstract.**QueryClause**(*_name: Optional[str] = None*, *accept_children: bool = True*, *keyed: bool = True*, *_children: Any = None*, ***body*)

Bases: pandagg.node._node.Node

line_repr(*depth: int*, ***kwargs*) → Tuple[str, str]

Control how node is displayed in tree representation. First returned string is how node is represented on left, second string is how node is represented on right.

MyTree ├── one OneEnd │ └── two twoEnd └── three threeEnd

name

to_dict() → Dict[str, Any]

pandagg.node.query.compound module

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

KEY = 'boosting'

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

Compound clauses can encapsulate other query clauses:

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

KEY = 'constant_score'

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

KEY = 'dis_max'

class pandagg.node.query.compound.**FunctionScore**(*_name: Optional[str] = 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: Optional[str] = None*, *_name: Optional[str] = None*, *_expand__to_dot: bool = True*, ***params*)
Bases: *pandagg.node.query.abstract.KeyFieldQueryClause*

KEY = 'common'

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

KEY = 'intervals'

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

KEY = 'match'

class pandagg.node.query.full_text.**MatchBoolPrefix**(*field: Optional[str] = None, _name: Optional[str] = None, _expand__to_dot: bool = True, **params*)

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

KEY = 'match_bool_prefix'

class pandagg.node.query.full_text.**MatchPhrase**(*field: Optional[str] = None, _name: Optional[str] = None, _expand__to_dot: bool = True, **params*)

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

KEY = 'match_phrase'

class pandagg.node.query.full_text.**MatchPhrasePrefix**(*field: Optional[str] = None, _name: Optional[str] = None, _expand__to_dot: bool = True, **params*)

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

KEY = 'match_phrase_prefix'

class pandagg.node.query.full_text.**MultiMatch**(*fields: List[str], _name: Optional[str] = None, **body*)

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

KEY = 'multi_match'

class pandagg.node.query.full_text.**QueryString**(*_name: Optional[str] = None, **body*)

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

KEY = 'query_string'

class pandagg.node.query.full_text.**SimpleQueryString**(*_name: Optional[str] = None, **body*)

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

KEY = 'simple_string'

pandagg.node.query.geo module

class pandagg.node.query.geo.**GeoBoundingBox**(*field: Optional[str] = None, _name: Optional[str] = None, _expand__to_dot: bool = True, **params*)

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

KEY = 'geo_bounding_box'

class pandagg.node.query.geo.**GeoDistance**(*distance: str, **body*)

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

KEY = 'geo_distance'

line_repr(*depth: int, **kwargs*) → Tuple[str, str]

Control how node is displayed in tree representation. First returned string is how node is represented on left, second string is how node is represented on right.

MyTree ├── one OneEnd │ └── two twoEnd └── three threeEnd

class pandagg.node.query.geo.**GeoPolygone**(*field: Optional[str] = None*, *_name: Optional[str] = None*, *_expand__to_dot: bool = True*, ***params*)

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

KEY = 'geo_polygon'

class pandagg.node.query.geo.**GeoShape**(*field: Optional[str] = None*, *_name: Optional[str] = None*, *_expand__to_dot: bool = True*, ***params*)

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

KEY = 'geo_shape'

pandagg.node.query.joining module

class pandagg.node.query.joining.**HasChild**(*_name: Optional[str] = None*, ***body*)

Bases: *pandagg.node.query.compound.CompoundClause*

KEY = 'has_child'

class pandagg.node.query.joining.**HasParent**(*_name: Optional[str] = None*, ***body*)

Bases: *pandagg.node.query.compound.CompoundClause*

KEY = 'has_parent'

class pandagg.node.query.joining.**Nested**(*path: str*, ***body*)

Bases: *pandagg.node.query.compound.CompoundClause*

KEY = 'nested'

class pandagg.node.query.joining.**ParentId**(*_name: Optional[str] = None*, ***body*)

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

KEY = 'parent_id'

pandagg.node.query.shape module

class pandagg.node.query.shape.**Shape**(*_name: Optional[str] = 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: str*, *_name: Optional[str] = None*, ***body*)

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

KEY = 'distance_feature'

class pandagg.node.query.specialized.**MoreLikeThis**(*fields: List[str]*, *_name: Optional[str] = None*, ***body*)

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

KEY = 'more_like_this'

class pandagg.node.query.specialized.**Percolate**(*field: str*, *_name: Optional[str] = None*, ***body*)

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

KEY = 'percolate'

class pandagg.node.query.specialized.**RankFeature**(*field: str*, *_name: Optional[str] = None*, ***body*)

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

KEY = 'rank_feature'

class pandagg.node.query.specialized.**Script**(*_name: Optional[str] = None*, ***body*)

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

KEY = 'script'

class pandagg.node.query.specialized.**Wrapper**(*_name: Optional[str] = None*, ***body*)

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

KEY = 'wrapper'

pandagg.node.query.specialized_compound module

class pandagg.node.query.specialized_compound.**PinnedQuery**(*_name: Optional[str] = None*, ***body*)

Bases: *pandagg.node.query.compound.CompoundClause*

KEY = 'pinned'

class pandagg.node.query.specialized_compound.**ScriptScore**(*_name: Optional[str] = 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: str*, *_name: Optional[str] = None*)

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

KEY = 'exists'

line_repr(*depth: int*, ***kwargs*) → Tuple[str, str]

Control how node is displayed in tree representation. First returned string is how node is represented on left, second string is how node is represented on right.

MyTree ├── one OneEnd │ └── two twoEnd └── three threeEnd

class pandagg.node.query.term_level.**Fuzzy**(*field: Optional[str] = None*, *_name: Optional[str] = None*, *_expand__to_dot: bool = True*, ***params*)

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

KEY = 'fuzzy'

class pandagg.node.query.term_level.**Ids**(*values: List[Union[str, int]]*, *_name: Optional[str] = None*)

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

KEY = 'ids'

line_repr(*depth: int*, ***kwargs*) → Tuple[str, str]
Control how node is displayed in tree representation. First returned string is how node is represented on left, second string is how node is represented on right.

MyTree ├── one OneEnd │ └── two twoEnd └── three threeEnd

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

KEY = 'prefix'

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

KEY = 'range'

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

KEY = 'regexp'

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

KEY = 'terms_set'

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

KEY = 'type'

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

KEY = 'wildcard'

Module contents

pandagg.node.response package

Submodules

pandagg.node.response.bucket module

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: Union[Dict[str, Union[Dict[str, Dict[str, Any]], pandagg.node.aggs.abstract.AggClause]], Aggs, None] = None, mappings: Union[pandagg.types.MappingsDict, Mappings, None] = None, nested_autocorrect: bool = False, _groupby_ptr: Optional[str] = None*)

Bases: pandagg.tree._tree.TreeReprMixin, lighttree.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: str, type_or_agg: Union[str, Dict[str, Dict[str, Any]], pandagg.node.aggs.abstract.AggClause, None] = None, insert_below: Optional[str] = None, at_root: bool = False, **body*) → pandagg.tree.aggs.Aggs

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: Union[Dict[str, Union[Dict[str, Dict[str, Any]], pandagg.node.aggs.abstract.AggClause]], Aggs], insert_below: Optional[str] = None, at_root: bool = False*) → pandagg.tree.aggs.Aggs

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: str*) → Optional[str]

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: Optional[str] = None*) → None

as_composite(*size: int*, *after: Optional[Dict[str*, *Any]] = None*) → pandagg.tree.aggs.Aggs

Convert current aggregation into composite aggregation. For now, simply support conversion of the root aggregation clause, and doesn't handle multi-source.

get_composition_supporting_agg() → Tuple[str, pandagg.node.aggs.abstract.AggClause]

Return first composite-compatible aggregation clause if possible, raise an error otherwise.

groupby(*name: str, type_or_agg: Union[str, Dict[str, Dict[str, Any]], pandagg.node.aggs.abstract.AggClause, None] = None, insert_below: Optional[str] = None, at_root: bool = False, **body*) → pandagg.tree.aggs.Aggs

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: Optional[str] = None*, *deepest: bool = False*) → pandagg.tree.aggs.Aggs

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.

id_from_key(*key: str*) → str

Find node identifier based on key. If multiple nodes have the same key, takes the first one.

Useful because of how pandagg implements lighttree.Tree. A bit of context:

ElasticSearch allows queries to contain multiple similarly named clauses (for queries and aggregations). As a consequence clauses names are not used as clauses identifier in Trees, and internally pandagg (as lighttree) uses auto-generated uuids to distinguish them.

But for usability reasons, notably when declaring that an aggregation clause must be placed relatively to another one, the latter is identified by its name rather than its internal id. Since it is technically possible that multiple clauses share the same name (not recommended, but allowed), some pandagg features are ambiguous and not recommended in such context.

show(**args*, *line_max_length: int = 80*, ***kwargs*) → str

Return compact representation of Aggs.

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

(continues on next page)

(continued from previous page)

```
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_: Optional[str] = None*, *depth: Optional[int] = None*) → Dict[str, Dict[str, Dict[str, Any]]]

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: Optional[Dict[str, Union[Dict[str, Any], pandagg.node.mappings.abstract.Field]]] = None*, *dynamic: Optional[bool] = None*, ***body*)

Bases: pandagg.tree._tree.TreeReprMixin, lighttree.tree.Tree

list_nesteds_at_field(*field_path: str*) → List[str]

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: str*) → str

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: str*) → Optional[str]
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'
```

to_dict(*from_: Optional[str] = None*, *depth: Optional[int] = None*) → pandagg.types.MappingsDict
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: pandagg.node.aggs.abstract.AggClause*, *exc: bool = True*) → bool
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: Union[DocSource, DocumentSource]*) → None

class pandagg.tree.mappings.**MappingsDictOrNode**
Bases: dict

pandagg.tree.query module

class pandagg.tree.query.**Query**(*q: Union[str, Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause, Query, None] = None*, *mappings: Union[pandagg.types.MappingsDict, pandagg.tree.mappings.Mappings, None] = None*, *nested_autocorrect: bool = False*)
Bases: lighttree.tree.Tree

applied_nested_path_at_node(*nid: str*) → Optional[str]
Return nested path applied at a clause.

Parameters **nid** – clause identifier

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

bool(*must: Union[Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause, List[Union[Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause]], None] = None, should: Union[Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause, List[Union[Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause]], None] = None, must_not: Union[Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause, List[Union[Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause]], None] = None, filter: Union[Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause, List[Union[Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause]], None] = None, insert_below: Optional[str] = None, on: Optional[str] = None, mode: typing_extensions.Literal['add', 'replace', 'replace_all'][add, replace, replace_all] = 'add', **body*) → pandagg.tree.query.Query

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

boosting(*positive: Union[Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause, None] = None, negative: Union[Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause, None] = None, insert_below: Optional[str] = None, on: Optional[str] = None, mode: typing_extensions.Literal['add', 'replace', 'replace_all'][add, replace, replace_all] = 'add', **body*) → pandagg.tree.query.Query

constant_score(*filter: Union[Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause, None] = None, boost: Optional[float] = None, insert_below: Optional[str] = None, on: Optional[str] = None, mode: typing_extensions.Literal['add', 'replace', 'replace_all'][add, replace, replace_all] = 'add', **body*) → pandagg.tree.query.Query

dis_max(*queries: List[Union[Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause]], insert_below: Optional[str] = None, on: Optional[str] = None, mode: typing_extensions.Literal['add', 'replace', 'replace_all'][add, replace, replace_all] = 'add', **body*) → pandagg.tree.query.Query

filter(*type_or_query: Union[str, Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause, Query], insert_below: Optional[str] = None, on: Optional[str] = None, mode: typing_extensions.Literal['add', 'replace', 'replace_all'][add, replace, replace_all] = 'add', bool_body: Dict[str, Any] = None, **body*) → pandagg.tree.query.Query

function_score(*query: Union[Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause, None], insert_below: Optional[str] = None, on: Optional[str] = None, mode: typing_extensions.Literal['add', 'replace', 'replace_all'][add, replace, replace_all] = 'add', **body*) → pandagg.tree.query.Query

has_child(*query: Union[Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause, None], insert_below: Optional[str] = None, on: Optional[str] = None, mode: typing_extensions.Literal['add', 'replace', 'replace_all'][add, replace, replace_all] = 'add', **body*) → pandagg.tree.query.Query

has_parent(*query: Union[Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause, None], insert_below: Optional[str] = None, on: Optional[str] = None, mode: typing_extensions.Literal['add', 'replace', 'replace_all'][add, replace, replace_all] = 'add', **body*) → pandagg.tree.query.Query

must(*type_or_query: Union[str, Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause, Query], insert_below: Optional[str] = None, on: Optional[str] = None, mode: typing_extensions.Literal['add', 'replace', 'replace_all'][add, replace, replace_all] = 'add', bool_body: Optional[Dict[str, Any]] = None, **body*) → pandagg.tree.query.Query

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: Union[str, Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause, Query], insert_below: Optional[str] = None, on: Optional[str] = None, mode: typing_extensions.Literal['add', 'replace', 're-place_all']['add, replace, replace_all] = 'add', bool_body: Dict[str, Any] = None, **body*) → pandagg.tree.query.Query

nested(*path: str, query: Union[Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause, None] = None, insert_below: Optional[str] = None, on: Optional[str] = None, mode: typing_extensions.Literal['add', 'replace', 'replace_all']['add, replace, replace_all] = 'add', **body*) → pandagg.tree.query.Query

pinned_query(*organic: Union[Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause, None], insert_below: Optional[str] = None, on: Optional[str] = None, mode: typ-ing_extensions.Literal['add', 'replace', 'replace_all']['add, replace, replace_all] = 'add', **body*) → pandagg.tree.query.Query

query(*type_or_query: Union[str, Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause, Query], insert_below: Optional[str] = None, on: Optional[str] = None, mode: typ-ing_extensions.Literal['add', 'replace', 'replace_all']['add, replace, replace_all] = 'add', com-pound_param: str = None, **body*) → pandagg.tree.query.Query

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: Union[Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause, None], insert_below: Optional[str] = None, on: Optional[str] = None, mode: typing_extensions.Literal['add', 'replace', 'replace_all'][add, replace, replace_all] = 'add', **body*) → pandagg.tree.query.Query

should(*type_or_query: Union[str, Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause, Query], insert_below: Optional[str] = None, on: Optional[str] = None, mode: typing_extensions.Literal['add', 'replace', 'replace_all'][add, replace, replace_all] = 'add', bool_body: Optional[Dict[str, Any]] = None, **body*) → pandagg.tree.query.Query

show(**args*, *line_max_length: int = 80*, ***kwargs*) → str

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.

to_dict(*from_: Optional[str] = None*) → Optional[Dict[str, Dict[str, Any]]]

pandagg.tree.response module

4.1.3.2 Module contents

4.2 Submodules

4.2.1 pandagg.aggs module

class pandagg.aggs.**Aggs**(*aggs: Union[Dict[str, Union[Dict[str, Dict[str, Any]], pandagg.node.aggs.abstract.AggClause]], Aggs, None] = None, mappings: Union[pandagg.types.MappingsDict, Mappings, None] = None, nested_autocorrect: bool = False, _groupby_ptr: Optional[str] = None*)

Bases: `pandagg.tree._tree.TreeReprMixin`, `lighttree.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: str, type_or_agg: Union[str, Dict[str, Dict[str, Any]], pandagg.node.aggs.abstract.AggClause, None] = None, insert_below: Optional[str] = None, at_root: bool = False, **body*) → pandagg.tree.aggs.Aggs

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: Union[Dict[str, Union[Dict[str, Dict[str, Any]], pandagg.node.aggs.abstract.AggClause]], Aggs], insert_below: Optional[str] = None, at_root: bool = False*) → pandagg.tree.aggs.Aggs

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: str*) → Optional[str]
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: Optional[str] = None*) → None

as_composite (*size: int*, *after: Optional[Dict[str, Any]] = None*) → pandagg.tree.aggs.Aggs
Convert current aggregation into composite aggregation. For now, simply support conversion of the root aggregation clause, and doesn't handle multi-source.

get_composition_supporting_agg () → Tuple[str, pandagg.node.aggs.abstract.AggClause]
Return first composite-compatible aggregation clause if possible, raise an error otherwise.

groupby (*name: str, type_or_agg: Union[str, Dict[str, Dict[str, Any]], pandagg.node.aggs.abstract.AggClause, None] = None, insert_below: Optional[str] = None, at_root: bool = False, **body*) → pandagg.tree.aggs.Aggs
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: Optional[str] = None*, *deepest: bool = False*) → pandagg.tree.aggs.Aggs
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.

id_from_key (*key: str*) → str
Find node identifier based on key. If multiple nodes have the same key, takes the first one.

Useful because of how pandagg implements lighttree.Tree. A bit of context:

ElasticSearch allows queries to contain multiple similarly named clauses (for queries and aggregations). As a consequence clauses names are not used as clauses identifier in Trees, and internally pandagg (as lighttree) uses auto-generated uuids to distinguish them.

But for usability reasons, notably when declaring that an aggregation clause must be placed relatively to another one, the latter is identified by its name rather than its internal id. Since it is technically possible that multiple clauses share the same name (not recommended, but allowed), some pandagg features are ambiguous and not recommended in such context.

show(**args*, *line_max_length: int = 80*, ***kwargs*) → str

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_: Optional[str] = None*, *depth: Optional[int] = None*) → Dict[str, Dict[str, Dict[str, Any]]]

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: str*, *missing: Union[str, int, None] = None*, *size: Optional[int] = None*, ***body*)

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

Terms aggregation.

KEY = 'terms'

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

is_convertible_to_composite_source() → bool

class pandagg.aggs.**Filters**(*filters: Dict[str, Dict[str, Dict[str, Any]]]*, *other_bucket: bool = False*, *other_bucket_key: Optional[str] = None*, ***body*)

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

DEFAULT_OTHER_KEY = '_other_'

IMPLICIT_KEYED = True

KEY = 'filters'

VALUE_ATTRS = ['doc_count']

class pandagg.aggs.**Histogram**(*field: str*, *interval: int*, ***body*)
Bases: *pandagg.node.aggs.abstract.MultipleBucketAgg*

KEY = 'histogram'

VALUE_ATTRS = ['doc_count']

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

is_convertible_to_composite_source() → bool

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

KEY = 'date_histogram'

VALUE_ATTRS = ['doc_count']

WHITELISTED_MAPPING_TYPES = ['date']

is_convertible_to_composite_source() → bool

class pandagg.aggs.**Range**(*field: str, ranges: List[pandagg.types.RangeDict], keyed: bool = False, **body*)
Bases: *pandagg.node.aggs.abstract.MultipleBucketAgg*

KEY = 'range'

VALUE_ATTRS = ['doc_count']

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

class pandagg.aggs.**Global**(***body*)
Bases: *pandagg.node.aggs.abstract.UniqueBucketAgg*

KEY = 'global'

VALUE_ATTRS = ['doc_count']

class pandagg.aggs.**Filter**(*filter: Optional[Dict[str, Dict[str, Any]]] = None*, *meta: Optional[Dict[str, Any]] = None*, ***body*)
Bases: *pandagg.node.aggs.abstract.UniqueBucketAgg*

KEY = 'filter'

VALUE_ATTRS = ['doc_count']

class pandagg.aggs.**Missing**(*field: str*, ***body*)
Bases: *pandagg.node.aggs.abstract.UniqueBucketAgg*

KEY = 'missing'

VALUE_ATTRS = ['doc_count']

class pandagg.aggs.**Nested**(*path: str*, ***body*)
Bases: *pandagg.node.aggs.abstract.UniqueBucketAgg*

KEY = 'nested'

VALUE_ATTRS = ['doc_count']

WHITELISTED_MAPPING_TYPES = ['nested']

class pandagg.aggs.**ReverseNested**(*path: Optional[str] = None*, ***body*)
Bases: *pandagg.node.aggs.abstract.UniqueBucketAgg*

KEY = 'reverse_nested'

VALUE_ATTRS = ['doc_count']

WHITELISTED_MAPPING_TYPES = ['nested']

class pandagg.aggs.**Avg**(*field: Optional[str] = None*, *script: Optional[pandagg.types.Script] = 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: Optional[str] = None*, *script: Optional[pandagg.types.Script] = 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: Optional[str] = None*, *script: Optional[pandagg.types.Script] = 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: Optional[str] = None*, *script: Optional[pandagg.types.Script] = 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: Optional[str] = None*, *script: Optional[pandagg.types.Script] = None*, ***body*)

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

KEY = 'cardinality'

VALUE_ATTRS = ['value']

class pandagg.aggs.**Stats**(*field: Optional[str] = None*, *script: Optional[pandagg.types.Script] = 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: Optional[str] = None*, *script: Optional[pandagg.types.Script] = 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: Optional[str] = None, script: Optional[pandagg.types.Script] = 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: str, values: List[float], **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.aggs.**GeoBound**(*field: Optional[str] = None, script: Optional[pandagg.types.Script] = None, **body*)

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

KEY = 'geo_bounds'

VALUE_ATTRS = ['bounds']

WHITELISTED_MAPPING_TYPES = ['geo_point']

class pandagg.aggs.**GeoCentroid**(*field: Optional[str] = None, script: Optional[pandagg.types.Script] = None, **body*)

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

KEY = 'geo_centroid'

VALUE_ATTRS = ['location']

WHITELISTED_MAPPING_TYPES = ['geo_point']

class pandagg.aggs.**TopHits**(*meta: Optional[Dict[str, Any]] = None, identifier: Optional[str] = None, **body*)

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

KEY = 'top_hits'

VALUE_ATTRS = ['hits']

class pandagg.aggs.**ValueCount**(*field: Optional[str] = None, script: Optional[pandagg.types.Script] = None, **body*)

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

KEY = 'value_count'

VALUE_ATTRS = ['value']

class pandagg.aggs.**AvgBucket**(*buckets_path: str, gap_policy: Optional[typing_extensions.Literal['skip', 'insert_zeros', 'keep_values']][skip, insert_zeros, keep_values]] = None, **body*)

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

KEY = 'avg_bucket'

VALUE_ATTRS = ['value']

class pandagg.aggs.**Derivative**(*buckets_path: str, gap_policy: Optional[typing_extensions.Literal['skip', 'insert_zeros', 'keep_values']][skip, insert_zeros, keep_values]] = None, **body*)

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

KEY = 'derivative'

VALUE_ATTRS = ['value']

class pandagg.aggs.**MaxBucket**(*buckets_path: str, gap_policy: Optional[typing_extensions.Literal['skip', 'insert_zeros', 'keep_values']][skip, insert_zeros, keep_values]] = None, **body*)

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

KEY = 'max_bucket'

VALUE_ATTRS = ['value']

class pandagg.aggs.**MinBucket**(*buckets_path: str, gap_policy: Optional[typing_extensions.Literal['skip', 'insert_zeros', 'keep_values']][skip, insert_zeros, keep_values]] = None, **body*)

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

KEY = 'min_bucket'

VALUE_ATTRS = ['value']

class pandagg.aggs.**SumBucket**(*buckets_path: str, gap_policy: Optional[typing_extensions.Literal['skip', 'insert_zeros', 'keep_values']][skip, insert_zeros, keep_values]] = None, **body*)

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

KEY = 'sum_bucket'

VALUE_ATTRS = ['value']

class pandagg.aggs.**StatsBucket**(*buckets_path: str, gap_policy: Optional[typing_extensions.Literal['skip', 'insert_zeros', 'keep_values']][skip, insert_zeros, keep_values]] = None, **body*)

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

KEY = 'stats_bucket'

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

class pandagg.aggs.**ExtendedStatsBucket**(*buckets_path: str, gap_policy: Optional[typing_extensions.Literal['skip', 'insert_zeros', 'keep_values']][skip, insert_zeros, keep_values]] = 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: str, gap_policy: Optional[typing_extensions.Literal['skip', 'insert_zeros', 'keep_values']][skip, insert_zeros, keep_values]] = None, **body*)

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

KEY = 'percentiles_bucket'

VALUE_ATTRS = ['values']

class pandagg.aggs.**MovingAvg**(*buckets_path: str*, *gap_policy: Optional[typing_extensions.Literal['skip', 'insert_zeros', 'keep_values']][skip, insert_zeros, keep_values]] = None*, ***body*)

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

KEY = 'moving_avg'

VALUE_ATTRS = ['value']

class pandagg.aggs.**CumulativeSum**(*buckets_path: str*, *gap_policy: Optional[typing_extensions.Literal['skip', 'insert_zeros', 'keep_values']][skip, insert_zeros, keep_values]] = None*, ***body*)

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

KEY = 'cumulative_sum'

VALUE_ATTRS = ['value']

class pandagg.aggs.**BucketScript**(*script: pandagg.types.Script*, *buckets_path: str*, *gap_policy: Optional[typing_extensions.Literal['skip', 'insert_zeros', 'keep_values']][skip, insert_zeros, keep_values]] = None*, ***body*)

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

KEY = 'bucket_script'

VALUE_ATTRS = ['value']

class pandagg.aggs.**BucketSelector**(*script: pandagg.types.Script*, *buckets_path: str*, *gap_policy: Optional[typing_extensions.Literal['skip', 'insert_zeros', 'keep_values']][skip, insert_zeros, keep_values]] = None*, ***body*)

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

KEY = 'bucket_selector'

VALUE_ATTRS = []

class pandagg.aggs.**BucketSort**(*script: pandagg.types.Script*, *buckets_path: str*, *gap_policy: Optional[typing_extensions.Literal['skip', 'insert_zeros', 'keep_values']][skip, insert_zeros, keep_values]] = None*, ***body*)

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

KEY = 'bucket_sort'

VALUE_ATTRS = []

class pandagg.aggs.**SerialDiff**(*buckets_path: str*, *gap_policy: Optional[typing_extensions.Literal['skip', 'insert_zeros', 'keep_values']][skip, insert_zeros, keep_values]] = None*, ***body*)

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

KEY = 'serial_diff'

VALUE_ATTRS = ['value']

class pandagg.aggs.**MatchAll**(***body*)

Bases: *pandagg.node.aggs.bucket.Filter*

class pandagg.aggs.**Composite**(*sources: List[Dict[str, Dict[str, Dict[str, Any]]]], size: Optional[int] = None, after: Optional[Dict[str, Any]] = None, **body*)

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

KEY = 'composite'

VALUE_ATTRS = ['doc_count']

after

extract_buckets(*response_value: Union[pandagg.types.BucketsWrapperDict, Dict[str, Any]]*) → Iterator[Tuple[Dict[str, Union[str, float, None]], Dict[str, Any]]]

size

source_names

sources

class pandagg.aggs.**GeoHashGrid**(*field: str, precision: Optional[int] = None, bounds: Optional[Dict[KT, VT]] = None, size: Optional[int] = None, shard_size: Optional[int] = None, **body*)

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

KEY = 'geohash_grid'

VALUE_ATTRS = ['doc_count']

WHITELISTED_MAPPING_TYPES = ['geo_point', 'geo_shape']

class pandagg.aggs.**GeoDistance**(*field: str, origin: str, ranges: List[pandagg.types.RangeDict], unit: Optional[str] = None, distance_type: Optional[typing_extensions.Literal['arc', 'plane']][arc, plane]] = None, keyed: bool = False, **body*)

Bases: *pandagg.node.aggs.bucket.Range*

KEY = 'geo_distance'

VALUE_ATTRS = ['doc_count']

WHITELISTED_MAPPING_TYPES = ['geo_point']

class pandagg.aggs.**AdjacencyMatrix**(*filters: Dict[str, Dict[str, Dict[str, Any]]], separator: Optional[str] = None, **body*)

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

KEY = 'adjacency_matrix'

VALUE_ATTRS = ['doc_count']

class pandagg.aggs.**AutoDateHistogram**(*field: str, buckets: Optional[int] = None, format: Optional[str] = None, time_zone: Optional[str] = None, minimum_interval: Optional[str] = None, missing: Optional[str] = None, key_as_string: bool = True, **body*)

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

KEY = 'auto_date_histogram'

VALUE_ATTRS = ['doc_count']

class pandagg.aggs.**VariableWidthHistogram**(*field: str, buckets: int, **body*)

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

KEY = 'variable_width_histogram'

VALUE_ATTRS = ['doc_count', 'min', 'max']

class pandagg.aggs.**SignificantTerms**(*field: str*, ***body*)
Bases: *pandagg.node.aggs.abstract.MultipleBucketAgg*

KEY = 'significant_terms'

VALUE_ATTRS = ['doc_count', 'score', 'bg_count']

class pandagg.aggs.**RareTerms**(*field: str, max_doc_count: Optional[int] = None, precision: Optional[float] = None, include: Union[str, List[str], None] = None, exclude: Union[str, List[str], None] = None, missing: Optional[Any] = None, **body*)
Bases: *pandagg.node.aggs.abstract.MultipleBucketAgg*

KEY = 'rare_terms'

VALUE_ATTRS = ['doc_count']

class pandagg.aggs.**GeoTileGrid**(*field: str, precision: Optional[int] = None, bounds: Optional[Dict[KT, VT]] = None, size: Optional[int] = None, shard_size: Optional[int] = None, **body*)
Bases: *pandagg.node.aggs.abstract.MultipleBucketAgg*

KEY = 'geotile_grid'

VALUE_ATTRS = ['doc_count']

WHITELISTED_MAPPING_TYPES = ['geo_point', 'geo_shape']

class pandagg.aggs.**IPRange**(*field: str, ranges: List[pandagg.types.RangeDict], keyed: bool = False, **body*)
Bases: *pandagg.node.aggs.bucket.Range*

KEY = 'ip_range'

VALUE_ATTRS = ['doc_count']

WHITELISTED_MAPPING_TYPES = ['ip']

class pandagg.aggs.**Sampler**(*shard_size: Optional[int] = None*, ***body*)
Bases: *pandagg.node.aggs.abstract.UniqueBucketAgg*

KEY = 'sampler'

VALUE_ATTRS = ['doc_count']

class pandagg.aggs.**DiversifiedSampler**(*field: str, shard_size: Optional[int], max_docs_per_value: Optional[int] = None, execution_hint: Optional[typing_extensions.Literal['map', 'global_ordinals', 'bytes_hash']][map, global_ordinals, bytes_hash]] = None, **body*)
Bases: *pandagg.node.aggs.abstract.UniqueBucketAgg*

KEY = 'diversified_sampler'

VALUE_ATTRS = ['doc_count']

class pandagg.aggs.**Children**(*type: str*, ***body*)
Bases: *pandagg.node.aggs.abstract.UniqueBucketAgg*

KEY = 'children'

VALUE_ATTRS = ['doc_count']

class pandagg.aggs.**Parent**(*type: str*, ***body*)
Bases: *pandagg.node.aggs.abstract.UniqueBucketAgg*

KEY = 'parent'

VALUE_ATTRS = ['doc_count']

class pandagg.aggs.**SignificantText**(*field: str*, ***body*)
Bases: *pandagg.node.aggs.abstract.MultipleBucketAgg*

KEY = 'significant_text'

VALUE_ATTRS = ['doc_count', 'score', 'bg_count']

WHITELISTED_MAPPING_TYPES = ['text']

class pandagg.aggs.**MultiTerms**(*terms: List[Dict[KT, VT]]*, ***body*)
Bases: *pandagg.node.aggs.abstract.MultipleBucketAgg*

KEY = 'multi_terms'

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

4.2.2 pandagg.discovery module

class pandagg.discovery.**Index**(*name: str*, *settings: Dict[str, Any]*, *mappings: pandagg.types.MappingsDict*, *aliases: Any*, *client: Union[elasticsearch.client.Elasticsearch, NoneType] = None*)
Bases: object

client = None

imappings

search(*nested_autocorrect: bool = True*, *repr_auto_execute: bool = True*) → pandagg.search.Search

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

pandagg.discovery.**discover**(*using: elasticsearch.client.Elasticsearch*, *index: str = '*'*) → pandagg.discovery.Indices

Parameters

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

4.2.3 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.

4.2.4 pandagg.mappings module

class pandagg.mappings.**Mappings**(*properties: Optional[Dict[str, Union[Dict[str, Any], pandagg.node.mappings.abstract.Field]]] = None, dynamic: Optional[bool] = None, **body*)
Bases: pandagg.tree._tree.TreeReprMixin, lighttree.tree.Tree

list_nesteds_at_field(*field_path: str*) → List[str]
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: str*) → str
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: str*) → Optional[str]
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'},
```

(continues on next page)

(continued from previous page)

```
>>>         'date': {'type': 'date'}
>>>     }}
>>> })
>>> mappings.nested_at_field('id')
None
>>> mappings.nested_at_field('comments')
'comments'
>>> mappings.nested_at_field('comments.comment_text')
'comments'
```

to_dict(*from_: Optional[str] = None*, *depth: Optional[int] = None*) → pandagg.types.MappingsDict
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: pandagg.node.aggs.abstract.AggClause*, *exc: bool = True*) → bool
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: Union[DocSource, DocumentSource]*) → None

class pandagg.mappings.**IMappings**(*mappings: pandagg.tree.mappings.Mappings*, *client: Optional[elasticsearch.client.Elasticsearch] = None*, *index: Optional[List[str]] = None*, *depth: int = 1*, *root_path: Optional[str] = None*, *initial_tree: Optional[pandagg.tree.mappings.Mappings] = 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.**IpRange**(***body*)
Bases: *pandagg.node.mappings.abstract.RegularField*

KEY = 'ip_range'

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**(*properties: Optional[Union[Dict, Type[DocumentSource]]] = None*, **body)
Bases: *pandagg.node.mappings.abstract.ComplexField*

KEY = 'object'

class pandagg.mappings.**Nested**(*properties: Optional[Union[Dict, Type[DocumentSource]]] = None*, **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: Optional[bool] = None*, *required: bool = False*, **body)
Bases: *pandagg.node.mappings.abstract.Field*

The index to which the document belongs.

KEY = '_index'

class pandagg.mappings.**Type**(*, *multiple: Optional[bool] = None*, *required: bool = False*, **body)
Bases: *pandagg.node.mappings.abstract.Field*

The document's mappings type.

KEY = '_type'

class pandagg.mappings.**Id**(*, *multiple: Optional[bool] = None*, *required: bool = False*, **body)
Bases: *pandagg.node.mappings.abstract.Field*

The document's ID.

KEY = '_id'

class pandagg.mappings.**FieldNames**(*, *multiple: Optional[bool] = None*, *required: bool = False*, **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: Optional[bool] = None*, *required: bool = False*, **body)
Bases: *pandagg.node.mappings.abstract.Field*

The original JSON representing the body of the document.

KEY = '_source'

class pandagg.mappings.**Size**(*, *multiple: Optional[bool] = None*, *required: bool = False*, **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: Optional[bool] = None*, *required: bool = False*, ***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: Optional[bool] = None*, *required: bool = False*, ***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: Optional[bool] = None*, *required: bool = False*, ***body*)

Bases: *pandagg.node.mappings.abstract.Field*

Application specific metadata.

KEY = '_meta'

4.2.5 pandagg.query module

class pandagg.query.**Query**(*q: Union[str, Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause, Query, None] = None*, *mappings: Union[pandagg.types.MappingsDict, pandagg.tree.mappings.Mappings, None] = None*, *nested_autocorrect: bool = False*)

Bases: lighttree.tree.Tree

applied_nested_path_at_node(*nid: str*) → Optional[str]

Return nested path applied at a clause.

Parameters **nid** – clause identifier

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

bool(*must: Union[Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause, List[Union[Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause]], None] = None*, *should: Union[Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause, List[Union[Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause]], None] = None*, *must_not: Union[Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause, List[Union[Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause]], None] = None*, *filter: Union[Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause, List[Union[Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause]], None] = None*, *insert_below: Optional[str] = None*, *on: Optional[str] = None*, *mode: typing_extensions.Literal['add', 'replace', 'replace_all'][add, replace, replace_all] = 'add'*, ***body*) → pandagg.tree.query.Query

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

boosting(*positive: Union[Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause, None] = None*, *negative: Union[Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause, None] = None*, *insert_below: Optional[str] = None*, *on: Optional[str] = None*, *mode: typing_extensions.Literal['add', 'replace', 'replace_all'][add, replace, replace_all] = 'add'*, ***body*) → pandagg.tree.query.Query

constant_score(*filter: Union[Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause, None] = None, boost: Optional[float] = None, insert_below: Optional[str] = None, on: Optional[str] = None, mode: typing_extensions.Literal['add', 'replace', 'replace_all']*[add, replace, replace_all] = 'add', **body) → pandagg.tree.query.Query

dis_max(*queries: List[Union[Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause]], insert_below: Optional[str] = None, on: Optional[str] = None, mode: typing_extensions.Literal['add', 'replace', 'replace_all']*[add, replace, replace_all] = 'add', **body) → pandagg.tree.query.Query

filter(*type_or_query: Union[str, Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause, Query], insert_below: Optional[str] = None, on: Optional[str] = None, mode: typing_extensions.Literal['add', 'replace', 'replace_all']*[add, replace, replace_all] = 'add', *bool_body: Dict[str, Any] = None*, **body) → pandagg.tree.query.Query

function_score(*query: Union[Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause, None], insert_below: Optional[str] = None, on: Optional[str] = None, mode: typing_extensions.Literal['add', 'replace', 'replace_all']*[add, replace, replace_all] = 'add', **body) → pandagg.tree.query.Query

has_child(*query: Union[Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause, None], insert_below: Optional[str] = None, on: Optional[str] = None, mode: typing_extensions.Literal['add', 'replace', 'replace_all']*[add, replace, replace_all] = 'add', **body) → pandagg.tree.query.Query

has_parent(*query: Union[Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause, None], insert_below: Optional[str] = None, on: Optional[str] = None, mode: typing_extensions.Literal['add', 'replace', 'replace_all']*[add, replace, replace_all] = 'add', **body) → pandagg.tree.query.Query

must(*type_or_query: Union[str, Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause, Query], insert_below: Optional[str] = None, on: Optional[str] = None, mode: typing_extensions.Literal['add', 'replace', 'replace_all']*[add, replace, replace_all] = 'add', *bool_body: Optional[Dict[str, Any]] = None*, **body) → pandagg.tree.query.Query

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: Union[str, Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause, Query], insert_below: Optional[str] = None, on: Optional[str] = None, mode: typing_extensions.Literal['add', 'replace', 'replace_all']['add, replace, replace_all] = 'add', bool_body: Dict[str, Any] = None, **body*) → pandagg.tree.query.Query

nested (*path: str, query: Union[Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause, None] = None, insert_below: Optional[str] = None, on: Optional[str] = None, mode: typing_extensions.Literal['add', 'replace', 'replace_all']['add, replace, replace_all] = 'add', **body*) → pandagg.tree.query.Query

pinned_query (*organic: Union[Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause, None], insert_below: Optional[str] = None, on: Optional[str] = None, mode: typing_extensions.Literal['add', 'replace', 'replace_all']['add, replace, replace_all] = 'add', **body*) → pandagg.tree.query.Query

query (*type_or_query: Union[str, Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause, Query], insert_below: Optional[str] = None, on: Optional[str] = None, mode: typing_extensions.Literal['add', 'replace', 'replace_all']['add, replace, replace_all] = 'add', compound_param: str = None, **body*) → pandagg.tree.query.Query

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: Union[Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause, None], insert_below: Optional[str] = None, on: Optional[str] = None, mode: typing_extensions.Literal['add', 'replace', 'replace_all']['add, replace, replace_all] = 'add', **body*) → pandagg.tree.query.Query

should(*type_or_query: Union[str, Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause, Query], insert_below: Optional[str] = None, on: Optional[str] = None, mode: typing_extensions.Literal['add', 'replace', 'replace_all'][add, replace, replace_all] = 'add', bool_body: Optional[Dict[str, Any]] = None, **body*) → pandagg.tree.query.Query

show(**args*, *line_max_length: int = 80*, ***kwargs*) → str

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.

to_dict(*from_: Optional[str] = None*) → Optional[Dict[str, Dict[str, Any]]]

class pandagg.query.**Exists**(*field: str*, *_name: Optional[str] = None*)

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

KEY = 'exists'

line_repr(*depth: int*, ***kwargs*) → Tuple[str, str]

Control how node is displayed in tree representation. First returned string is how node is represented on left, second string is how node is represented on right.

MyTree ├── one OneEnd │ └── two twoEnd └── three threeEnd

class pandagg.query.**Fuzzy**(*field: Optional[str] = None*, *_name: Optional[str] = None*, *_expand__to_dot: bool = True*, ***params*)

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

KEY = 'fuzzy'

class pandagg.query.**Ids**(*values: List[Union[str, int]]*, *_name: Optional[str] = None*)

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

KEY = 'ids'

line_repr(*depth: int*, ***kwargs*) → Tuple[str, str]

Control how node is displayed in tree representation. First returned string is how node is represented on left, second string is how node is represented on right.

MyTree ├── one OneEnd │ └── two twoEnd └── three threeEnd

class pandagg.query.**Prefix**(*field: Optional[str] = None*, *_name: Optional[str] = None*, *_expand__to_dot: bool = True*, ***params*)

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

KEY = 'prefix'

class pandagg.query.**Range**(*field: Optional[str] = None*, *_name: Optional[str] = None*, *_expand__to_dot: bool = True*, ***params*)

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

KEY = 'range'

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

KEY = 'regexp'

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

KEY = 'terms_set'

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

KEY = 'type'

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

KEY = 'wildcard'

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

KEY = 'intervals'

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

KEY = 'match'

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

KEY = 'match_bool_prefix'

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

KEY = 'match_phrase'

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

KEY = 'match_phrase_prefix'

class pandagg.query.**MultiMatch**(*fields: List[str]*, *_name: Optional[str] = None*, ***body*)
Bases: *pandagg.node.query.abstract.MultiFieldsQueryClause*

KEY = 'multi_match'

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

KEY = 'common'

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

KEY = 'query_string'

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

KEY = 'simple_string'

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

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

KEY = 'bool'

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

KEY = 'boosting'

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

KEY = 'constant_score'

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

KEY = 'function_score'

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

KEY = 'dis_max'

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

KEY = 'nested'

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

KEY = 'has_parent'

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

KEY = 'has_child'

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

KEY = 'parent_id'

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

KEY = 'shape'

class pandagg.query.**GeoShape**(*field: Optional[str] = None*, *_name: Optional[str] = None*, *_expand__to_dot: bool = True*, ***params*)
Bases: *pandagg.node.query.abstract.KeyFieldQueryClause*

KEY = 'geo_shape'

class pandagg.query.**GeoPolygone**(*field: Optional[str] = None*, *_name: Optional[str] = None*, *_expand__to_dot: bool = True*, ***params*)
Bases: *pandagg.node.query.abstract.KeyFieldQueryClause*

KEY = 'geo_polygon'

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

KEY = 'geo_distance'

line_repr(*depth: int*, ***kwargs*) → Tuple[str, str]
Control how node is displayed in tree representation. First returned string is how node is represented on left, second string is how node is represented on right.

MyTree ├── one OneEnd │ └── two twoEnd └── three threeEnd

class pandagg.query.**GeoBoundingBox**(*field: Optional[str] = None*, *_name: Optional[str] = None*, *_expand__to_dot: bool = True*, ***params*)
Bases: *pandagg.node.query.abstract.KeyFieldQueryClause*

KEY = 'geo_bounding_box'

class pandagg.query.**DistanceFeature**(*field: str*, *_name: Optional[str] = None*, ***body*)
Bases: *pandagg.node.query.abstract.FlatFieldQueryClause*

KEY = 'distance_feature'

class pandagg.query.**MoreLikeThis**(*fields: List[str]*, *_name: Optional[str] = None*, ***body*)
Bases: *pandagg.node.query.abstract.MultiFieldsQueryClause*

KEY = 'more_like_this'

class pandagg.query.**Percolate**(*field: str*, *_name: Optional[str] = None*, ***body*)
Bases: *pandagg.node.query.abstract.FlatFieldQueryClause*

KEY = 'percolate'

class pandagg.query.**RankFeature**(*field: str*, *_name: Optional[str] = None*, ***body*)
Bases: *pandagg.node.query.abstract.FlatFieldQueryClause*

KEY = 'rank_feature'

class pandagg.query.**Script**(*_name: Optional[str] = None*, ***body*)
Bases: *pandagg.node.query.abstract.LeafQueryClause*

KEY = 'script'

class pandagg.query.**Wrapper**(*_name: Optional[str] = None*, ***body*)
Bases: *pandagg.node.query.abstract.LeafQueryClause*

KEY = 'wrapper'

class pandagg.query.**ScriptScore**(*_name: Optional[str] = None*, ***body*)
Bases: *pandagg.node.query.compound.CompoundClause*

KEY = 'script_score'

class pandagg.query.**PinnedQuery**(*_name: Optional[str] = None*, ***body*)
Bases: *pandagg.node.query.compound.CompoundClause*

KEY = 'pinned'

4.2.6 pandagg.response module

class pandagg.response.**Aggregations**(*data: 'AggregationsResponseDict'*, *_search: 'Search'*)
Bases: object

keys() → List[str]

parse_group_by(**, response: Dict[str, Union[pandagg.types.BucketsWrapperDict, Dict[str, Any]]], until: Optional[str], with_single_bucket_groups: bool = False, row_as_tuple: bool = False*) → Tuple[List[str], Union[List[Tuple[Tuple[Union[None, str, float], ...], Dict[str, Any]]], List[Tuple[Dict[str, Union[str, float, None]], Dict[str, Any]]]]]

to_dataframe(*grouped_by: Optional[str] = None*, *normalize_children: bool = True*, *with_single_bucket_groups: bool = False*) → pd.DataFrame

to_normalized() → pandagg.response.NormalizedBucketDict

to_tabular(**, index_orient: bool = True*, *grouped_by: Optional[str] = None*, *expand_columns: bool = True*, *expand_sep: str = '|'*, *normalize: bool = True*, *with_single_bucket_groups: bool = False*) → Tuple[List[str], Union[Dict[Tuple[Union[None, str, float], ...], Dict[str, Any]], List[Dict[str, Any]]]]

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 dictionary
- **grouped_by** – name of the aggregation node used as last grouping level
- **normalize** – if True, normalize columns buckets

Returns index_names, values

class pandagg.response.**Hit**(*data: 'HitDict'*, *_document_class: 'Optional[DocumentMeta]'*)
Bases: object

class pandagg.response.**Hits**(*data: 'Optional[HitsDict]'*, *_document_class: 'Optional[DocumentMeta]'*)

Bases: object

hits

max_score

to_dataframe(*expand_source: bool = True*, *source_only: bool = True*) → pd.DataFrame

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)

total

class pandagg.response.**NormalizedBucketDict**

Bases: dict

class pandagg.response.**SearchResponse**(*data: 'SearchResponseDict'*, *_search: 'Search'*)

Bases: object

aggregations

hits

profile

success

timed_out

took

4.2.7 pandagg.search module

class pandagg.search.**MultiSearch**(*using: Optional[elasticsearch.client.Elasticsearch]*, *index: Union[str, Tuple[str], List[str], None] = None*)

Bases: *pandagg.search.Request*

Combine multiple Search objects into a single request.

add(*search: pandagg.search.Search*) → MultiSearch

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() → List[pandagg.types.SearchResponseDict]

Execute the multi search request and return a list of search results.

to_dict() → List[Union[Dict[KT, VT], pandagg.types.SearchDict]]

class pandagg.search.**Request**(*using: Optional[elasticsearch.client.Elasticsearch]*, *index: Union[str, Tuple[str], List[str], None] = None*)

Bases: object

index(**index*) → T

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*) → T

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: elasticsearch.client.Elasticsearch*) → T

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: Optional[Elasticsearch] = None, index: Optional[Union[str, Tuple[str], List[str]]] = None, mappings: Optional[Union[MappingsDict, Mappings]] = None, nested_autocorrect: bool = False, repr_auto_execute: bool = False, document_class: DocumentMeta = None*)

Bases: *pandagg.utils.DSLMixin*, *pandagg.search.Request*

agg(*name: str, type_or_agg: Union[str, Dict[str, Dict[str, Any]], pandagg.node.aggs.abstract.AggClause, None] = None, insert_below: Optional[str] = None, at_root: bool = False, **body*) → Search

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: Union[Dict[str, Union[Dict[str, Dict[str, Any]], pandagg.node.aggs.abstract.AggClause]], Aggs], insert_below: Optional[str] = None, at_root: bool = False*) → Search

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: Union[Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause, List[Union[Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause]], None] = None, should: Union[Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause, List[Union[Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause]], None] = None, must_not: Union[Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause, List[Union[Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause]], None] = None, filter: Union[Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause, List[Union[Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause]], None] = None, insert_below: Optional[str] = None, on: Optional[str] = None, mode: typing_extensions.Literal['add', 'replace', 'replace_all'][add, replace, replace_all] = 'add', **body*) → Search

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

count() → int
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: Union[str, Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause, Query], insert_below: Optional[str] = None, on: Optional[str] = None, mode: typing_extensions.Literal['add', 'replace', 'replace_all'][add, replace, replace_all] = 'add', bool_body: Optional[Dict[str, Any]] = None, **body*) → Search
Must not wrapped in filter context.

execute() → pandagg.response.SearchResponse
Execute the search and return an instance of `Response` wrapping all the data.

filter(*type_or_query: Union[str, Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause, Query], insert_below: Optional[str] = None, on: Optional[str] = None, mode: typing_extensions.Literal['add', 'replace', 'replace_all'][add, replace, replace_all] = 'add', bool_body: Optional[Dict[str, Any]] = None, **body*) → Search

classmethod from_dict(*d: Dict[KT, VT]*) → Search
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: str, type_or_agg: Union[str, Dict[str, Dict[str, Any]], pandagg.node.aggs.abstract.AggClause, None] = None, insert_below: Optional[str] = None, at_root: bool = False, **body*) → Search

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*) → Search

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": {
            "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}
```

(continues on next page)

(continued from previous page)

```
            }
        }
}
```

highlight_options(***kwargs*) → Search

Update the global highlighting options used for this request. For example:

```
s = Search()
s = s.highlight_options(order='score')
```

must(*type_or_query: Union[str, Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause, Query], insert_below: Optional[str] = None, on: Optional[str] = None, mode: typing_extensions.Literal['add', 'replace', 'replace_all']][add, replace, replace_all] = 'add', bool_body: Optional[Dict[str, Any]] = None, **body*) → Search

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: Union[str, Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause, Query], insert_below: Optional[str] = None, on: Optional[str] = None, mode: typing_extensions.Literal['add', 'replace', 'replace_all']][add, replace, replace_all] = 'add', bool_body: Optional[Dict[str, Any]] = None, **body*) → Search

post_filter(*type_or_query: Union[str, Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause, Query], insert_below: Optional[str] = None, on: Optional[str] = None, mode: typing_extensions.Literal['add', 'replace', 'replace_all']][add, replace, replace_all] = 'add', compound_param: Optional[str] = None, **body*) → Search

query(*type_or_query: Union[str, Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause, Query], insert_below: Optional[str] = None, on: Optional[str] = None, mode: typing_extensions.Literal['add', 'replace', 'replace_all']][add, replace, replace_all] = 'add', compound_param: Optional[str] = None, **body*) → Search

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() → Iterator[pandagg.response.Hit]

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

scan_composite_agg(*size: int*) → Iterator[Dict[str, Any]]

Iterate over the whole aggregation composed buckets, yields buckets.

scan_composite_agg_at_once(*size: int*) → pandagg.response.Aggregations

Iterate over the whole aggregation composed buckets (converting Aggs into composite agg if possible), and return all buckets at once in a Aggregations instance.

script_fields(***kwargs*) → Search

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}
        }
```

(continues on next page)

(continued from previous page)

```
    }
)
```

should(*type_or_query: Union[str, Dict[str, Dict[str, Any]], pandagg.node.query.abstract.QueryClause, Query], insert_below: Optional[str] = None, on: Optional[str] = None, mode: typing_extensions.Literal['add', 'replace', 'replace_all'][add, replace, replace_all] = 'add', bool_body: Optional[Dict[str, Any]] = None, **body*) → Search

size(*size: int*) → Search

Equivalent to:

```
s = Search().params(size=size)
```

sort(**keys*) → Search

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: Union[str, List[str], Dict[str, Any], None] = None, **kwargs*) → Search

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: str*, *text: str*, ***kwargs*) → Search

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: bool = False*, ***kwargs*) → pandagg.types.SearchDict
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: Dict[KT, VT]*) → Search
Apply options from a serialized body to the current instance. Modifies the object in-place. Used mostly by `from_dict`.

4.2.8 pandagg.types module

class pandagg.types.**Action**
Bases: dict

class pandagg.types.**AliasValue**
Bases: dict

class pandagg.types.**BucketsWrapperDict**
Bases: dict

class pandagg.types.**DeleteByQueryResponse**
Bases: dict

class pandagg.types.**FieldDict**
Bases: dict

class pandagg.types.**HitDict**
Bases: dict

class pandagg.types.**HitsDict**
Bases: dict

class pandagg.types.**MappingsDict**
Bases: dict

class pandagg.types.**PointInTimeDict**
Bases: dict

class pandagg.types.**ProfileDict**
Bases: dict

class pandagg.types.**ProfileShardDict**
Bases: dict

class pandagg.types.**RangeDict**
Bases: dict

class pandagg.types.**RetriesDict**
Bases: dict

class pandagg.types.**RunTimeMappingDict**
Bases: dict

class pandagg.types.**Script**
Bases: dict

class pandagg.types.**SearchDict**
Bases: dict

class pandagg.types.**SearchResponseDict**
Bases: dict

class pandagg.types.**ShardsDict**
Bases: dict

class pandagg.types.**SourceIncludeDict**
Bases: dict

class pandagg.types.**SuggestedItemDict**
Bases: dict

class pandagg.types.**TotalDict**
Bases: 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.

classmethod get_dsl_class(*name: str*) → pandagg.utils.DslMeta

static get_dsl_type(*name: str*) → pandagg.utils.DslMeta

class pandagg.utils.**DslMeta**(*name: str, bases: Tuple, attrs: Dict[KT, VT]*)
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).

Types will be: 'agg', 'query', 'field'

Each of those types will hold a *_classes* dictionary pointing to all classes of same type.

KEY = ''

pandagg.utils.**equal_queries**(*d1: Any*, *d2: Any*) → bool
Compares if two queries are equivalent (do not consider nested list orders).

pandagg.utils.**equal_search**(*s1: Any*, *s2: Any*) → bool

pandagg.utils.**get_action_modifier**(*index_name: str*, *_op_type_overwrite: Optional[typing_extensions.Literal['create', 'index', 'update', 'delete']][create, index, update, delete]] = None*) → Callable

pandagg.utils.**is_subset**(*subset: Any*, *superset: Any*) → bool

pandagg.utils.**ordered**(*obj: Any*) → Any

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

pandagg, 95
pandagg.aggs, 61
pandagg.discovery, 72
pandagg.exceptions, 72
pandagg.interactive, 29
pandagg.interactive.mappings, 29
pandagg.mappings, 73
pandagg.node, 54
pandagg.node.aggs, 41
pandagg.node.aggs.abstract, 30
pandagg.node.aggs.bucket, 32
pandagg.node.aggs.composite, 36
pandagg.node.aggs.metric, 37
pandagg.node.aggs.pipeline, 39
pandagg.node.mappings, 47
pandagg.node.mappings.abstract, 41
pandagg.node.mappings.field_datatypes, 42
pandagg.node.mappings.meta_fields, 46
pandagg.node.query, 53
pandagg.node.query.abstract, 47
pandagg.node.query.compound, 49
pandagg.node.query.full_text, 49
pandagg.node.query.geo, 50
pandagg.node.query.joining, 51
pandagg.node.query.shape, 51
pandagg.node.query.span, 51
pandagg.node.query.specialized, 51
pandagg.node.query.specialized_compound, 52
pandagg.node.query.term_level, 52
pandagg.node.types, 54
pandagg.query, 79
pandagg.response, 86
pandagg.search, 87
pandagg.tree, 61
pandagg.tree.aggs, 54
pandagg.tree.mappings, 57
pandagg.tree.query, 58
pandagg.types, 94
pandagg.utils, 95

Index

A

A() (*in module pandagg.node.aggs.abstract*), 30
AbsentMappingFieldError, 72
AbstractSingleFieldQueryClause (*class in pandagg.node.query.abstract*), 47
Action (*class in pandagg.types*), 94
add() (*pandagg.search.MultiSearch method*), 87
AdjacencyMatrix (*class in pandagg.aggs*), 70
AdjacencyMatrix (*class in pandagg.node.aggs.bucket*), 32
after (*pandagg.aggs.Composite attribute*), 70
after (*pandagg.node.aggs.composite.Composite attribute*), 36
agg() (*pandagg.aggs.Aggs method*), 62
agg() (*pandagg.search.Search method*), 88
agg() (*pandagg.tree.aggs.Aggs method*), 54
AggClause (*class in pandagg.node.aggs.abstract*), 30
Aggregations (*class in pandagg.response*), 86
aggregations (*pandagg.response.SearchResponse attribute*), 87
Aggs (*class in pandagg.aggs*), 61
Aggs (*class in pandagg.tree.aggs*), 54
aggs() (*pandagg.aggs.Aggs method*), 62
aggs() (*pandagg.search.Search method*), 88
aggs() (*pandagg.tree.aggs.Aggs method*), 55
Alias (*class in pandagg.mappings*), 77
Alias (*class in pandagg.node.mappings.field_datatypes*), 42
AliasValue (*class in pandagg.types*), 94
applied_nested_path_at_node() (*pandagg.aggs.Aggs method*), 63
applied_nested_path_at_node() (*pandagg.query.Query method*), 79
applied_nested_path_at_node() (*pandagg.tree.aggs.Aggs method*), 55
applied_nested_path_at_node() (*pandagg.tree.query.Query method*), 58
apply_reverse_nested() (*pandagg.aggs.Aggs method*), 63
apply_reverse_nested() (*pandagg.tree.aggs.Aggs method*), 55
as_composite() (*pandagg.aggs.Aggs method*), 63
as_composite() (*pandagg.tree.aggs.Aggs method*), 55
AutoDateHistogram (*class in pandagg.aggs*), 70
AutoDateHistogram (*class in pandagg.node.aggs.bucket*), 32
Avg (*class in pandagg.aggs*), 66
Avg (*class in pandagg.node.aggs.metric*), 37
AvgBucket (*class in pandagg.aggs*), 67
AvgBucket (*class in pandagg.node.aggs.pipeline*), 39

B

Binary (*class in pandagg.mappings*), 75
Binary (*class in pandagg.node.mappings.field_datatypes*), 42
Bool (*class in pandagg.node.query.compound*), 49
Bool (*class in pandagg.query*), 84
bool() (*pandagg.query.Query method*), 79
bool() (*pandagg.search.Search method*), 89
bool() (*pandagg.tree.query.Query method*), 58
Boolean (*class in pandagg.mappings*), 75
Boolean (*class in pandagg.node.mappings.field_datatypes*), 42
Boosting (*class in pandagg.node.query.compound*), 49
Boosting (*class in pandagg.query*), 84
boosting() (*pandagg.query.Query method*), 79
boosting() (*pandagg.tree.query.Query method*), 59
BucketAggClause (*class in pandagg.node.aggs.abstract*), 31
BucketScript (*class in pandagg.aggs*), 69
BucketScript (*class in pandagg.node.aggs.pipeline*), 39
BucketSelector (*class in pandagg.aggs*), 69
BucketSelector (*class in pandagg.node.aggs.pipeline*), 39
BucketSort (*class in pandagg.aggs*), 69
BucketSort (*class in pandagg.node.aggs.pipeline*), 39
BucketsWrapperDict (*class in pandagg.types*), 94

Byte (*class in pandagg.mappings*), 75
Byte (*class in pandagg.node.mappings.field_datatypes*), 42

C

Cardinality (*class in pandagg.aggs*), 66
Cardinality (*class in pandagg.node.aggs.metric*), 37
Children (*class in pandagg.aggs*), 71
Children (*class in pandagg.node.aggs.bucket*), 32
client (*pandagg.discovery.Index attribute*), 72
Common (*class in pandagg.node.query.full_text*), 49
Common (*class in pandagg.query*), 84
Completion (*class in pandagg.mappings*), 76
Completion (*class in pandagg.node.mappings.field_datatypes*), 42
ComplexField (*class in pandagg.node.mappings.abstract*), 41
Composite (*class in pandagg.aggs*), 69
Composite (*class in pandagg.node.aggs.composite*), 36
CompoundClause (*class in pandagg.node.query.compound*), 49
constant_score() (*pandagg.query.Query method*), 79
constant_score() (*pandagg.tree.query.Query method*), 59
ConstantKeyword (*class in pandagg.mappings*), 74
ConstantKeyword (*class in pandagg.node.mappings.field_datatypes*), 42
ConstantScore (*class in pandagg.node.query.compound*), 49
ConstantScore (*class in pandagg.query*), 84
count() (*pandagg.search.Search method*), 89
CumulativeSum (*class in pandagg.aggs*), 69
CumulativeSum (*class in pandagg.node.aggs.pipeline*), 39

D

Date (*class in pandagg.mappings*), 75
Date (*class in pandagg.node.mappings.field_datatypes*), 42
DateHistogram (*class in pandagg.aggs*), 65
DateHistogram (*class in pandagg.node.aggs.bucket*), 33
DateNanos (*class in pandagg.mappings*), 75
DateNanos (*class in pandagg.node.mappings.field_datatypes*), 42
DateRange (*class in pandagg.mappings*), 76
DateRange (*class in pandagg.node.aggs.bucket*), 33
DateRange (*class in pandagg.node.mappings.field_datatypes*), 42
DEFAULT_OTHER_KEY (*pandagg.aggs.Filters attribute*), 64
DEFAULT_OTHER_KEY (*pandagg.node.aggs.bucket.Filters attribute*), 33
delete() (*pandagg.search.Search method*), 89
DeleteByQueryResponse (*class in pandagg.types*), 94
DenseVector (*class in pandagg.mappings*), 77
DenseVector (*class in pandagg.node.mappings.field_datatypes*), 43
Derivative (*class in pandagg.aggs*), 68
Derivative (*class in pandagg.node.aggs.pipeline*), 39
dis_max() (*pandagg.query.Query method*), 80
dis_max() (*pandagg.tree.query.Query method*), 59
discover() (*in module pandagg.discovery*), 72
DisMax (*class in pandagg.node.query.compound*), 49
DisMax (*class in pandagg.query*), 84
DistanceFeature (*class in pandagg.node.query.specialized*), 51
DistanceFeature (*class in pandagg.query*), 85
DiversifiedSampler (*class in pandagg.aggs*), 71
DiversifiedSampler (*class in pandagg.node.aggs.bucket*), 33
Double (*class in pandagg.mappings*), 75
Double (*class in pandagg.node.mappings.field_datatypes*), 43
DoubleRange (*class in pandagg.mappings*), 76
DoubleRange (*class in pandagg.node.mappings.field_datatypes*), 43
DslMeta (*class in pandagg.utils*), 95
DSLMixin (*class in pandagg.utils*), 95

E

equal_queries() (*in module pandagg.utils*), 95
equal_search() (*in module pandagg.utils*), 95
exclude() (*pandagg.search.Search method*), 89
execute() (*pandagg.search.MultiSearch method*), 87
execute() (*pandagg.search.Search method*), 89
Exists (*class in pandagg.node.query.term_level*), 52
Exists (*class in pandagg.query*), 82
ExtendedStats (*class in pandagg.aggs*), 66
ExtendedStats (*class in pandagg.node.aggs.metric*), 37
ExtendedStatsBucket (*class in pandagg.aggs*), 68
ExtendedStatsBucket (*class in pandagg.node.aggs.pipeline*), 40
extract_bucket_value() (*pandagg.node.aggs.abstract.AggClause class method*), 30
extract_bucket_value() (*pandagg.node.aggs.abstract.Root class*

method), 32
extract_buckets() (*pandagg.aggs.Composite method*), 70
extract_buckets() (*pandagg.node.aggs.abstract.AggClause method*), 30
extract_buckets() (*pandagg.node.aggs.abstract.BucketAggClause method*), 31
extract_buckets() (*pandagg.node.aggs.abstract.MetricAgg method*), 31
extract_buckets() (*pandagg.node.aggs.abstract.MultipleBucketAgg method*), 31
extract_buckets() (*pandagg.node.aggs.abstract.Root method*), 32
extract_buckets() (*pandagg.node.aggs.abstract.UniqueBucketAgg method*), 32
extract_buckets() (*pandagg.node.aggs.composite.Composite method*), 36

F

Field (*class in pandagg.node.mappings.abstract*), 41
FieldDict (*class in pandagg.types*), 94
FieldNames (*class in pandagg.mappings*), 78
FieldNames (*class in pandagg.node.mappings.meta_fields*), 46
FieldOrScriptMetricAgg (*class in pandagg.node.aggs.abstract*), 31
Filter (*class in pandagg.aggs*), 65
Filter (*class in pandagg.node.aggs.bucket*), 33
filter() (*pandagg.query.Query method*), 80
filter() (*pandagg.search.Search method*), 89
filter() (*pandagg.tree.query.Query method*), 59
Filters (*class in pandagg.aggs*), 64
Filters (*class in pandagg.node.aggs.bucket*), 33
FlatFieldQueryClause (*class in pandagg.node.query.abstract*), 47
Flattened (*class in pandagg.mappings*), 78
Flattened (*class in pandagg.node.mappings.field_datatypes*), 43
Float (*class in pandagg.mappings*), 75
Float (*class in pandagg.node.mappings.field_datatypes*), 43
FloatRange (*class in pandagg.mappings*), 76
FloatRange (*class in pandagg.node.mappings.field_datatypes*), 43
from_dict() (*pandagg.search.Search class method*), 89
function_score() (*pandagg.query.Query method*), 80
function_score() (*pandagg.tree.query.Query method*), 59
FunctionScore (*class in pandagg.node.query.compound*), 49
FunctionScore (*class in pandagg.query*), 84
Fuzzy (*class in pandagg.node.query.term_level*), 52
Fuzzy (*class in pandagg.query*), 82

G

GeoBound (*class in pandagg.aggs*), 67
GeoBound (*class in pandagg.node.aggs.metric*), 37
GeoBoundingBox (*class in pandagg.node.query.geo*), 50
GeoBoundingBox (*class in pandagg.query*), 85
GeoCentroid (*class in pandagg.aggs*), 67
GeoCentroid (*class in pandagg.node.aggs.metric*), 37
GeoDistance (*class in pandagg.aggs*), 70
GeoDistance (*class in pandagg.node.aggs.bucket*), 33
GeoDistance (*class in pandagg.node.query.geo*), 50
GeoDistance (*class in pandagg.query*), 85
GeoHashGrid (*class in pandagg.aggs*), 70
GeoHashGrid (*class in pandagg.node.aggs.bucket*), 34
GeoPoint (*class in pandagg.mappings*), 76
GeoPoint (*class in pandagg.node.mappings.field_datatypes*), 43
GeoPolygone (*class in pandagg.node.query.geo*), 50
GeoPolygone (*class in pandagg.query*), 85
GeoShape (*class in pandagg.mappings*), 76
GeoShape (*class in pandagg.node.mappings.field_datatypes*), 43
GeoShape (*class in pandagg.node.query.geo*), 51
GeoShape (*class in pandagg.query*), 85
GeoTileGrid (*class in pandagg.aggs*), 71
GeoTileGrid (*class in pandagg.node.aggs.bucket*), 34
get_action_modifier() (*in module pandagg.utils*), 95
get_composition_supporting_agg() (*pandagg.aggs.Aggs method*), 63
get_composition_supporting_agg() (*pandagg.tree.aggs.Aggs method*), 55
get_dsl_class() (*pandagg.utils.DSLMixin class method*), 95
get_dsl_type() (*pandagg.utils.DSLMixin static method*), 95
Global (*class in pandagg.aggs*), 65
Global (*class in pandagg.node.aggs.bucket*), 34
groupby() (*pandagg.aggs.Aggs method*), 63
groupby() (*pandagg.search.Search method*), 89
groupby() (*pandagg.tree.aggs.Aggs method*), 55
grouped_by() (*pandagg.aggs.Aggs method*), 63
grouped_by() (*pandagg.tree.aggs.Aggs method*), 56

H

HalfFloat (*class in pandagg.mappings*), 75
HalfFloat (*class in pandagg.node.mappings.field_datatypes*), 43
has_child() (*pandagg.query.Query method*), 80
has_child() (*pandagg.tree.query.Query method*), 59
has_parent() (*pandagg.query.Query method*), 80
has_parent() (*pandagg.tree.query.Query method*), 59
HasChild (*class in pandagg.node.query.joining*), 51
HasChild (*class in pandagg.query*), 84
HasParent (*class in pandagg.node.query.joining*), 51
HasParent (*class in pandagg.query*), 84
highlight() (*pandagg.search.Search method*), 90
highlight_options() (*pandagg.search.Search method*), 91
Histogram (*class in pandagg.aggs*), 65
Histogram (*class in pandagg.mappings*), 78
Histogram (*class in pandagg.node.aggs.bucket*), 34
Histogram (*class in pandagg.node.mappings.field_datatypes*), 43
Hit (*class in pandagg.response*), 86
HitDict (*class in pandagg.types*), 94
Hits (*class in pandagg.response*), 86
hits (*pandagg.response.Hits attribute*), 87
hits (*pandagg.response.SearchResponse attribute*), 87
HitsDict (*class in pandagg.types*), 94

I

Id (*class in pandagg.mappings*), 78
Id (*class in pandagg.node.mappings.meta_fields*), 46
id_from_key() (*pandagg.aggs.Aggs method*), 63
id_from_key() (*pandagg.tree.aggs.Aggs method*), 56
Ids (*class in pandagg.node.query.term_level*), 52
Ids (*class in pandagg.query*), 82
Ignored (*class in pandagg.mappings*), 78
Ignored (*class in pandagg.node.mappings.meta_fields*), 46
IMappings (*class in pandagg.interactive.mappings*), 29
IMappings (*class in pandagg.mappings*), 74
imappings (*pandagg.discovery.Index attribute*), 72
IMPLICIT_KEYED (*pandagg.aggs.Filters attribute*), 64
IMPLICIT_KEYED (*pandagg.node.aggs.abstract.MultipleBucketAgg attribute*), 31
IMPLICIT_KEYED (*pandagg.node.aggs.bucket.Filters attribute*), 33
Index (*class in pandagg.discovery*), 72
Index (*class in pandagg.mappings*), 78
Index (*class in pandagg.node.mappings.meta_fields*), 46
index() (*pandagg.search.Request method*), 87
Indices (*class in pandagg.discovery*), 72
Integer (*class in pandagg.mappings*), 75
Integer (*class in pandagg.node.mappings.field_datatypes*), 44
IntegerRange (*class in pandagg.mappings*), 75
IntegerRange (*class in pandagg.node.mappings.field_datatypes*), 44
Intervals (*class in pandagg.node.query.full_text*), 49
Intervals (*class in pandagg.query*), 83
InvalidAggregation, 72
InvalidOperationMappingFieldError, 72
IP (*class in pandagg.mappings*), 76
IP (*class in pandagg.node.mappings.field_datatypes*), 43
IPRange (*class in pandagg.aggs*), 71
IpRange (*class in pandagg.mappings*), 74
IPRange (*class in pandagg.node.aggs.bucket*), 34
IpRange (*class in pandagg.node.mappings.field_datatypes*), 44
is_convertible_to_composite_source() (*pandagg.aggs.DateHistogram method*), 65
is_convertible_to_composite_source() (*pandagg.aggs.Histogram method*), 65
is_convertible_to_composite_source() (*pandagg.aggs.Terms method*), 64
is_convertible_to_composite_source() (*pandagg.node.aggs.abstract.AggClause method*), 30
is_convertible_to_composite_source() (*pandagg.node.aggs.bucket.DateHistogram method*), 33
is_convertible_to_composite_source() (*pandagg.node.aggs.bucket.Histogram method*), 34
is_convertible_to_composite_source() (*pandagg.node.aggs.bucket.Terms method*), 36
is_subset() (*in module pandagg.utils*), 95
is_valid_value() (*pandagg.node.mappings.abstract.ComplexField method*), 41
is_valid_value() (*pandagg.node.mappings.abstract.Field method*), 41
is_valid_value() (*pandagg.node.mappings.abstract.RegularField method*), 42

J

Join (*class in pandagg.mappings*), 77
Join (*class in pandagg.node.mappings.field_datatypes*), 44

K

KEY (*pandagg.aggs.AdjacencyMatrix attribute*), 70
KEY (*pandagg.aggs.AutoDateHistogram attribute*), 70
KEY (*pandagg.aggs.Avg attribute*), 66
KEY (*pandagg.aggs.AvgBucket attribute*), 67
KEY (*pandagg.aggs.BucketScript attribute*), 69

KEY (*pandagg.aggs.BucketSelector attribute*), 69
KEY (*pandagg.aggs.BucketSort attribute*), 69
KEY (*pandagg.aggs.Cardinality attribute*), 66
KEY (*pandagg.aggs.Children attribute*), 71
KEY (*pandagg.aggs.Composite attribute*), 70
KEY (*pandagg.aggs.CumulativeSum attribute*), 69
KEY (*pandagg.aggs.DateHistogram attribute*), 65
KEY (*pandagg.aggs.Derivative attribute*), 68
KEY (*pandagg.aggs.DiversifiedSampler attribute*), 71
KEY (*pandagg.aggs.ExtendedStats attribute*), 66
KEY (*pandagg.aggs.ExtendedStatsBucket attribute*), 68
KEY (*pandagg.aggs.Filter attribute*), 65
KEY (*pandagg.aggs.Filters attribute*), 64
KEY (*pandagg.aggs.GeoBound attribute*), 67
KEY (*pandagg.aggs.GeoCentroid attribute*), 67
KEY (*pandagg.aggs.GeoDistance attribute*), 70
KEY (*pandagg.aggs.GeoHashGrid attribute*), 70
KEY (*pandagg.aggs.GeoTileGrid attribute*), 71
KEY (*pandagg.aggs.Global attribute*), 65
KEY (*pandagg.aggs.Histogram attribute*), 65
KEY (*pandagg.aggs.IPRange attribute*), 71
KEY (*pandagg.aggs.Max attribute*), 66
KEY (*pandagg.aggs.MaxBucket attribute*), 68
KEY (*pandagg.aggs.Min attribute*), 66
KEY (*pandagg.aggs.MinBucket attribute*), 68
KEY (*pandagg.aggs.Missing attribute*), 65
KEY (*pandagg.aggs.MovingAvg attribute*), 69
KEY (*pandagg.aggs.MultiTerms attribute*), 72
KEY (*pandagg.aggs.Nested attribute*), 65
KEY (*pandagg.aggs.Parent attribute*), 71
KEY (*pandagg.aggs.PercentileRanks attribute*), 67
KEY (*pandagg.aggs.Percentiles attribute*), 67
KEY (*pandagg.aggs.PercentilesBucket attribute*), 68
KEY (*pandagg.aggs.Range attribute*), 65
KEY (*pandagg.aggs.RareTerms attribute*), 71
KEY (*pandagg.aggs.ReverseNested attribute*), 65
KEY (*pandagg.aggs.Sampler attribute*), 71
KEY (*pandagg.aggs.SerialDiff attribute*), 69
KEY (*pandagg.aggs.SignificantTerms attribute*), 71
KEY (*pandagg.aggs.SignificantText attribute*), 72
KEY (*pandagg.aggs.Stats attribute*), 66
KEY (*pandagg.aggs.StatsBucket attribute*), 68
KEY (*pandagg.aggs.Sum attribute*), 66
KEY (*pandagg.aggs.SumBucket attribute*), 68
KEY (*pandagg.aggs.Terms attribute*), 64
KEY (*pandagg.aggs.TopHits attribute*), 67
KEY (*pandagg.aggs.ValueCount attribute*), 67
KEY (*pandagg.aggs.VariableWidthHistogram attribute*), 70
KEY (*pandagg.mappings.Alias attribute*), 77
KEY (*pandagg.mappings.Binary attribute*), 75
KEY (*pandagg.mappings.Boolean attribute*), 75
KEY (*pandagg.mappings.Byte attribute*), 75
KEY (*pandagg.mappings.Completion attribute*), 76
KEY (*pandagg.mappings.ConstantKeyword attribute*), 74
KEY (*pandagg.mappings.Date attribute*), 75
KEY (*pandagg.mappings.DateNanos attribute*), 75
KEY (*pandagg.mappings.DateRange attribute*), 76
KEY (*pandagg.mappings.DenseVector attribute*), 77
KEY (*pandagg.mappings.Double attribute*), 75
KEY (*pandagg.mappings.DoubleRange attribute*), 76
KEY (*pandagg.mappings.FieldNames attribute*), 78
KEY (*pandagg.mappings.Flattened attribute*), 78
KEY (*pandagg.mappings.Float attribute*), 75
KEY (*pandagg.mappings.FloatRange attribute*), 76
KEY (*pandagg.mappings.GeoPoint attribute*), 76
KEY (*pandagg.mappings.GeoShape attribute*), 76
KEY (*pandagg.mappings.HalfFloat attribute*), 75
KEY (*pandagg.mappings.Histogram attribute*), 78
KEY (*pandagg.mappings.Id attribute*), 78
KEY (*pandagg.mappings.Ignored attribute*), 79
KEY (*pandagg.mappings.Index attribute*), 78
KEY (*pandagg.mappings.Integer attribute*), 75
KEY (*pandagg.mappings.IntegerRange attribute*), 75
KEY (*pandagg.mappings.IP attribute*), 76
KEY (*pandagg.mappings.IpRange attribute*), 74
KEY (*pandagg.mappings.Join attribute*), 77
KEY (*pandagg.mappings.Keyword attribute*), 74
KEY (*pandagg.mappings.Long attribute*), 75
KEY (*pandagg.mappings.LongRange attribute*), 76
KEY (*pandagg.mappings.MapperAnnotatedText attribute*), 77
KEY (*pandagg.mappings.MapperMurMur3 attribute*), 77
KEY (*pandagg.mappings.Meta attribute*), 79
KEY (*pandagg.mappings.Nested attribute*), 76
KEY (*pandagg.mappings.Object attribute*), 76
KEY (*pandagg.mappings.Percolator attribute*), 77
KEY (*pandagg.mappings.RankFeature attribute*), 77
KEY (*pandagg.mappings.RankFeatures attribute*), 77
KEY (*pandagg.mappings.Routing attribute*), 79
KEY (*pandagg.mappings.ScaledFloat attribute*), 75
KEY (*pandagg.mappings.SearchAsYouType attribute*), 77
KEY (*pandagg.mappings.Shape attribute*), 78
KEY (*pandagg.mappings.Short attribute*), 75
KEY (*pandagg.mappings.Size attribute*), 78
KEY (*pandagg.mappings.Source attribute*), 78
KEY (*pandagg.mappings.SparseVector attribute*), 77
KEY (*pandagg.mappings.Text attribute*), 74
KEY (*pandagg.mappings.TokenCount attribute*), 76
KEY (*pandagg.mappings.Type attribute*), 78
KEY (*pandagg.mappings.WildCard attribute*), 75
KEY (*pandagg.node.aggs.abstract.Root attribute*), 32
KEY (*pandagg.node.aggs.bucket.AdjacencyMatrix attribute*), 32
KEY (*pandagg.node.aggs.bucket.AutoDateHistogram attribute*), 32
KEY (*pandagg.node.aggs.bucket.Children attribute*), 33

KEY (*pandagg.node.aggs.bucket.DateHistogram attribute*), 33
KEY (*pandagg.node.aggs.bucket.DateRange attribute*), 33
KEY (*pandagg.node.aggs.bucket.DiversifiedSampler attribute*), 33
KEY (*pandagg.node.aggs.bucket.Filter attribute*), 33
KEY (*pandagg.node.aggs.bucket.Filters attribute*), 33
KEY (*pandagg.node.aggs.bucket.GeoDistance attribute*), 34
KEY (*pandagg.node.aggs.bucket.GeoHashGrid attribute*), 34
KEY (*pandagg.node.aggs.bucket.GeoTileGrid attribute*), 34
KEY (*pandagg.node.aggs.bucket.Global attribute*), 34
KEY (*pandagg.node.aggs.bucket.Histogram attribute*), 34
KEY (*pandagg.node.aggs.bucket.IPRange attribute*), 34
KEY (*pandagg.node.aggs.bucket.Missing attribute*), 35
KEY (*pandagg.node.aggs.bucket.MultiTerms attribute*), 35
KEY (*pandagg.node.aggs.bucket.Nested attribute*), 35
KEY (*pandagg.node.aggs.bucket.Parent attribute*), 35
KEY (*pandagg.node.aggs.bucket.Range attribute*), 35
KEY (*pandagg.node.aggs.bucket.RareTerms attribute*), 35
KEY (*pandagg.node.aggs.bucket.ReverseNested attribute*), 35
KEY (*pandagg.node.aggs.bucket.Sampler attribute*), 35
KEY (*pandagg.node.aggs.bucket.SignificantTerms attribute*), 36
KEY (*pandagg.node.aggs.bucket.SignificantText attribute*), 36
KEY (*pandagg.node.aggs.bucket.Terms attribute*), 36
KEY (*pandagg.node.aggs.bucket.VariableWidthHistogram attribute*), 36
KEY (*pandagg.node.aggs.composite.Composite attribute*), 36
KEY (*pandagg.node.aggs.metric.Avg attribute*), 37
KEY (*pandagg.node.aggs.metric.Cardinality attribute*), 37
KEY (*pandagg.node.aggs.metric.ExtendedStats attribute*), 37
KEY (*pandagg.node.aggs.metric.GeoBound attribute*), 37
KEY (*pandagg.node.aggs.metric.GeoCentroid attribute*), 37
KEY (*pandagg.node.aggs.metric.Max attribute*), 37
KEY (*pandagg.node.aggs.metric.Min attribute*), 37
KEY (*pandagg.node.aggs.metric.PercentileRanks attribute*), 38
KEY (*pandagg.node.aggs.metric.Percentiles attribute*), 38
KEY (*pandagg.node.aggs.metric.Stats attribute*), 38
KEY (*pandagg.node.aggs.metric.Sum attribute*), 38
KEY (*pandagg.node.aggs.metric.TopHits attribute*), 38
KEY (*pandagg.node.aggs.metric.ValueCount attribute*), 38
KEY (*pandagg.node.aggs.pipeline.AvgBucket attribute*), 39
KEY (*pandagg.node.aggs.pipeline.BucketScript attribute*), 39
KEY (*pandagg.node.aggs.pipeline.BucketSelector attribute*), 39
KEY (*pandagg.node.aggs.pipeline.BucketSort attribute*), 39
KEY (*pandagg.node.aggs.pipeline.CumulativeSum attribute*), 39
KEY (*pandagg.node.aggs.pipeline.Derivative attribute*), 40
KEY (*pandagg.node.aggs.pipeline.ExtendedStatsBucket attribute*), 40
KEY (*pandagg.node.aggs.pipeline.MaxBucket attribute*), 40
KEY (*pandagg.node.aggs.pipeline.MinBucket attribute*), 40
KEY (*pandagg.node.aggs.pipeline.MovingAvg attribute*), 40
KEY (*pandagg.node.aggs.pipeline.PercentilesBucket attribute*), 40
KEY (*pandagg.node.aggs.pipeline.SerialDiff attribute*), 41
KEY (*pandagg.node.aggs.pipeline.StatsBucket attribute*), 41
KEY (*pandagg.node.aggs.pipeline.SumBucket attribute*), 41
KEY (*pandagg.node.mappings.abstract.Root attribute*), 42
KEY (*pandagg.node.mappings.field_datatypes.Alias attribute*), 42
KEY (*pandagg.node.mappings.field_datatypes.Binary attribute*), 42
KEY (*pandagg.node.mappings.field_datatypes.Boolean attribute*), 42
KEY (*pandagg.node.mappings.field_datatypes.Byte attribute*), 42
KEY (*pandagg.node.mappings.field_datatypes.Completion attribute*), 42
KEY (*pandagg.node.mappings.field_datatypes.ConstantKeyword attribute*), 42
KEY (*pandagg.node.mappings.field_datatypes.Date attribute*), 42
KEY (*pandagg.node.mappings.field_datatypes.DateNanos attribute*), 42
KEY (*pandagg.node.mappings.field_datatypes.DateRange attribute*), 43
KEY (*pandagg.node.mappings.field_datatypes.DenseVector attribute*), 43
KEY (*pandagg.node.mappings.field_datatypes.Double attribute*), 43
KEY (*pandagg.node.mappings.field_datatypes.DoubleRange attribute*), 43

KEY (*pandagg.node.mappings.field_datatypes.Flattened attribute*), 43
KEY (*pandagg.node.mappings.field_datatypes.Float attribute*), 43
KEY (*pandagg.node.mappings.field_datatypes.FloatRange attribute*), 43
KEY (*pandagg.node.mappings.field_datatypes.GeoPoint attribute*), 43
KEY (*pandagg.node.mappings.field_datatypes.GeoShape attribute*), 43
KEY (*pandagg.node.mappings.field_datatypes.HalfFloat attribute*), 43
KEY (*pandagg.node.mappings.field_datatypes.Histogram attribute*), 43
KEY (*pandagg.node.mappings.field_datatypes.Integer attribute*), 44
KEY (*pandagg.node.mappings.field_datatypes.IntegerRange attribute*), 44
KEY (*pandagg.node.mappings.field_datatypes.IP attribute*), 44
KEY (*pandagg.node.mappings.field_datatypes.IpRange attribute*), 44
KEY (*pandagg.node.mappings.field_datatypes.Join attribute*), 44
KEY (*pandagg.node.mappings.field_datatypes.Keyword attribute*), 44
KEY (*pandagg.node.mappings.field_datatypes.Long attribute*), 44
KEY (*pandagg.node.mappings.field_datatypes.LongRange attribute*), 44
KEY (*pandagg.node.mappings.field_datatypes.MapperAnnotatedText attribute*), 44
KEY (*pandagg.node.mappings.field_datatypes.MapperMurMur3 attribute*), 44
KEY (*pandagg.node.mappings.field_datatypes.Nested attribute*), 44
KEY (*pandagg.node.mappings.field_datatypes.Object attribute*), 45
KEY (*pandagg.node.mappings.field_datatypes.Percolator attribute*), 45
KEY (*pandagg.node.mappings.field_datatypes.RankFeature attribute*), 45
KEY (*pandagg.node.mappings.field_datatypes.RankFeatures attribute*), 45
KEY (*pandagg.node.mappings.field_datatypes.ScaledFloat attribute*), 45
KEY (*pandagg.node.mappings.field_datatypes.SearchAsYouType attribute*), 45
KEY (*pandagg.node.mappings.field_datatypes.Shape attribute*), 45
KEY (*pandagg.node.mappings.field_datatypes.Short attribute*), 45
KEY (*pandagg.node.mappings.field_datatypes.SparseVector attribute*), 45
KEY (*pandagg.node.mappings.field_datatypes.Text attribute*), 45
KEY (*pandagg.node.mappings.field_datatypes.TokenCount attribute*), 45
KEY (*pandagg.node.mappings.field_datatypes.WildCard attribute*), 46
KEY (*pandagg.node.mappings.meta_fields.FieldNames attribute*), 46
KEY (*pandagg.node.mappings.meta_fields.Id attribute*), 46
KEY (*pandagg.node.mappings.meta_fields.Ignored attribute*), 46
KEY (*pandagg.node.mappings.meta_fields.Index attribute*), 46
KEY (*pandagg.node.mappings.meta_fields.Meta attribute*), 46
KEY (*pandagg.node.mappings.meta_fields.Routing attribute*), 46
KEY (*pandagg.node.mappings.meta_fields.Size attribute*), 46
KEY (*pandagg.node.mappings.meta_fields.Source attribute*), 47
KEY (*pandagg.node.mappings.meta_fields.Type attribute*), 47
KEY (*pandagg.node.query.compound.Bool attribute*), 49
KEY (*pandagg.node.query.compound.Boosting attribute*), 49
KEY (*pandagg.node.query.compound.ConstantScore attribute*), 49
KEY (*pandagg.node.query.compound.DisMax attribute*), 49
KEY (*pandagg.node.query.compound.FunctionScore attribute*), 49
KEY (*pandagg.node.query.full_text.Common attribute*), 49
KEY (*pandagg.node.query.full_text.Intervals attribute*), 49
KEY (*pandagg.node.query.full_text.Match attribute*), 49
KEY (*pandagg.node.query.full_text.MatchBoolPrefix attribute*), 50
KEY (*pandagg.node.query.full_text.MatchPhrase attribute*), 50
KEY (*pandagg.node.query.full_text.MatchPhrasePrefix attribute*), 50
KEY (*pandagg.node.query.full_text.MultiMatch attribute*), 50
KEY (*pandagg.node.query.full_text.QueryString attribute*), 50
KEY (*pandagg.node.query.full_text.SimpleQueryString attribute*), 50
KEY (*pandagg.node.query.geo.GeoBoundingBox attribute*), 50
KEY (*pandagg.node.query.geo.GeoDistance attribute*), 50

KEY (*pandagg.node.query.geo.GeoPolygone attribute*), 51
KEY (*pandagg.node.query.geo.GeoShape attribute*), 51
KEY (*pandagg.node.query.joining.HasChild attribute*), 51
KEY (*pandagg.node.query.joining.HasParent attribute*), 51
KEY (*pandagg.node.query.joining.Nested attribute*), 51
KEY (*pandagg.node.query.joining.ParentId attribute*), 51
KEY (*pandagg.node.query.shape.Shape attribute*), 51
KEY (*pandagg.node.query.specialized.DistanceFeature attribute*), 51
KEY (*pandagg.node.query.specialized.MoreLikeThis attribute*), 51
KEY (*pandagg.node.query.specialized.Percolate attribute*), 52
KEY (*pandagg.node.query.specialized.RankFeature attribute*), 52
KEY (*pandagg.node.query.specialized.Script attribute*), 52
KEY (*pandagg.node.query.specialized.Wrapper attribute*), 52
KEY (*pandagg.node.query.specialized_compound.PinnedQuery attribute*), 52
KEY (*pandagg.node.query.specialized_compound.ScriptScore attribute*), 52
KEY (*pandagg.node.query.term_level.Exists attribute*), 52
KEY (*pandagg.node.query.term_level.Fuzzy attribute*), 52
KEY (*pandagg.node.query.term_level.Ids attribute*), 52
KEY (*pandagg.node.query.term_level.Prefix attribute*), 53
KEY (*pandagg.node.query.term_level.Range attribute*), 53
KEY (*pandagg.node.query.term_level.Regexp attribute*), 53
KEY (*pandagg.node.query.term_level.Term attribute*), 53
KEY (*pandagg.node.query.term_level.Terms attribute*), 53
KEY (*pandagg.node.query.term_level.TermsSet attribute*), 53
KEY (*pandagg.node.query.term_level.Type attribute*), 53
KEY (*pandagg.node.query.term_level.Wildcard attribute*), 53
KEY (*pandagg.query.Bool attribute*), 84
KEY (*pandagg.query.Boosting attribute*), 84
KEY (*pandagg.query.Common attribute*), 84
KEY (*pandagg.query.ConstantScore attribute*), 84
KEY (*pandagg.query.DisMax attribute*), 84
KEY (*pandagg.query.DistanceFeature attribute*), 85
KEY (*pandagg.query.Exists attribute*), 82
KEY (*pandagg.query.FunctionScore attribute*), 84
KEY (*pandagg.query.Fuzzy attribute*), 82
KEY (*pandagg.query.GeoBoundingBox attribute*), 85
KEY (*pandagg.query.GeoDistance attribute*), 85
KEY (*pandagg.query.GeoPolygone attribute*), 85
KEY (*pandagg.query.GeoShape attribute*), 85
KEY (*pandagg.query.HasChild attribute*), 84
KEY (*pandagg.query.HasParent attribute*), 84
KEY (*pandagg.query.Ids attribute*), 82
KEY (*pandagg.query.Intervals attribute*), 83
KEY (*pandagg.query.Match attribute*), 83
KEY (*pandagg.query.MatchBoolPrefix attribute*), 83
KEY (*pandagg.query.MatchPhrase attribute*), 83
KEY (*pandagg.query.MatchPhrasePrefix attribute*), 83
KEY (*pandagg.query.MoreLikeThis attribute*), 85
KEY (*pandagg.query.MultiMatch attribute*), 83
KEY (*pandagg.query.Nested attribute*), 84
KEY (*pandagg.query.ParentId attribute*), 84
KEY (*pandagg.query.Percolate attribute*), 85
KEY (*pandagg.query.PinnedQuery attribute*), 86
KEY (*pandagg.query.Prefix attribute*), 82
KEY (*pandagg.query.QueryString attribute*), 84
KEY (*pandagg.query.Range attribute*), 82
KEY (*pandagg.query.RankFeature attribute*), 85
KEY (*pandagg.query.Regexp attribute*), 83
KEY (*pandagg.query.Script attribute*), 85
KEY (*pandagg.query.ScriptScore attribute*), 85
KEY (*pandagg.query.Shape attribute*), 85
KEY (*pandagg.query.SimpleQueryString attribute*), 84
KEY (*pandagg.query.Term attribute*), 83
KEY (*pandagg.query.Terms attribute*), 83
KEY (*pandagg.query.TermsSet attribute*), 83
KEY (*pandagg.query.Type attribute*), 83
KEY (*pandagg.query.Wildcard attribute*), 83
KEY (*pandagg.query.Wrapper attribute*), 85
KEY (*pandagg.utils.DslMeta attribute*), 95
KeyFieldQueryClause (*class in pandagg.node.query.abstract*), 47
keys() (*pandagg.response.Aggregations method*), 86
Keyword (*class in pandagg.mappings*), 74
Keyword (*class in pandagg.node.mappings.field_datatypes*), 44

L

LeafQueryClause (*class in pandagg.node.query.abstract*), 48
line_repr() (*pandagg.node.aggs.abstract.AggClause method*), 30
line_repr() (*pandagg.node.aggs.abstract.Root method*), 32
line_repr() (*pandagg.node.mappings.abstract.Field method*), 41
line_repr() (*pandagg.node.mappings.abstract.Root method*), 42
line_repr() (*pandagg.node.query.abstract.KeyFieldQueryClause method*), 48
line_repr() (*pandagg.node.query.abstract.MultiFieldsQueryClause method*), 48
line_repr() (*pandagg.node.query.abstract.ParentParameterClause method*), 48

line_repr() (*pandagg.node.query.abstract.QueryClause method*), 48
line_repr() (*pandagg.node.query.geo.GeoDistance method*), 50
line_repr() (*pandagg.node.query.term_level.Exists method*), 52
line_repr() (*pandagg.node.query.term_level.Ids method*), 52
line_repr() (*pandagg.query.Exists method*), 82
line_repr() (*pandagg.query.GeoDistance method*), 85
line_repr() (*pandagg.query.Ids method*), 82
list_nesteds_at_field() (*pandagg.mappings.Mappings method*), 73
list_nesteds_at_field() (*pandagg.tree.mappings.Mappings method*), 57
Long (*class in pandagg.mappings*), 75
Long (*class in pandagg.node.mappings.field_datatypes*), 44
LongRange (*class in pandagg.mappings*), 76
LongRange (*class in pandagg.node.mappings.field_datatypes*), 44

M

MapperAnnotatedText (*class in pandagg.mappings*), 77
MapperAnnotatedText (*class in pandagg.node.mappings.field_datatypes*), 44
MapperMurMur3 (*class in pandagg.mappings*), 76
MapperMurMur3 (*class in pandagg.node.mappings.field_datatypes*), 44
mapping_type_of_field() (*pandagg.mappings.Mappings method*), 73
mapping_type_of_field() (*pandagg.tree.mappings.Mappings method*), 57
MappingError, 72
Mappings (*class in pandagg.mappings*), 73
Mappings (*class in pandagg.tree.mappings*), 57
MappingsDict (*class in pandagg.types*), 94
MappingsDictOrNode (*class in pandagg.tree.mappings*), 58
Match (*class in pandagg.node.query.full_text*), 49
Match (*class in pandagg.query*), 83
MatchAll (*class in pandagg.aggs*), 69
MatchAll (*class in pandagg.node.aggs.bucket*), 34
MatchBoolPrefix (*class in pandagg.node.query.full_text*), 49
MatchBoolPrefix (*class in pandagg.query*), 83
MatchPhrase (*class in pandagg.node.query.full_text*), 50
MatchPhrase (*class in pandagg.query*), 83
MatchPhrasePrefix (*class in pandagg.node.query.full_text*), 50
MatchPhrasePrefix (*class in pandagg.query*), 83
Max (*class in pandagg.aggs*), 66
Max (*class in pandagg.node.aggs.metric*), 37
max_score (*pandagg.response.Hits attribute*), 87
MaxBucket (*class in pandagg.aggs*), 68
MaxBucket (*class in pandagg.node.aggs.pipeline*), 40
Meta (*class in pandagg.mappings*), 79
Meta (*class in pandagg.node.mappings.meta_fields*), 46
MetricAgg (*class in pandagg.node.aggs.abstract*), 31
Min (*class in pandagg.aggs*), 66
Min (*class in pandagg.node.aggs.metric*), 37
MinBucket (*class in pandagg.aggs*), 68
MinBucket (*class in pandagg.node.aggs.pipeline*), 40
Missing (*class in pandagg.aggs*), 65
Missing (*class in pandagg.node.aggs.bucket*), 35
MoreLikeThis (*class in pandagg.node.query.specialized*), 51
MoreLikeThis (*class in pandagg.query*), 85
MovingAvg (*class in pandagg.aggs*), 69
MovingAvg (*class in pandagg.node.aggs.pipeline*), 40
MultiFieldsQueryClause (*class in pandagg.node.query.abstract*), 48
MultiMatch (*class in pandagg.node.query.full_text*), 50
MultiMatch (*class in pandagg.query*), 83
MultipleBucketAgg (*class in pandagg.node.aggs.abstract*), 31
MultiSearch (*class in pandagg.search*), 87
MultiTerms (*class in pandagg.aggs*), 72
MultiTerms (*class in pandagg.node.aggs.bucket*), 35
must() (*pandagg.query.Query method*), 80
must() (*pandagg.search.Search method*), 91
must() (*pandagg.tree.query.Query method*), 59
must_not() (*pandagg.query.Query method*), 80
must_not() (*pandagg.search.Search method*), 91
must_not() (*pandagg.tree.query.Query method*), 60

N

name (*pandagg.node.query.abstract.QueryClause attribute*), 48
Nested (*class in pandagg.aggs*), 65
Nested (*class in pandagg.mappings*), 76
Nested (*class in pandagg.node.aggs.bucket*), 35
Nested (*class in pandagg.node.mappings.field_datatypes*), 44
Nested (*class in pandagg.node.query.joining*), 51
Nested (*class in pandagg.query*), 84
nested() (*pandagg.query.Query method*), 81
nested() (*pandagg.tree.query.Query method*), 60

nested_at_field() (*pandagg.mappings.Mappings method*), 73
nested_at_field() (*pandagg.tree.mappings.Mappings method*), 57
NormalizedBucketDict (*class in pandagg.response*), 87

O

Object (*class in pandagg.mappings*), 76
Object (*class in pandagg.node.mappings.field_datatypes*), 44
ordered() (*in module pandagg.utils*), 95

P

pandagg (*module*), 95
pandagg.aggs (*module*), 61
pandagg.discovery (*module*), 72
pandagg.exceptions (*module*), 72
pandagg.interactive (*module*), 29
pandagg.interactive.mappings (*module*), 29
pandagg.mappings (*module*), 73
pandagg.node (*module*), 54
pandagg.node.aggs (*module*), 41
pandagg.node.aggs.abstract (*module*), 30
pandagg.node.aggs.bucket (*module*), 32
pandagg.node.aggs.composite (*module*), 36
pandagg.node.aggs.metric (*module*), 37
pandagg.node.aggs.pipeline (*module*), 39
pandagg.node.mappings (*module*), 47
pandagg.node.mappings.abstract (*module*), 41
pandagg.node.mappings.field_datatypes (*module*), 42
pandagg.node.mappings.meta_fields (*module*), 46
pandagg.node.query (*module*), 53
pandagg.node.query.abstract (*module*), 47
pandagg.node.query.compound (*module*), 49
pandagg.node.query.full_text (*module*), 49
pandagg.node.query.geo (*module*), 50
pandagg.node.query.joining (*module*), 51
pandagg.node.query.shape (*module*), 51
pandagg.node.query.span (*module*), 51
pandagg.node.query.specialized (*module*), 51
pandagg.node.query.specialized_compound (*module*), 52
pandagg.node.query.term_level (*module*), 52
pandagg.node.types (*module*), 54
pandagg.query (*module*), 79
pandagg.response (*module*), 86
pandagg.search (*module*), 87
pandagg.tree (*module*), 61
pandagg.tree.aggs (*module*), 54
pandagg.tree.mappings (*module*), 57
pandagg.tree.query (*module*), 58
pandagg.types (*module*), 94
pandagg.utils (*module*), 95
params() (*pandagg.search.Request method*), 88
Parent (*class in pandagg.aggs*), 71
Parent (*class in pandagg.node.aggs.bucket*), 35
ParentId (*class in pandagg.node.query.joining*), 51
ParentId (*class in pandagg.query*), 84
ParentParameterClause (*class in pandagg.node.query.abstract*), 48
parse_group_by() (*pandagg.response.Aggregations method*), 86
PercentileRanks (*class in pandagg.aggs*), 67
PercentileRanks (*class in pandagg.node.aggs.metric*), 38
Percentiles (*class in pandagg.aggs*), 67
Percentiles (*class in pandagg.node.aggs.metric*), 38
PercentilesBucket (*class in pandagg.aggs*), 68
PercentilesBucket (*class in pandagg.node.aggs.pipeline*), 40
Percolate (*class in pandagg.node.query.specialized*), 51
Percolate (*class in pandagg.query*), 85
Percolator (*class in pandagg.mappings*), 77
Percolator (*class in pandagg.node.mappings.field_datatypes*), 45
pinned_query() (*pandagg.query.Query method*), 81
pinned_query() (*pandagg.tree.query.Query method*), 60
PinnedQuery (*class in pandagg.node.query.specialized_compound*), 52
PinnedQuery (*class in pandagg.query*), 86
Pipeline (*class in pandagg.node.aggs.abstract*), 31
PointInTimeDict (*class in pandagg.types*), 94
post_filter() (*pandagg.search.Search method*), 91
Prefix (*class in pandagg.node.query.term_level*), 53
Prefix (*class in pandagg.query*), 82
profile (*pandagg.response.SearchResponse attribute*), 87
ProfileDict (*class in pandagg.types*), 94
ProfileShardDict (*class in pandagg.types*), 94

Q

Q() (*in module pandagg.node.query.abstract*), 48
Query (*class in pandagg.query*), 79
Query (*class in pandagg.tree.query*), 58
query() (*pandagg.query.Query method*), 81
query() (*pandagg.search.Search method*), 91
query() (*pandagg.tree.query.Query method*), 60
QueryClause (*class in pandagg.node.query.abstract*), 48

QueryString (*class in pandagg.node.query.full_text*), 50
QueryString (*class in pandagg.query*), 84

R

Range (*class in pandagg.aggs*), 65
Range (*class in pandagg.node.aggs.bucket*), 35
Range (*class in pandagg.node.query.term_level*), 53
Range (*class in pandagg.query*), 82
RangeDict (*class in pandagg.types*), 94
RankFeature (*class in pandagg.mappings*), 77
RankFeature (*class in pandagg.node.mappings.field_datatypes*), 45
RankFeature (*class in pandagg.node.query.specialized*), 52
RankFeature (*class in pandagg.query*), 85
RankFeatures (*class in pandagg.mappings*), 77
RankFeatures (*class in pandagg.node.mappings.field_datatypes*), 45
RareTerms (*class in pandagg.aggs*), 71
RareTerms (*class in pandagg.node.aggs.bucket*), 35
Regexp (*class in pandagg.node.query.term_level*), 53
Regexp (*class in pandagg.query*), 82
RegularField (*class in pandagg.node.mappings.abstract*), 41
Request (*class in pandagg.search*), 87
RetriesDict (*class in pandagg.types*), 94
ReverseNested (*class in pandagg.aggs*), 65
ReverseNested (*class in pandagg.node.aggs.bucket*), 35
Root (*class in pandagg.node.aggs.abstract*), 31
Root (*class in pandagg.node.mappings.abstract*), 42
Routing (*class in pandagg.mappings*), 79
Routing (*class in pandagg.node.mappings.meta_fields*), 46
RunTimeMappingDict (*class in pandagg.types*), 94

S

Sampler (*class in pandagg.aggs*), 71
Sampler (*class in pandagg.node.aggs.bucket*), 35
ScaledFloat (*class in pandagg.mappings*), 75
ScaledFloat (*class in pandagg.node.mappings.field_datatypes*), 45
scan() (*pandagg.search.Search method*), 92
scan_composite_agg() (*pandagg.search.Search method*), 92
scan_composite_agg_at_once() (*pandagg.search.Search method*), 92
Script (*class in pandagg.node.query.specialized*), 52
Script (*class in pandagg.query*), 85
Script (*class in pandagg.types*), 94
script_fields() (*pandagg.search.Search method*), 92
script_score() (*pandagg.query.Query method*), 81
script_score() (*pandagg.tree.query.Query method*), 61
ScriptPipeline (*class in pandagg.node.aggs.abstract*), 32
ScriptScore (*class in pandagg.node.query.specialized_compound*), 52
ScriptScore (*class in pandagg.query*), 85
Search (*class in pandagg.search*), 88
search() (*pandagg.discovery.Index method*), 72
SearchAsYouType (*class in pandagg.mappings*), 77
SearchAsYouType (*class in pandagg.node.mappings.field_datatypes*), 45
SearchDict (*class in pandagg.types*), 95
SearchResponse (*class in pandagg.response*), 87
SearchResponseDict (*class in pandagg.types*), 95
SerialDiff (*class in pandagg.aggs*), 69
SerialDiff (*class in pandagg.node.aggs.pipeline*), 41
Shape (*class in pandagg.mappings*), 78
Shape (*class in pandagg.node.mappings.field_datatypes*), 45
Shape (*class in pandagg.node.query.shape*), 51
Shape (*class in pandagg.query*), 84
ShardsDict (*class in pandagg.types*), 95
Short (*class in pandagg.mappings*), 75
Short (*class in pandagg.node.mappings.field_datatypes*), 45
should() (*pandagg.query.Query method*), 81
should() (*pandagg.search.Search method*), 93
should() (*pandagg.tree.query.Query method*), 61
show() (*pandagg.aggs.Aggs method*), 64
show() (*pandagg.query.Query method*), 82
show() (*pandagg.tree.aggs.Aggs method*), 56
show() (*pandagg.tree.query.Query method*), 61
SignificantTerms (*class in pandagg.aggs*), 71
SignificantTerms (*class in pandagg.node.aggs.bucket*), 36
SignificantText (*class in pandagg.aggs*), 72
SignificantText (*class in pandagg.node.aggs.bucket*), 36
SimpleQueryString (*class in pandagg.node.query.full_text*), 50
SimpleQueryString (*class in pandagg.query*), 84
Size (*class in pandagg.mappings*), 78
Size (*class in pandagg.node.mappings.meta_fields*), 46
size (*pandagg.aggs.Composite attribute*), 70
size (*pandagg.node.aggs.composite.Composite attribute*), 36
size() (*pandagg.search.Search method*), 93
sort() (*pandagg.search.Search method*), 93

Source (*class in pandagg.mappings*), 78
Source (*class in pandagg.node.mappings.meta_fields*), 46
source() (*pandagg.search.Search method*), 93
source_names (*pandagg.aggs.Composite attribute*), 70
source_names (*pandagg.node.aggs.composite.Composite attribute*), 36
SourceIncludeDict (*class in pandagg.types*), 95
sources (*pandagg.aggs.Composite attribute*), 70
sources (*pandagg.node.aggs.composite.Composite attribute*), 36
SparseVector (*class in pandagg.mappings*), 77
SparseVector (*class in pandagg.node.mappings.field_datatypes*), 45
Stats (*class in pandagg.aggs*), 66
Stats (*class in pandagg.node.aggs.metric*), 38
StatsBucket (*class in pandagg.aggs*), 68
StatsBucket (*class in pandagg.node.aggs.pipeline*), 41
success (*pandagg.response.SearchResponse attribute*), 87
suggest() (*pandagg.search.Search method*), 93
SuggestedItemDict (*class in pandagg.types*), 95
Sum (*class in pandagg.aggs*), 66
Sum (*class in pandagg.node.aggs.metric*), 38
SumBucket (*class in pandagg.aggs*), 68
SumBucket (*class in pandagg.node.aggs.pipeline*), 41

T

Term (*class in pandagg.node.query.term_level*), 53
Term (*class in pandagg.query*), 83
Terms (*class in pandagg.aggs*), 64
Terms (*class in pandagg.node.aggs.bucket*), 36
Terms (*class in pandagg.node.query.term_level*), 53
Terms (*class in pandagg.query*), 83
TermsSet (*class in pandagg.node.query.term_level*), 53
TermsSet (*class in pandagg.query*), 83
Text (*class in pandagg.mappings*), 74
Text (*class in pandagg.node.mappings.field_datatypes*), 45
timed_out (*pandagg.response.SearchResponse attribute*), 87
to_dataframe() (*pandagg.response.Aggregations method*), 86
to_dataframe() (*pandagg.response.Hits method*), 87
to_dict() (*pandagg.aggs.Aggs method*), 64
to_dict() (*pandagg.mappings.Mappings method*), 74
to_dict() (*pandagg.node.aggs.abstract.AggClause method*), 30
to_dict() (*pandagg.node.mappings.abstract.Field method*), 41
to_dict() (*pandagg.node.query.abstract.QueryClause method*), 48
to_dict() (*pandagg.query.Query method*), 82
to_dict() (*pandagg.search.MultiSearch method*), 87
to_dict() (*pandagg.search.Search method*), 94
to_dict() (*pandagg.tree.aggs.Aggs method*), 57
to_dict() (*pandagg.tree.mappings.Mappings method*), 58
to_dict() (*pandagg.tree.query.Query method*), 61
to_normalized() (*pandagg.response.Aggregations method*), 86
to_tabular() (*pandagg.response.Aggregations method*), 86
TokenCount (*class in pandagg.mappings*), 76
TokenCount (*class in pandagg.node.mappings.field_datatypes*), 45
took (*pandagg.response.SearchResponse attribute*), 87
TopHits (*class in pandagg.aggs*), 67
TopHits (*class in pandagg.node.aggs.metric*), 38
total (*pandagg.response.Hits attribute*), 87
TotalDict (*class in pandagg.types*), 95
Type (*class in pandagg.mappings*), 78
Type (*class in pandagg.node.mappings.meta_fields*), 47
Type (*class in pandagg.node.query.term_level*), 53
Type (*class in pandagg.query*), 83

U

UniqueBucketAgg (*class in pandagg.node.aggs.abstract*), 32
update_from_dict() (*pandagg.search.Search method*), 94
using() (*pandagg.search.Request method*), 88

V

valid_on_field_type() (*pandagg.node.aggs.abstract.AggClause class method*), 31
validate_agg_clause() (*pandagg.mappings.Mappings method*), 74
validate_agg_clause() (*pandagg.tree.mappings.Mappings method*), 58
validate_document() (*pandagg.mappings.Mappings method*), 74
validate_document() (*pandagg.tree.mappings.Mappings method*), 58
VALUE_ATTRS (*pandagg.aggs.AdjacencyMatrix attribute*), 70
VALUE_ATTRS (*pandagg.aggs.AutoDateHistogram attribute*), 70
VALUE_ATTRS (*pandagg.aggs.Avg attribute*), 66
VALUE_ATTRS (*pandagg.aggs.AvgBucket attribute*), 67

VALUE_ATTRS (*pandagg.aggs.BucketScript attribute*), 69
VALUE_ATTRS (*pandagg.aggs.BucketSelector attribute*), 69
VALUE_ATTRS (*pandagg.aggs.BucketSort attribute*), 69
VALUE_ATTRS (*pandagg.aggs.Cardinality attribute*), 66
VALUE_ATTRS (*pandagg.aggs.Children attribute*), 71
VALUE_ATTRS (*pandagg.aggs.Composite attribute*), 70
VALUE_ATTRS (*pandagg.aggs.CumulativeSum attribute*), 69
VALUE_ATTRS (*pandagg.aggs.DateHistogram attribute*), 65
VALUE_ATTRS (*pandagg.aggs.Derivative attribute*), 68
VALUE_ATTRS (*pandagg.aggs.DiversifiedSampler attribute*), 71
VALUE_ATTRS (*pandagg.aggs.ExtendedStats attribute*), 66
VALUE_ATTRS (*pandagg.aggs.ExtendedStatsBucket attribute*), 68
VALUE_ATTRS (*pandagg.aggs.Filter attribute*), 65
VALUE_ATTRS (*pandagg.aggs.Filters attribute*), 64
VALUE_ATTRS (*pandagg.aggs.GeoBound attribute*), 67
VALUE_ATTRS (*pandagg.aggs.GeoCentroid attribute*), 67
VALUE_ATTRS (*pandagg.aggs.GeoDistance attribute*), 70
VALUE_ATTRS (*pandagg.aggs.GeoHashGrid attribute*), 70
VALUE_ATTRS (*pandagg.aggs.GeoTileGrid attribute*), 71
VALUE_ATTRS (*pandagg.aggs.Global attribute*), 65
VALUE_ATTRS (*pandagg.aggs.Histogram attribute*), 65
VALUE_ATTRS (*pandagg.aggs.IPRange attribute*), 71
VALUE_ATTRS (*pandagg.aggs.Max attribute*), 66
VALUE_ATTRS (*pandagg.aggs.MaxBucket attribute*), 68
VALUE_ATTRS (*pandagg.aggs.Min attribute*), 66
VALUE_ATTRS (*pandagg.aggs.MinBucket attribute*), 68
VALUE_ATTRS (*pandagg.aggs.Missing attribute*), 65
VALUE_ATTRS (*pandagg.aggs.MovingAvg attribute*), 69
VALUE_ATTRS (*pandagg.aggs.MultiTerms attribute*), 72
VALUE_ATTRS (*pandagg.aggs.Nested attribute*), 65
VALUE_ATTRS (*pandagg.aggs.Parent attribute*), 72
VALUE_ATTRS (*pandagg.aggs.PercentileRanks attribute*), 67
VALUE_ATTRS (*pandagg.aggs.Percentiles attribute*), 67
VALUE_ATTRS (*pandagg.aggs.PercentilesBucket attribute*), 68
VALUE_ATTRS (*pandagg.aggs.Range attribute*), 65
VALUE_ATTRS (*pandagg.aggs.RareTerms attribute*), 71
VALUE_ATTRS (*pandagg.aggs.ReverseNested attribute*), 66
VALUE_ATTRS (*pandagg.aggs.Sampler attribute*), 71
VALUE_ATTRS (*pandagg.aggs.SerialDiff attribute*), 69
VALUE_ATTRS (*pandagg.aggs.SignificantTerms attribute*), 71
VALUE_ATTRS (*pandagg.aggs.SignificantText attribute*), 72
VALUE_ATTRS (*pandagg.aggs.Stats attribute*), 66
VALUE_ATTRS (*pandagg.aggs.StatsBucket attribute*), 68
VALUE_ATTRS (*pandagg.aggs.Sum attribute*), 66
VALUE_ATTRS (*pandagg.aggs.SumBucket attribute*), 68
VALUE_ATTRS (*pandagg.aggs.Terms attribute*), 64
VALUE_ATTRS (*pandagg.aggs.TopHits attribute*), 67
VALUE_ATTRS (*pandagg.aggs.ValueCount attribute*), 67
VALUE_ATTRS (*pandagg.aggs.VariableWidthHistogram attribute*), 70
VALUE_ATTRS (*pandagg.node.aggs.abstract.ScriptPipeline attribute*), 32
VALUE_ATTRS (*pandagg.node.aggs.bucket.AdjacencyMatrix attribute*), 32
VALUE_ATTRS (*pandagg.node.aggs.bucket.AutoDateHistogram attribute*), 32
VALUE_ATTRS (*pandagg.node.aggs.bucket.Children attribute*), 33
VALUE_ATTRS (*pandagg.node.aggs.bucket.DateHistogram attribute*), 33
VALUE_ATTRS (*pandagg.node.aggs.bucket.DateRange attribute*), 33
VALUE_ATTRS (*pandagg.node.aggs.bucket.DiversifiedSampler attribute*), 33
VALUE_ATTRS (*pandagg.node.aggs.bucket.Filter attribute*), 33
VALUE_ATTRS (*pandagg.node.aggs.bucket.Filters attribute*), 33
VALUE_ATTRS (*pandagg.node.aggs.bucket.GeoDistance attribute*), 34
VALUE_ATTRS (*pandagg.node.aggs.bucket.GeoHashGrid attribute*), 34
VALUE_ATTRS (*pandagg.node.aggs.bucket.GeoTileGrid attribute*), 34
VALUE_ATTRS (*pandagg.node.aggs.bucket.Global attribute*), 34
VALUE_ATTRS (*pandagg.node.aggs.bucket.Histogram attribute*), 34
VALUE_ATTRS (*pandagg.node.aggs.bucket.IPRange attribute*), 34
VALUE_ATTRS (*pandagg.node.aggs.bucket.Missing attribute*), 35
VALUE_ATTRS (*pandagg.node.aggs.bucket.MultiTerms attribute*), 35
VALUE_ATTRS (*pandagg.node.aggs.bucket.Nested attribute*), 35
VALUE_ATTRS (*pandagg.node.aggs.bucket.Parent attribute*), 35
VALUE_ATTRS (*pandagg.node.aggs.bucket.Range at-*

tribute), 35
VALUE_ATTRS (*pandagg.node.aggs.bucket.RareTerms attribute*), 35
VALUE_ATTRS (*pandagg.node.aggs.bucket.ReverseNested attribute*), 35
VALUE_ATTRS (*pandagg.node.aggs.bucket.Sampler attribute*), 35
VALUE_ATTRS (*pandagg.node.aggs.bucket.SignificantTerms attribute*), 36
VALUE_ATTRS (*pandagg.node.aggs.bucket.SignificantText attribute*), 36
VALUE_ATTRS (*pandagg.node.aggs.bucket.Terms attribute*), 36
VALUE_ATTRS (*pandagg.node.aggs.bucket.VariableWidthHistogram attribute*), 36
VALUE_ATTRS (*pandagg.node.aggs.composite.Composite attribute*), 36
VALUE_ATTRS (*pandagg.node.aggs.metric.Avg attribute*), 37
VALUE_ATTRS (*pandagg.node.aggs.metric.Cardinality attribute*), 37
VALUE_ATTRS (*pandagg.node.aggs.metric.ExtendedStats attribute*), 37
VALUE_ATTRS (*pandagg.node.aggs.metric.GeoBound attribute*), 37
VALUE_ATTRS (*pandagg.node.aggs.metric.GeoCentroid attribute*), 37
VALUE_ATTRS (*pandagg.node.aggs.metric.Max attribute*), 37
VALUE_ATTRS (*pandagg.node.aggs.metric.Min attribute*), 38
VALUE_ATTRS (*pandagg.node.aggs.metric.PercentileRanks attribute*), 38
VALUE_ATTRS (*pandagg.node.aggs.metric.Percentiles attribute*), 38
VALUE_ATTRS (*pandagg.node.aggs.metric.Stats attribute*), 38
VALUE_ATTRS (*pandagg.node.aggs.metric.Sum attribute*), 38
VALUE_ATTRS (*pandagg.node.aggs.metric.TopHits attribute*), 38
VALUE_ATTRS (*pandagg.node.aggs.metric.ValueCount attribute*), 38
VALUE_ATTRS (*pandagg.node.aggs.pipeline.AvgBucket attribute*), 39
VALUE_ATTRS (*pandagg.node.aggs.pipeline.BucketScript attribute*), 39
VALUE_ATTRS (*pandagg.node.aggs.pipeline.BucketSelector attribute*), 39
VALUE_ATTRS (*pandagg.node.aggs.pipeline.BucketSort attribute*), 39
VALUE_ATTRS (*pandagg.node.aggs.pipeline.CumulativeSum attribute*), 39
VALUE_ATTRS (*pandagg.node.aggs.pipeline.Derivative attribute*), 40
VALUE_ATTRS (*pandagg.node.aggs.pipeline.ExtendedStatsBucket attribute*), 40
VALUE_ATTRS (*pandagg.node.aggs.pipeline.MaxBucket attribute*), 40
VALUE_ATTRS (*pandagg.node.aggs.pipeline.MinBucket attribute*), 40
VALUE_ATTRS (*pandagg.node.aggs.pipeline.MovingAvg attribute*), 40
VALUE_ATTRS (*pandagg.node.aggs.pipeline.PercentilesBucket attribute*), 40
VALUE_ATTRS (*pandagg.node.aggs.pipeline.SerialDiff attribute*), 41
VALUE_ATTRS (*pandagg.node.aggs.pipeline.StatsBucket attribute*), 41
VALUE_ATTRS (*pandagg.node.aggs.pipeline.SumBucket attribute*), 41
ValueCount (*class in pandagg.aggs*), 67
ValueCount (*class in pandagg.node.aggs.metric*), 38
VariableWidthHistogram (*class in pandagg.aggs*), 70
VariableWidthHistogram (*class in pandagg.node.aggs.bucket*), 36
VersionIncompatibilityError, 73

W

WHITELISTED_MAPPING_TYPES (*pandagg.aggs.Avg attribute*), 66
WHITELISTED_MAPPING_TYPES (*pandagg.aggs.DateHistogram attribute*), 65
WHITELISTED_MAPPING_TYPES (*pandagg.aggs.ExtendedStats attribute*), 67
WHITELISTED_MAPPING_TYPES (*pandagg.aggs.GeoBound attribute*), 67
WHITELISTED_MAPPING_TYPES (*pandagg.aggs.GeoCentroid attribute*), 67
WHITELISTED_MAPPING_TYPES (*pandagg.aggs.GeoDistance attribute*), 70
WHITELISTED_MAPPING_TYPES (*pandagg.aggs.GeoHashGrid attribute*), 70
WHITELISTED_MAPPING_TYPES (*pandagg.aggs.GeoTileGrid attribute*), 71
WHITELISTED_MAPPING_TYPES (*pandagg.aggs.Histogram attribute*), 65
WHITELISTED_MAPPING_TYPES (*pandagg.aggs.IPRange attribute*), 71
WHITELISTED_MAPPING_TYPES (*pandagg.aggs.Max attribute*), 66
WHITELISTED_MAPPING_TYPES (*pandagg.aggs.Min attribute*), 66

WHITELISTED_MAPPING_TYPES (*pandagg.aggs.Nested attribute*), 65
WHITELISTED_MAPPING_TYPES (*pandagg.aggs.PercentileRanks attribute*), 67
WHITELISTED_MAPPING_TYPES (*pandagg.aggs.Percentiles attribute*), 67
WHITELISTED_MAPPING_TYPES (*pandagg.aggs.Range attribute*), 65
WHITELISTED_MAPPING_TYPES (*pandagg.aggs.ReverseNested attribute*), 66
WHITELISTED_MAPPING_TYPES (*pandagg.aggs.SignificantText attribute*), 72
WHITELISTED_MAPPING_TYPES (*pandagg.aggs.Stats attribute*), 66
WHITELISTED_MAPPING_TYPES (*pandagg.aggs.Sum attribute*), 66
WHITELISTED_MAPPING_TYPES (*pandagg.node.aggs.bucket.DateHistogram attribute*), 33
WHITELISTED_MAPPING_TYPES (*pandagg.node.aggs.bucket.DateRange attribute*), 33
WHITELISTED_MAPPING_TYPES (*pandagg.node.aggs.bucket.GeoDistance attribute*), 34
WHITELISTED_MAPPING_TYPES (*pandagg.node.aggs.bucket.GeoHashGrid attribute*), 34
WHITELISTED_MAPPING_TYPES (*pandagg.node.aggs.bucket.GeoTileGrid attribute*), 34
WHITELISTED_MAPPING_TYPES (*pandagg.node.aggs.bucket.Histogram attribute*), 34
WHITELISTED_MAPPING_TYPES (*pandagg.node.aggs.bucket.IPRange attribute*), 34
WHITELISTED_MAPPING_TYPES (*pandagg.node.aggs.bucket.Nested attribute*), 35
WHITELISTED_MAPPING_TYPES (*pandagg.node.aggs.bucket.Range attribute*), 35
WHITELISTED_MAPPING_TYPES (*pandagg.node.aggs.bucket.ReverseNested attribute*), 35
WHITELISTED_MAPPING_TYPES (*pandagg.node.aggs.bucket.SignificantText attribute*), 36
WHITELISTED_MAPPING_TYPES (*pandagg.node.aggs.metric.Avg attribute*), 37
WHITELISTED_MAPPING_TYPES (*pandagg.node.aggs.metric.ExtendedStats attribute*), 37
WHITELISTED_MAPPING_TYPES (*pandagg.node.aggs.metric.GeoBound attribute*), 37
WHITELISTED_MAPPING_TYPES (*pandagg.node.aggs.metric.GeoCentroid attribute*), 37
WHITELISTED_MAPPING_TYPES (*pandagg.node.aggs.metric.Max attribute*), 37
WHITELISTED_MAPPING_TYPES (*pandagg.node.aggs.metric.Min attribute*), 38
WHITELISTED_MAPPING_TYPES (*pandagg.node.aggs.metric.PercentileRanks attribute*), 38
WHITELISTED_MAPPING_TYPES (*pandagg.node.aggs.metric.Percentiles attribute*), 38
WHITELISTED_MAPPING_TYPES (*pandagg.node.aggs.metric.Stats attribute*), 38
WHITELISTED_MAPPING_TYPES (*pandagg.node.aggs.metric.Sum attribute*), 38
WildCard (*class in pandagg.mappings*), 74
WildCard (*class in pandagg.node.mappings.field_datatypes*), 46
Wildcard (*class in pandagg.node.query.term_level*), 53
Wildcard (*class in pandagg.query*), 83
Wrapper (*class in pandagg.node.query.specialized*), 52
Wrapper (*class in pandagg.query*), 85