PyES Documentation

Release 0.99.5 (stable)

Elastic Search

Sep 27, 2017

Contents

Contents:

CHAPTER 1

Getting Started

Release 0.99.5

Date Sep 27, 2017

Installation

You can install PyES either via the Python Package Index (PyPI) or from source.

To install using *pip*,

```
$ pip install pyes
```

To install using *easy_install*,

```
$ easy_install pyes
```

Downloading and installing from source

Download the latest version of PyES from http://pypi.python.org/pypi/pyes/

You can install it by doing the following,

```
$ tar xvfz pyes-0.0.0.tar.gz
$ cd pyes-0.0.0
$ python setup.py build
# python setup.py install # as root
```

Using the development version

You can clone the repository by doing the following

```
$ git clone git://github.com/aparo/pyes.git
$ cd pyes
$ python setup.py develop
```

To update:

```
$ cd pyes
$ git pull origin master
```

Usage

Creating a connection. (See more details here *Connections*)

```
>>> from pyes import *
>>> conn = ES('127.0.0.1:9200') # Use HTTP
```

Deleting an index:

```
>>> try:
>>>     conn.indices.delete_index("test-index")
>>> except:
>>>     pass
```

(an exception is raised if the index is not present)

Create an index:

```
>>> conn.indices.create_index("test-index")
```

Creating a mapping via dictionary:

```
>>> mapping = {
>>>     'parsedtext': {
>>>         'boost': 1.0,
>>>         'index': 'analyzed',
>>>         'store': 'yes',
>>>         'type': 'string',
>>>         "term_vector": "with_positions_offsets"
>>>     },
>>>     'name': {
>>>         'boost': 1.0,
>>>         'index': 'analyzed',
>>>         'store': 'yes',
>>>         'type': 'string',
>>>         "term_vector": "with_positions_offsets"
>>>     },
>>>     'title': {
>>>         'boost': 1.0,
>>>         'index': 'analyzed',
>>>         'store': 'yes',
>>>         'type': 'string',
>>>         "term_vector": "with_positions_offsets"
>>>     },
>>>     'pos': {
>>>         'store': 'yes',
>>>         'type': 'integer'
```

```
>>>     },
>>>     'uuid': {
>>>         'boost': 1.0,
>>>         'index': 'not_analyzed',
>>>         'store': 'yes',
>>>         'type': 'string'
>>>     }
>>> }
>>> conn.indices.put_mapping("test-type", {'properties':mapping}, ["test-index"])
```

Creating a mapping via objects:

```
>>> from pyes.mappings import *
>>> docmapping = DocumentObjectField(name=self.document_type)
>>> docmapping.add_property(
>>>     StringField(name="parsedtext", store=True, term_vector="with_positions_offsets
↪", index="analyzed"))
>>> docmapping.add_property(
>>>     StringField(name="name", store=True, term_vector="with_positions_offsets",␣
↪index="analyzed"))
>>> docmapping.add_property(
>>>     StringField(name="title", store=True, term_vector="with_positions_offsets",␣
↪index="analyzed"))
>>> docmapping.add_property(IntegerField(name="position", store=True))
>>> docmapping.add_property(StringField(name="uuid", store=True, index="not_analyzed
↪"))
>>> nested_object = NestedObject(name="nested")
>>> nested_object.add_property(StringField(name="name", store=True))
>>> nested_object.add_property(StringField(name="value", store=True))
>>> nested_object.add_property(IntegerField(name="num", store=True))
>>> docmapping.add_property(nested_object)
>>> settings.add_mapping(docmapping)
>>> conn.ensure_index(self.index_name, settings)
```

Index some documents:

```
>>> conn.index({"name":"Joe Tester", "parsedtext":"Joe Testere nice guy", "uuid":
↪"11111", "position":1}, "test-index", "test-type", 1)
>>> conn.index({"name":"Bill Baloney", "parsedtext":"Joe Testere nice guy", "uuid":
↪"22222", "position":2}, "test-index", "test-type", 2)
```

Refreshing indexes:

```
>>> conn.indices.refresh("test-index") # Single index.
>>> conn.indices.refresh(["test-index", "test-index-2"]) # Multiple Indexes
```

Execute a query. (See *Queries*)

```
>>> q = TermQuery("name", "joe")
>>> results = conn.search(query = q)
```

results is a (See *ResultSet*), you can iterate it. It caches some results and pages them. The default returned objects are ElasticSearchModel (See *Models*).

Iterate on results:

```
>>> for r in results:
>>>    print r
```

Execute a query via queryset, via a simple ORM django like interface. (See *Queryset*)

```
>>> model = generate_model("test-index", "test-type")
>>> results = model.objects.all()
>>> results = model.objects.filter(name="joe")
```

The tests directory there are a lot of examples of functionalities.

Connections

Import the module:

```
>>> import pyes
```

pyes is able to use standard http or thrift protocol. If your port starts with "92" http protocol is used, otherwise thrift.

For a single connection (which is _not_ thread-safe), pass a list of servers.

For thrift:

```
>>> conn = pyes.ES() # Defaults to connecting to the server at '127.0.0.1:9500'
>>> conn = pyes.ES(['127.0.0.1:9500'])
>>> conn = pyes.ES(("thrift", "127.0.0.1", "9500"))
>>> conn = pyes.ES([("thrift", "127.0.0.1", "9500"), ("thrift", "192.168.1.1", "9500
↪"),])
```

For http:

```
>>> conn = pyes.ES(['127.0.0.1:9200'])
>>> conn = pyes.ES(("http", "127.0.0.1","9200"))
>>> conn = pyes.ES([("thrift", "127.0.0.1", "9200"), ("thrift", "192.168.1.1", "8000
↪"),])
```

Connections are robust to server failures. Upon a disconnection, it will attempt to connect to each server in the list in turn. If no server is available, it will raise a NoServerAvailable exception.

Timeouts are also supported and should be used in production to prevent a thread from freezing while waiting for the server to return.

```
>>> conn = pyes.ES(timeout=3.5) # 3.5 second timeout
(Make some pyes calls and the connection to the server suddenly becomes unresponsive.)

Traceback (most recent call last):
...
pyes.connection.NoServerAvailable
```

Note that this only handles socket timeouts.

Models

DotDict

The DotDict is the base model used. It allows to use a dict with the DotNotation.

```
>>> dotdict = DotDict(foo="bar")
>>> dotdict2 = deepcopy(dotdict)
>>> dotdict2["foo"] = "baz"
>>> dotdict.foo = "bar"
>>> dotdict2.foo== "baz"
    True
```

ElasticSearchModel

It extends DotDict adding methods for common uses.

Every search return an ElasticSearchModel as result. Iterating on results, you iterate on ElasticSearchModel objects.

You can create a new one with the factory or get one by search/get methods.

```
obj = self.conn.factory_object(self.index_name, self.document_type, {"name": "test",
↪"val": 1})
assert obj.name=="test"
```

You can change value via dot notation or dictionary.

```
obj.name = "aaa"
assert obj.name == "aaa"
assert obj.val == 1
```

You can change ES info via ._meta property or get_meta call.

```
assert obj._meta.id is None
obj._meta.id = "dasdas"
assert obj._meta.id == "dasdas"
```

Remember that it works as a dict object.

```
assert sorted(obj.keys()) == ["name", "val"]
```

You can save it.

```
obj.save()
obj.name = "test2"
obj.save()

reloaded = self.conn.get(self.index_name, self.document_type, obj._meta.id)
assert reloaded.name, "test2")
```

Queries

Indexing data

Before query data you should put some data in ElasticSearch.

Creating a connection:

```
>>> from pyes import *
>>> conn = ES('127.0.0.1:9200')
```

Create an index

```
>>> conn.create_index("test-index")
```

Putting some document type.

```
>>> mapping = { u'parsedtext': {'boost': 1.0,
                        'index': 'analyzed',
                        'store': 'yes',
                        'type': u'string',
                        "term_vector" : "with_positions_offsets"},
              u'name': {'boost': 1.0,
                        'index': 'analyzed',
                        'store': 'yes',
                        'type': u'string',
                        "term_vector" : "with_positions_offsets"},
              u'title': {'boost': 1.0,
                        'index': 'analyzed',
                        'store': 'yes',
                        'type': u'string',
                        "term_vector" : "with_positions_offsets"},
              u'pos': {'store': 'yes',
                        'type': u'integer'},
              u'uuid': {'boost': 1.0,
                        'index': 'not_analyzed',
                        'store': 'yes',
                        'type': u'string'}}
>>> conn.put_mapping("test-type", {'properties':mapping}, ["test-index"])
>>> conn.put_mapping("test-type2", {"_parent" : {"type" : "test-type"}}, ["test-index
↪"])
```

Index some data:

```
>>> conn.index({"name":"Joe Tester", "parsedtext":"Joe Testere nice guy", "uuid":
↪"11111", "position":1}, "test-index", "test-type", 1)
>>> conn.index({"name":"data1", "value":"value1"}, "test-index", "test-type2", 1,␣
↪parent=1)
>>> conn.index({"name":"Bill Baloney", "parsedtext":"Bill Testere nice guy", "uuid":
↪"22222", "position":2}, "test-index", "test-type", 2)
>>> conn.index({"name":"data2", "value":"value2"}, "test-index", "test-type2", 2,␣
↪parent=2)
>>> conn.index({"name":"Bill Clinton", "parsedtext":"""Bill is not
             nice guy""", "uuid":"33333", "position":3}, "test-index", "test-type", 3)
```

TIP: you can define default search indices setting the default_indices variable.

```
>>> conn.default_indices=["test-index"]
```

TIP: Remember to refresh the index before query to obtain latest insert document

```
>>> conn.refresh()
```

Querying

You can query ES with :

- a Query object or derivative

- a Search object or derivative
- a dict
- a json string

A Query wrapped in a Search it's the more safe and simple way.

Execute a query

```
>>> q = TermQuery("name", "joe")
>>> results = self.conn.search(query = q)
```

Iterate on results:

```
>>> for r in results:
>>>    print r
```

For more examples looks at the tests.

ResultSet

This object is returned as result of a query. It's lazy.

```
>>> resultset = self.conn.search(Search(MatchAllQuery(), size=20), self.index_name,␣
↪self.document_type)
```

It contains the matched and limited records. Very useful to use in pagination.

```
>>> len([p for p in resultset])
20
```

The total matched results is in the total property.

```
>>> resultset.total
1000
```

You can slice it.

```
>>> resultset = self.conn.search(Search(MatchAllQuery(), size=10), self.index_name,␣
↪self.document_type)
>>> len([p for p in resultset[:10]])
10
```

Remember all result are default ElasticSearchModel objects

```
>>> resultset[10].uuid
"11111"
```

Queryset

Creating a connection:

```
>>> from pyes.queryset import generate_model
>>> model = generate_model("myindex", "mytype")
```

Filtering:

```
>>> results = model.objects.all()
>>> len(results)
3
>>> results = model.objects.filter("name", "joe")
>>> len(results)
1
>>> results = model.objects.filter(uuid="33333")
>>> len(results)
1
>>> results = model.objects.filter(position=1).filter(position=3)
>>> len(results)
0
>>> results = model.objects.filter(position__gt=1, position__lt=3)
>>> len(results)
1
>>> results.count()
1
>>> [r.position for r in results]
[2]
>>> results = model.objects.exclude(position__in=[1, 2])
>>> len(results)
1
>>> results.count()
1
```

Retrieve an item:

```
>>> item = model.objects.get(position=1)
>>> item.position
1
>>> item = model.objects.get(position=0)
raise DoesNotExist
```

Ordering:

```
>>> items = model.objects.order_by("position")
>>> items[0].position
1
>>> items = model.objects.order_by("-position")
>>> items[0].position
3
```

Get or create:

```
>>> item, created = model.objects.get_or_create(position=1, defaults={"name": "nasty"}
↪)
>>> created
False
>>> position
1
>>> item.get_meta().id
"1"

>>> item, created = model.objects.get_or_create(position=10, defaults={"name": "nasty
↪"})
>>> created
True
```

```
>>> position
10
>>> item.name
"nasty"
```

Returns values:

```
>>> values = list(model.objects.values("uuid", "position"))
>>> len(values)
3
>>> list(values)
[{u'position': 1, u'uuid': u'11111'},{u'position': 2, u'uuid': u'22222'},{u'position
↪': 3, u'uuid': u'33333'}]
>>> values = list(model.objects.values_list("uuid", flat=True))
>>> len(values)
3
>>> list(values)
[u'11111', u'22222',u'33333']
>>> model.objects.dates("date", kind="year")
>>> len(values)
1
>>> list(values)
[datetime(2012, 1, 1, 1, 0)]


     facets = model.objects.facet("uuid").size(0).facets
     uuid_facets=facets["uuid"]
     self.assertEqual(uuid_facets["total"], 3)
     self.assertEqual(uuid_facets["terms"][0]["count"], 1)
```

Faceting counting (can be concatenated).

```
>>> facets = model.objects.facet("uuid").size(0).facets
>>> facets["uuid"]["total"]
3
>>> facets["uuid"][0]["count"]
1
```

More examples are available in test_queryset.py in tests dir.

CHAPTER 2

Frequently Asked Questions

General

What connection type should I use?

For general usage I suggest to use HTTP connection versus your server.

For more fast performance, mainly in indexing, I suggest to use thrift because its latency is lower.

How you can return a plain dict from a resultset?

ResultSet iterates on ElasticSearchModel by default, to change this behaviour you need to pass a an object that receive a connection and a dict object.

To return plain dict object, you must pass to the search call a model parameter:

```
model=lambda x,y:y
```

CHAPTER 3

API Reference

Release 0.99.5

Date Sep 27, 2017

pyes.connection

pyes.connection.**connect**(*servers=None*, *framed_transport=False*, *timeout=None*, *retry_time=60*, *recycle=None*, *round_robin=None*, *max_retries=3*)

Constructs a single ElasticSearch connection. Connects to a randomly chosen server on the list.

If the connection fails, it will attempt to connect to each server on the list in turn until one succeeds. If it is unable to find an active server, it will throw a NoServerAvailable exception.

Failing servers are kept on a separate list and eventually retried, no sooner than *retry_time* seconds after failure.

Parameters

- **servers** – [server] List of ES servers with format: "hostname:port" Default: [("127.0.0.1",9500)]
- **framed_transport** – If True, use a TFramedTransport instead of a TBufferedTransport
- **timeout** – Timeout in seconds (e.g. 0.5) Default: None (it will stall forever)
- **retry_time** – Minimum time in seconds until a failed server is reinstated. (e.g. 0.5) Default: 60
- **recycle** – Max time in seconds before an open connection is closed and returned to the pool. Default: None (Never recycle)
- **max_retries** – Max retry time on connection down
- **round_robin** – *DEPRECATED*

:return ES client

pyes.connection.**connect_thread_local**(*servers=None*, *framed_transport=False*, *timeout=None*, *retry_time=60*, *recycle=None*, *round_robin=None*, *max_retries=3*)

Constructs a single ElasticSearch connection. Connects to a randomly chosen server on the list.

If the connection fails, it will attempt to connect to each server on the list in turn until one succeeds. If it is unable to find an active server, it will throw a NoServerAvailable exception.

Failing servers are kept on a separate list and eventually retried, no sooner than *retry_time* seconds after failure.

Parameters

- **servers** – [server] List of ES servers with format: "hostname:port" Default: [("127.0.0.1",9500)]
- **framed_transport** – If True, use a TFramedTransport instead of a TBufferedTransport
- **timeout** – Timeout in seconds (e.g. 0.5) Default: None (it will stall forever)
- **retry_time** – Minimum time in seconds until a failed server is reinstated. (e.g. 0.5) Default: 60
- **recycle** – Max time in seconds before an open connection is closed and returned to the pool. Default: None (Never recycle)
- **max_retries** – Max retry time on connection down
- **round_robin** – *DEPRECATED*

:return ES client

exception pyes.connection.**NoServerAvailable**

pyes.connection_http

pyes.connection_http.**connect**

alias of Connection

pyes.convert_errors

Routines for converting error responses to appropriate exceptions.

pyes.convert_errors.**raise_if_error**(*status*, *result*, *request=None*)

Raise an appropriate exception if the result is an error.

Any result with a status code of 400 or higher is considered an error.

The exception raised will either be an ElasticSearchException, or a more specific subclass of ElasticSearchException if the type is recognised.

The status code and result can be retrieved from the exception by accessing its status and result properties.

Optionally, this can take the original RestRequest instance which generated this error, which will then get included in the exception.

pyes.decorators

pyes.decorators.**deprecated**(*description=None*, *deprecation=None*, *removal=None*, *alternative=None*)

pyes.decorators.**warn_deprecated**(*description=None*, *deprecation=None*, *removal=None*, *alternative=None*)

pyes.es

class pyes.es.**ES**(*server='localhost:9200'*, *timeout=30.0*, *bulk_size=400*, *encoder=None*, *decoder=None*, *max_retries=3*, *retry_time=60*, *default_indices=None*, *default_types=None*, *log_curl=False*, *dump_curl=False*, *model=<class 'pyes.models.ElasticSearchModel'>*, *basic_auth=None*, *raise_on_bulk_item_failure=False*, *document_object_field=None*, *bulker_class=<class 'pyes.models.ListBulker'>*, *cert_reqs='CERT_OPTIONAL'*)

ES connection object.

bulk_size

Get the current bulk_size

Return a int the size of the bulk holder

collect_info()

Collect info about the connection and fill the info dictionary.

count(*query=None*, *indices=None*, *doc_types=None*, ***query_params*)

Execute a query against one or more indices and get hits count.

create_bulker()

Create a bulker object and return it to allow to manage custom bulk policies

create_percolator(*index*, *name*, *query*, ***kwargs*)

Create a percolator document

Any kwargs will be added to the document as extra properties.

create_river(*river*, *river_name=None*)

Create a river

decoder

alias of *ESJsonDecoder*

default_indices

delete(*index*, *doc_type*, *id*, *bulk=False*, ***query_params*)

Delete a typed JSON document from a specific index based on its id. If bulk is True, the delete operation is put in bulk mode.

delete_by_query(*indices*, *doc_types*, *query*, ***query_params*)

Delete documents from one or more indices and one or more types based on a query.

delete_percolator(*index*, *name*)

Delete a percolator document

delete_river(*river*, *river_name=None*)

Delete a river

delete_warmer(*doc_types=None*, *indices=None*, *name=None*, *querystring_args=None*)

Retrieve warmer

Parameters

- **doc_types** – list of document types
- **warmer** – anything with `serialize` method or a dictionary
- **name** – warmer name. If not provided, all warmers for given indices will be deleted
- **querystring_args** – additional arguments passed as GET params to ES

encode_json(*serializable*)
Serialize to json a serializable object (Search, Query, Filter, etc).

encoder
alias of *ESJsonEncoder*

ensure_index(*index*, *mappings=None*, *settings=None*, *clear=False*)
Ensure if an index with mapping exists

exists(*index*, *doc_type*, *id*, ***query_params*)
Return if a document exists

factory_object(*index*, *doc_type*, *data=None*, *id=None*)
Create a stub object to be manipulated

flush_bulk(*forced=False*)
Send pending operations if forced or if the bulk threshold is exceeded.

force_bulk()
Force executing of all bulk data.

Return the bulk response

get(*index*, *doc_type*, *id*, *fields=None*, *model=None*, ***query_params*)
Get a typed JSON document from an index based on its id.

get_file(*index*, *doc_type*, *id=None*)
Return the filename and memory data stream

get_warmer(*doc_types=None*, *indices=None*, *name=None*, *querystring_args=None*)
Retrieve warmer

Parameters

- **doc_types** – list of document types
- **warmer** – anything with `serialize` method or a dictionary
- **name** – warmer name. If not provided, all warmers will be returned
- **querystring_args** – additional arguments passed as GET params to ES

index(*doc*, *index*, *doc_type*, *id=None*, *parent=None*, *force_insert=False*, *op_type=None*, *bulk=False*, *version=None*, *querystring_args=None*, *ttl=None*)
Index a typed JSON document into a specific index and make it searchable.

index_raw_bulk(*header*, *document*)
Function helper for fast inserting

Parameters

- **header** – a string with the bulk header must be ended with a newline
- **header** – a json document string must be ended with a newline

mappings

mget(*ids*, *index=None*, *doc_type=None*, ***query_params*)
Get multi JSON documents.

ids can be: list of tuple: (index, type, id) list of ids: index and doc_type are required

morelikethis(*index*, *doc_type*, *id*, *fields*, ***query_params*)
Execute a "more like this" search query against one or more fields and get back search hits.

partial_update(*index*, *doc_type*, *id*, *script*, *params=None*, *upsert=None*, *querystring_args=None*)
Partially update a document with a script

percolate(*index*, *doc_types*, *query*)
Match a query with a document

put_file(*filename*, *index*, *doc_type*, *id=None*, *name=None*)
Store a file in a index

put_warmer(*doc_types=None*, *indices=None*, *name=None*, *warmer=None*, *querystring_args=None*)
Put new warmer into index (or type)

Parameters

- **doc_types** – list of document types
- **warmer** – anything with `serialize` method or a dictionary
- **name** – warmer name
- **querystring_args** – additional arguments passed as GET params to ES

raise_on_bulk_item_failure
Get the raise_on_bulk_item_failure status

Return a bool the status of raise_on_bulk_item_failure

search(*query*, *indices=None*, *doc_types=None*, *model=None*, *scan=False*, *headers=None*, ***query_params*)
Execute a search against one or more indices to get the resultset.

query must be a Search object, a Query object, or a custom dictionary of search parameters using the query DSL to be passed directly.

search_multi(*queries*, *indices_list=None*, *doc_types_list=None*, *routing_list=None*, *search_type_list=None*, *models=None*, *scans=None*)

search_raw(*query*, *indices=None*, *doc_types=None*, *headers=None*, ***query_params*)
Execute a search against one or more indices to get the search hits.

query must be a Search object, a Query object, or a custom dictionary of search parameters using the query DSL to be passed directly.

search_raw_multi(*queries*, *indices_list=None*, *doc_types_list=None*, *routing_list=None*, *search_type_list=None*)

search_scroll(*scroll_id*, *scroll='10m'*)
Executes a scrolling given an scroll_id

suggest(*name*, *text*, *field*, *type='term'*, *size=None*, *params=None*, ***kwargs*)
Execute suggester of given type.

Parameters

- **name** – name for the suggester
- **text** – text to search for
- **field** – field to search
- **type** – One of: completion, phrase, term

- **size** – number of results
- **params** – additional suggester params
- **kwargs** –

Returns

suggest_from_object(*suggest*, *indices=None*, *preference=None*, *routing=None*, *raw=False*, ***kwargs*)

update(*index*, *doc_type*, *id*, *script=None*, *lang='mvel'*, *params=None*, *document=None*, *upsert=None*, *model=None*, *bulk=False*, *querystring_args=None*, *retry_on_conflict=None*, *routing=None*, *doc_as_upsert=None*)

update_by_function(*extra_doc*, *index*, *doc_type*, *id*, *querystring_args=None*, *update_func=None*, *attempts=2*)

Update an already indexed typed JSON document.

The update happens client-side, i.e. the current document is retrieved, updated locally and finally pushed to the server. This may repeat up to `attempts` times in case of version conflicts.

Parameters

- **update_func** – A callable `update_func(current_doc, extra_doc)` that computes and returns the updated doc. Alternatively it may update `current_doc` in place and return None. The default `update_func` is `dict.update`.
- **attempts** – How many times to retry in case of version conflict.

update_mapping_meta(*doc_type*, *values*, *indices=None*)

Update mapping meta :param doc_type: a doc type or a list of doctypes :param values: the dict of meta :param indices: a list of indices :return:

validate_types(*types=None*)

Return a valid list of types.

types may be a string or a list of strings. If *types* is not supplied, returns the default_types.

class pyes.es.**ESJsonDecoder**(**args*, ***kwargs*)

dict_to_object(*d*)

Decode datetime value from string to datetime

string_to_datetime(*obj*)

Decode a datetime string to a datetime object

class pyes.es.**ESJsonEncoder**(*skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *encoding='utf-8'*, *default=None*)

default(*value*)

Convert rogue and mysterious data types. Conversion notes:

•`datetime.date` and `datetime.datetime` objects are

converted into datetime strings.

class pyes.es.**EmptyResultSet**(**args*, ***kwargs*)

aggs

count()

facets

total

class pyes.es.**ResultSet**(*connection*, *search*, *indices=None*, *doc_types=None*, *query_params=None*, *headers=None*, *auto_fix_keys=False*, *auto_clean_highlight=False*, *model=None*)

aggs

clean_highlight()
Remove the empty highlight

count()

facets

fix_aggs()
This function convert date_histogram aggs to datetime

fix_facets()
This function convert date_histogram facets to datetime

fix_keys()
Remove the _ from the keys of the results

get_suggested_texts()

max_score

next()

total

class pyes.es.**ResultSetList**(*items*, *model=None*)

count()

facets

next()

total

class pyes.es.**ResultSetMulti**(*connection*, *searches*, *indices_list=None*, *doc_types_list=None*, *routing_list=None*, *search_type_list=None*, *models=None*)

next()

pyes.es.**expand_suggest_text**(*suggest*)

pyes.es.**file_to_attachment**(*filename*, *filehandler=None*)
Convert a file to attachment

pyes.es.**get_id**(*text*)

pyes.exceptions

exception pyes.exceptions.**ESPendingDeprecationWarning**

exception pyes.exceptions.**ESDeprecationWarning**

exception pyes.exceptions.**NoServerAvailable**

exception pyes.exceptions.**QueryError**

exception pyes.exceptions.**NotFoundException**(*error*, *status=None*, *result=None*, *request=None*)

exception pyes.exceptions.**AlreadyExistsException**(*error*, *status=None*, *result=None*, *request=None*)

exception pyes.exceptions.**IndexAlreadyExistsException**(*error*, *status=None*, *result=None*, *request=None*)

exception pyes.exceptions.**IndexMissingException**(*error*, *status=None*, *result=None*, *request=None*)

exception pyes.exceptions.**SearchPhaseExecutionException**(*error*, *status=None*, *result=None*, *request=None*)

exception pyes.exceptions.**InvalidIndexNameException**(*error*, *status=None*, *result=None*, *request=None*)

exception pyes.exceptions.**InvalidSortOrder**

exception pyes.exceptions.**InvalidQuery**

exception pyes.exceptions.**InvalidParameterQuery**

exception pyes.exceptions.**InvalidParameter**

exception pyes.exceptions.**QueryParameterError**

exception pyes.exceptions.**ScriptFieldsError**

exception pyes.exceptions.**ReplicationShardOperationFailedException**(*error*, *status=None*, *result=None*, *request=None*)

exception pyes.exceptions.**ClusterBlockException**(*error*, *status=None*, *result=None*, *request=None*)

exception pyes.exceptions.**MapperParsingException**(*error*, *status=None*, *result=None*, *request=None*)

exception pyes.exceptions.**ElasticSearchException**(*error*, *status=None*, *result=None*, *request=None*)

Base class of exceptions raised as a result of parsing an error return from ElasticSearch.

An exception of this class will be raised if no more specific subclass is appropriate.

exception pyes.exceptions.**ReduceSearchPhaseException**(*error*, *status=None*, *result=None*, *request=None*)

exception pyes.exceptions.**VersionConflictEngineException**(*error*, *status=None*, *result=None*, *request=None*)

exception pyes.exceptions.**DocumentAlreadyExistsEngineException**(*error*, *status=None*, *result=None*, *request=None*)

exception pyes.exceptions.**DocumentAlreadyExistsException**(*error*, *status=None*, *result=None*, *request=None*)

exception pyes.exceptions.**TypeMissingException**(*error*, *status=None*, *result=None*, *request=None*)

exception pyes.exceptions.**BulkOperationException**(*errors*, *bulk_result*)

exception pyes.exceptions.**DocumentMissingException**(*error*, *status=None*, *result=None*, *request=None*)

pyes.facets

class pyes.facets.**DateHistogramFacet**(*name*, *field=None*, *interval=None*, *time_zone=None*, *pre_zone=None*, *post_zone=None*, *factor=None*, *pre_offset=None*, *post_offset=None*, *key_field=None*, *value_field=None*, *value_script=None*, *params=None*, ***kwargs*)

class pyes.facets.**Facet**(*name*, *scope=None*, *nested=None*, *is_global=None*, *facet_filter=None*, ***kwargs*)

serialize()

class pyes.facets.**FacetFactory**

add(*facet*)
Add a term factory

add_date_facet(**args*, ***kwargs*)
Add a date factory facet

add_geo_facet(**args*, ***kwargs*)
Add a geo factory facet

add_term_facet(**args*, ***kwargs*)
Add a term factory facet

reset()
Reset the facets

serialize()

class pyes.facets.**FacetQueryWrap**(*wrap_object*, ***kwargs*)

serialize()

class pyes.facets.**FilterFacet**(*name*, *filter*, ***kwargs*)

class pyes.facets.**GeoDistanceFacet**(*name*, *field*, *pin*, *ranges=None*, *value_field=None*, *value_script=None*, *distance_unit=None*, *distance_type=None*, *params=None*, ***kwargs*)

class pyes.facets.**HistogramFacet**(*name*, *field=None*, *interval=None*, *time_interval=None*, *key_field=None*, *value_field=None*, *key_script=None*, *value_script=None*, *params=None*, ***kwargs*)

class pyes.facets.**QueryFacet**(*name*, *query*, ***kwargs*)

class pyes.facets.**RangeFacet**(*name*, *field=None*, *ranges=None*, *key_field=None*, *value_field=None*, *key_script=None*, *value_script=None*, *params=None*, ***kwargs*)

class pyes.facets.**StatisticalFacet**(*name*, *field=None*, *script=None*, *params=None*, ***kwargs*)

class pyes.facets.**TermFacet**(*field=None*, *fields=None*, *name=None*, *size=10*, *order=None*, *exclude=None*, *regex=None*, *regex_flags='DOTALL'*, *script=None*, *lang=None*, *all_terms=None*, ***kwargs*)

class pyes.facets.**TermStatsFacet**(*name*, *size=10*, *order=None*, *key_field=None*, *value_field=None*, *key_script=None*, *value_script=None*, *params=None*, ***kwargs*)

pyes.fakettypes

class pyes.fakettypes.**Method**

DELETE = 3

GET = 0

HEAD = 4

OPTIONS = 5

POST = 2

PUT = 1

class pyes.fakettypes.**RestRequest**(*method=None*, *uri=None*, *parameters=None*, *headers=None*, *body=None*)

Attributes:

- method
- uri
- parameters
- headers
- body

class pyes.fakettypes.**RestResponse**(*status=None*, *headers=None*, *body=None*)

Attributes:

- status
- headers
- body

class pyes.fakettypes.**Status**

ACCEPTED = 202

BAD_GATEWAY = 502

BAD_REQUEST = 400

CONFLICT = 409

CONTINUE = 100

CREATED = 201

EXPECTATION_FAILED = 417

FAILED_DEPENDENCY = 424

FORBIDDEN = 403

FOUND = 302

GATEWAY_TIMEOUT = 504

GONE = 410

INSUFFICIENT_STORAGE = 506

INTERNAL_SERVER_ERROR = 500

LENGTH_REQUIRED = 411

LOCKED = 423

METHOD_NOT_ALLOWED = 405

MOVED_PERMANENTLY = 301

MULTIPLE_CHOICES = 300

MULTI_STATUS = 207

NON_AUTHORITATIVE_INFORMATION = 203

NOT_ACCEPTABLE = 406

NOT_FOUND = 404

NOT_IMPLEMENTED = 501

NOT_MODIFIED = 304

NO_CONTENT = 204

OK = 200

PARTIAL_CONTENT = 206

PAYMENT_REQUIRED = 402

PRECONDITION_FAILED = 412

PROXY_AUTHENTICATION = 407

REQUESTED_RANGE_NOT_SATISFIED = 416

REQUEST_ENTITY_TOO_LARGE = 413

REQUEST_TIMEOUT = 408

REQUEST_URI_TOO_LONG = 414

RESET_CONTENT = 205

SEE_OTHER = 303

SERVICE_UNAVAILABLE = 503

SWITCHING_PROTOCOLS = 101

TEMPORARY_REDIRECT = 307

UNAUTHORIZED = 401

UNPROCESSABLE_ENTITY = 422

UNSUPPORTED_MEDIA_TYPE = 415

USE_PROXY = 305

pyes.filters

class pyes.filters.**ANDFilter**(*filters*, ***kwargs*)
A filter that matches combinations of other filters using the AND operator

Example:

t1 = TermFilter('name', 'john') t2 = TermFilter('name', 'smith') f = ANDFilter([t1, t2]) q = FilteredQuery(MatchAllQuery(), f) results = conn.search(q)

class pyes.filters.**BoolFilter**(*must=None*, *must_not=None*, *should=None*, *minimum_number_should_match=None*, ***kwargs*)
A filter that matches documents matching boolean combinations of other queries. Similar in concept to Boolean query, except that the clauses are other filters. Can be placed within queries that accept a filter.

add_must(*queries*)

add_must_not(*queries*)

add_should(*queries*)

is_empty()

class pyes.filters.**ExistsFilter**(*field*, ***kwargs*)

class pyes.filters.**Filter**(***kwargs*)

serialize()

class pyes.filters.**FilterList**(*filters*, ***kwargs*)

class pyes.filters.**GeoBoundingBoxFilter**(*field*, *location_tl*, *location_br*, ***kwargs*)
http://github.com/elasticsearch/elasticsearch/issues/290

class pyes.filters.**GeoDistanceFilter**(*field*, *location*, *distance*, *distance_type='arc'*, *distance_unit=None*, *optimize_bbox='memory'*, ***kwargs*)
http://github.com/elasticsearch/elasticsearch/issues/279

class pyes.filters.**GeoIndexedShapeFilter**(*field=None*, *id=None*, *type=None*, *index=None*, *path=None*, ***kwargs*)
http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/query-dsl-geo-shape-filter.html#_pre_indexed_shape

class pyes.filters.**GeoPolygonFilter**(*field*, *points*, ***kwargs*)
http://github.com/elasticsearch/elasticsearch/issues/294

class pyes.filters.**GeoShapeFilter**(*field=None*, *coordinates=None*, *type=None*, ***kwargs*)
http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/query-dsl-geo-shape-filter.html

class pyes.filters.**HasChildFilter**(*type*, *query*, *_scope=None*, ***kwargs*)
The has_child filter accepts a query and the child type to run against, and results in parent documents that have child docs matching the query

class pyes.filters.**HasFilter**(*type*, *query*, *_scope=None*, ***kwargs*)

class pyes.filters.**HasParentFilter**(*type*, *query*, *_scope=None*, ***kwargs*)
The has_parent filter accepts a query and the parent type to run against, and results in child documents that have parent docs matching the query

class pyes.filters.**IdsFilter**(*values*, *type=None*, ***kwargs*)

class pyes.filters.**LimitFilter**(*value=100*, ***kwargs*)

class pyes.filters.**MatchAllFilter**(***kwargs*)
A filter that matches on all documents

class pyes.filters.**MissingFilter**(*field*, *existence=None*, *null_value=None*, ***kwargs*)

class pyes.filters.**NestedFilter**(*path*, *filter*, *join=None*, ***kwargs*)
A nested filter, works in a similar fashion to the nested query, except used as a filter. It follows exactly the same structure, but also allows to cache the results (set _cache to true), and have it named (set the _name value).

class pyes.filters.**NotFilter**(*filter*, ***kwargs*)

pyes.filters.**NumericRangeFilter**
alias of *RangeFilter*

class pyes.filters.**ORFilter**(*filters*, ***kwargs*)
A filter that matches combinations of other filters using the OR operator

Example:

t1 = TermFilter('name', 'john') t2 = TermFilter('name', 'smith') f = ORFilter([t1, t2]) q = FilteredQuery(MatchAllQuery(), f) results = conn.search(q)

class pyes.filters.**PrefixFilter**(*field=None*, *prefix=None*, ***kwargs*)

add(*field*, *prefix*)

class pyes.filters.**QueryFilter**(*query*, ***kwargs*)

class pyes.filters.**RangeFilter**(*qrange=None*, *execution=None*, ***kwargs*)

add(*qrange*)

negate()
Negate some ranges: useful to resolve a NotFilter(RangeFilter(**))

class pyes.filters.**RawFilter**(*filter_text_or_dict*, ***kwargs*)
Uses exactly the filter provided as an ES filter.

serialize()

class pyes.filters.**RegexTermFilter**(*field=None*, *value=None*, *ignorecase=False*, ***kwargs*)

add(*field*, *value*, *ignorecase=False*)

class pyes.filters.**ScriptFilter**(*script*, *params=None*, *lang=None*, ***kwargs*)

add(*field*, *value*)

class pyes.filters.**TermFilter**(*field=None*, *value=None*, ***kwargs*)

add(*field*, *value*)

class pyes.filters.**TermsFilter**(*field=None*, *values=None*, *execution=None*, ***kwargs*)
If you want to use the Terms lookup feature, you can do it like that:

from pyes.utils import TermsLookup

Example:

tl = TermsLookup(index='index', type='type', id='id', path='path') f = TermsFilter('key', tl)

q = FilteredQuery(MatchAllQuery(), f) results = conn.search(q)

add(*field*, *values*)

class pyes.filters.**TypeFilter**(*type*, ***kwargs*)

pyes.helpers

class pyes.helpers.**SettingsBuilder**(*settings=None*, *mappings=None*)

add_mapping(*data*, *name=None*)
Add a new mapping

as_dict()
Returns a dict

class pyes.helpers.**StatusProcessor**(*connection*)

get_indices_data()

pyes.highlight

class pyes.highlight.**HighLighter**(*pre_tags=None*, *post_tags=None*, *fields=None*, *fragment_size=None*, *number_of_fragments=None*, *fragment_offset=None*, *encoder=None*)
This object manage the highlighting

Parameters

- **pre_tags** – list of tags before the highlighted text. importance is ordered.. ex. `['<b>']`
- **post_tags** – list of end tags after the highlighted text. should line up with pre_tags. ex. `['</b>']`
- **fields** – list of fields to highlight
- **fragment_size** – the size of the grament
- **number_or_fragments** – the maximum number of fragments to return; if 0, then no fragments are returned and instead the entire field is returned and highlighted.
- **fragment_offset** – controls the margin to highlight from

Use this with a *pyes.query.Search* like this:

```
h = HighLighter(['<b>'], ['</b>'])
s = Search(TermQuery('foo'), highlight=h)
```

add_field(*name*, *fragment_size=150*, *number_of_fragments=3*, *fragment_offset=None*, *order='score'*, *type=None*)
Add a field to Highlinghter

serialize()

pyes.managers

class pyes.managers.**Cluster**(*conn*)

health(*indices=None*, *level='cluster'*, *wait_for_status=None*, *wait_for_relocating_shards=None*, *timeout=30*)
Check the current *cluster health*. Request Parameters

The cluster health API accepts the following request parameters:

Parameters

- **level** – Can be one of cluster, indices or shards. Controls the details level of the health information returned. Defaults to *cluster*.
- **wait_for_status** – One of green, yellow or red. Will wait (until the timeout provided) until the status of the cluster changes to the one provided. By default, will not wait for any status.
- **wait_for_relocating_shards** – A number controlling to how many relocating shards to wait for. Usually will be 0 to indicate to wait till all relocation have happened. Defaults to not to wait.
- **timeout** – A time based parameter controlling how long to wait if one of the wait_for_XXX are provided. Defaults to 30s.

info()
The cluster *nodes info* API allows to retrieve one or more (or all) of the cluster nodes information.

node_stats(*nodes=None*)
The cluster *nodes info* API allows to retrieve one or more (or all) of the cluster nodes information.

nodes_info(*nodes=None*)
The cluster *nodes info* API allows to retrieve one or more (or all) of the cluster nodes information.

shutdown(*all_nodes=False*, *master=False*, *local=False*, *nodes=[]*, *delay=None*)

state(*filter_nodes=None*, *filter_routing_table=None*, *filter_metadata=None*, *filter_blocks=None*, *filter_indices=None*)
Retrieve the *cluster state*.

Parameters

- **filter_nodes** – set to **true** to filter out the **nodes** part of the response.
- **filter_routing_table** – set to **true** to filter out the **routing_table** part of the response.
- **filter_metadata** – set to **true** to filter out the **metadata** part of the response.
- **filter_blocks** – set to **true** to filter out the **blocks** part of the response.
- **filter_indices** – when not filtering metadata, a comma separated list of indices to include in the response.

class pyes.managers.**Indices**(*conn*)

add_alias(*alias*, *indices*, ***kwargs*)
Add an alias to point to a set of indices. (See *Admin Indices Aliases*)

Parameters

- **alias** – the name of an alias

- **indices** – a list of indices

alias_params = ['filter', 'routing', 'search_routing', 'index_routing']

aliases(*indices=None*)
Retrieve the aliases of one or more indices. (See *Admin Indices Aliases*)

Parameters **indices** – an index or a list of indices

analyze(*text*, *index=None*, *analyzer=None*, *tokenizer=None*, *filters=None*, *field=None*)
Performs the analysis process on a text and return the tokens breakdown of the text

(See *Admin Indices Optimize*)

change_aliases(*commands*)
Change the aliases stored. (See *Admin Indices Aliases*)

Parameters **commands** – is a list of 3-tuples; (command, index, alias), where *command* is one of "add" or "remove", and *index* and *alias* are the index and alias to add or remove.

close_index(*index*)
Close an index. (See *Admin Indices Open Close*)

Parameters **index** – the name of the index

create_index(*index*, *settings=None*)
Creates an index with optional settings. *Admin Indices Create Index*

Parameters

- **index** – the name of the index
- **settings** – a settings object or a dict containing settings

create_index_if_missing(*index*, *settings=None*)
Creates an index if it doesn't already exist.

If supplied, settings must be a dictionary.

Parameters

- **index** – the name of the index
- **settings** – a settings object or a dict containing settings

delete_alias(*alias*, *indices*)
Delete an alias. (See *Admin Indices Aliases*)

The specified index or indices are deleted from the alias, if they are in it to start with. This won't report an error even if the indices aren't present in the alias.

Parameters

- **alias** – the name of an alias
- **indices** – a list of indices

delete_index(*index*)
Deletes an index. *Admin Indices Delete Index*

Parameters **index** – the name of the index

delete_index_if_exists(*index*)
Deletes an index if it exists.

Parameters **index** – the name of the index

delete_mapping(*index*, *doc_type*)
Delete a typed JSON document type from a specific index. (See *Admin Indices Delete Mapping*)

exists_index(*index*)
Check if an index exists. (See *Admin Indices Indices Exists*)

Parameters **index** – the name of the index

flush(*indices=None*, *refresh=None*)
Flushes one or more indices (clear memory) If a bulk is full, it sends it.

(See *Admin Indices Flush*)

Parameters

- **indices** – an index or a list of indices
- **refresh** – set the refresh parameter

gateway_snapshot(*indices=None*)
Gateway snapshot one or more indices (See *Admin Indices Gateway Snapshot*)

Parameters **indices** – a list of indices or None for default configured.

get_alias(*alias*)
Get the index or indices pointed to by a given alias. (See *Admin Indices Aliases*)

Parameters **alias** – the name of an alias

:return returns a list of index names. :raise IndexMissingException if the alias does not exist.

get_closed_indices()
Get all closed indices.

get_indices(*include_aliases=False*)
Get a dict holding an entry for each index which exists.

If include_alises is True, the dict will also contain entries for aliases.

The key for each entry in the dict is the index or alias name. The value is a dict holding the following properties:

•num_docs: Number of documents in the index or alias.

•alias_for: Only present for an alias: holds a list of indices which this is an alias for.

get_mapping(*doc_type=None*, *indices=None*, *raw=False*)
Register specific mapping definition for a specific type against one or more indices. (See *Admin Indices Get Mapping*)

get_settings(*index=None*)
Returns the current settings for an index. (See *Admin Indices Get Settings*)

open_index(*index*)
Open an index. (See *Admin Indices Open Close*)

Parameters **index** – the name of the index

optimize(*indices=None*, *wait_for_merge=False*, *max_num_segments=None*, *only_expunge_deletes=False*, *refresh=True*, *flush=True*)
Optimize one or more indices. (See *Admin Indices Optimize*)

Parameters

- **indices** – the list of indices to optimise. If not supplied, all default_indices are optimised.

- **wait_for_merge** – If True, the operation will not return until the merge has been completed. Defaults to False.
- **max_num_segments** – The number of segments to optimize to. To fully optimize the index, set it to 1. Defaults to half the number configured by the merge policy (which in turn defaults to 10).
- **only_expunge_deletes** – Should the optimize process only expunge segments with deletes in it. In Lucene, a document is not deleted from a segment, just marked as deleted. During a merge process of segments, a new segment is created that does have those deletes. This flag allow to only merge segments that have deletes. Defaults to false.
- **refresh** – Should a refresh be performed after the optimize. Defaults to true.
- **flush** – Should a flush be performed after the optimize. Defaults to true.

put_mapping(*doc_type=None*, *mapping=None*, *indices=None*, *ignore_conflicts=None*)
Register specific mapping definition for a specific type against one or more indices. (See *Admin Indices Put Mapping*)

refresh(*indices=None*, *timesleep=None*, *timeout=0*)
Refresh one or more indices If a bulk is full, it sends it. (See *Admin Indices Refresh*)

Parameters

- **indices** – an index or a list of indices
- **timesleep** – seconds to wait
- **timeout** – seconds to wait before timing out when waiting for the cluster's health.

set_alias(*alias*, *indices*, ***kwargs*)
Set an alias. (See *Admin Indices Aliases*)

This handles removing the old list of indices pointed to by the alias.

Warning: there is a race condition in the implementation of this function - if another client modifies the indices which this alias points to during this call, the old value of the alias may not be correctly set.

Parameters

- **alias** – the name of an alias
- **indices** – a list of indices

stats(*indices=None*)
Retrieve the statistic of one or more indices (See *Admin Indices Stats*)

Parameters **indices** – an index or a list of indices

status(*indices=None*)
Retrieve the status of one or more indices (See *Admin Indices Status*)

Parameters **indices** – an index or a list of indices

update_settings(*index*, *newvalues*)
Update Settings of an index. (See *Admin Indices Update Settings*)

pyes.mappings

class pyes.mappings.**AbstractField**(*index=True*, *store=False*, *boost=1.0*, *term_vector=False*, *term_vector_positions=False*, *term_vector_offsets=False*, *omit_norms=True*, *tokenize=True*, *omit_term_freq_and_positions=True*, *type=None*, *index_name=None*, *index_options=None*, *path=None*, *norms=None*, *analyzer=None*, *index_analyzer=None*, *search_analyzer=None*, *name=None*, *locale=None*, *fields=None*)

as_dict()

get_code(*num=0*)

class pyes.mappings.**AttachmentField**(*name*, *type=None*, *path=None*, *fields=None*)
An attachment field.

Requires the mapper-attachments plugin to be installed to be used.

as_dict()

class pyes.mappings.**BinaryField**(**args*, ***kwargs*)

as_dict()

class pyes.mappings.**BooleanField**(*null_value=None*, *include_in_all=None*, **args*, ***kwargs*)

as_dict()

class pyes.mappings.**ByteField**(**args*, ***kwargs*)

class pyes.mappings.**DateField**(*format=None*, ***kwargs*)

as_dict()

to_es(*value*)

to_python(*value*)

class pyes.mappings.**DocumentObjectField**(*_all=None*, *_boost=None*, *_id=None*, *_index=None*, *_source=None*, *_type=None*, *date_formats=None*, *_routing=None*, *_ttl=None*, *_parent=None*, *_timestamp=None*, *_analyzer=None*, *_size=None*, *date_detection=None*, *numeric_detection=None*, *dynamic_date_formats=None*, *_meta=None*, **args*, ***kwargs*)

add_property(*prop*)
Add a property to the object

as_dict()

enable_compression(*threshold='5kb'*)

get_code(*num=1*)

get_meta(*subtype=None*)
Return the meta data.

save()

class pyes.mappings.**DoubleField**(**args*, ***kwargs*)

class pyes.mappings.**FloatField**(**args*, ***kwargs*)

class pyes.mappings.**GeoPointField**(*null_value=None*, *include_in_all=None*, *lat_lon=None*, *geohash=None*, *geohash_precision=None*, *normalize_lon=None*, *normalize_lat=None*, *validate_lon=None*, *validate_lat=None*, **args*, ***kwargs*)

as_dict()

class pyes.mappings.**IntegerField**(**args*, ***kwargs*)

class pyes.mappings.**IpField**(**args*, ***kwargs*)

class pyes.mappings.**LongField**(**args*, ***kwargs*)

class pyes.mappings.**Mapper**(*data*, *connection=None*, *is_mapping=False*, *document_object_field=None*)

get_all_indices()

get_doctype(*index*, *name*)
Returns a doctype given an index and a name

get_doctypes(*index*, *edges=True*)
Returns a list of doctypes given an index

get_property(*index*, *doctype*, *name*)
Returns a property of a given type

:return a mapped property

migrate(*mapping*, *index*, *doc_type*)
Migrate a ES mapping

Parameters

- **mapping** – new mapping
- **index** – index of old mapping
- **doc_type** – type of old mapping

Returns The diff mapping

class pyes.mappings.**MultiField**(*name*, *type=None*, *path=None*, *fields=None*)

add_fields(*fields*)

as_dict()

get_diff(*other_mapping*)
Returns a Multifield with diff fields. If not changes, returns None :param other_mapping: :return: a Multifield or None

class pyes.mappings.**NestedObject**(**args*, ***kwargs*)

class pyes.mappings.**NumericFieldAbstract**(*null_value=None*, *include_in_all=None*, *precision_step=4*, *numeric_resolution=None*, *ignore_malformed=None*, ***kwargs*)

as_dict()

class pyes.mappings.**ObjectField**(*name=None*, *type=None*, *path=None*, *properties=None*, *dynamic=None*, *enabled=None*, *include_in_all=None*, *dynamic_templates=None*, *include_in_parent=None*, *include_in_root=None*, *connection=None*, *index_name=None*, **args*, ***kwargs*)

add_property(*prop*)
Add a property to the object

as_dict()

clear_properties()
Helper function to reset properties

get_available_facets()
Returns Available facets for the document

get_code(*num=1*)

get_datetime_properties(*recursive=True*)
Returns a dict of property.path and property.

:param recursive the name of the property :returns a dict

get_diff(*new_mapping*)
Given two mapping it extracts a schema evolution mapping. Returns None if no evolutions are required :param new_mapping: the new mapping :return: a new evolution mapping or None

get_properties_by_type(*type*, *recursive=True*, *parent_path=''*)
Returns a sorted list of fields that match the type.

:param type the type of the field "string","integer" or a list of types :param recursive recurse to sub object :returns a sorted list of fields the match the type

get_property_by_name(*name*)
Returns a mapped object.

:param name the name of the property :returns the mapped object or exception NotFoundMapping

save()

class pyes.mappings.**ShortField**(**args*, ***kwargs*)

class pyes.mappings.**StringField**(*null_value=None*, *include_in_all=None*, **args*, ***kwargs*)

as_dict()

pyes.mappings.**get_field**(*name*, *data*, *default='object'*, *document_object_field=None*, *is_document=False*)
Return a valid Field by given data

pyes.mappings.**to_bool**(*value*)
Convert a value to boolean :param value: the value to convert :type value: any type :return: a boolean value :rtype: a boolean

pyes.models

class pyes.models.**BaseBulker**(*conn*, *bulk_size=400*, *raise_on_bulk_item_failure=False*)
Base class to implement a bulker strategy

add(*content*)

bulk_size
Get the current bulk_size

Return a int the size of the bulk holder

flush_bulk(*forced=False*)

get_bulk_size()
Get the current bulk_size

Return a int the size of the bulk holder

set_bulk_size(*bulk_size*)
Set the bulk size

:param bulk_size the bulker size

class pyes.models.**DotDict**

class pyes.models.**ElasticSearchModel**(**args*, ***kwargs*)

delete(*bulk=False*)
Delete the object

get_bulk(*create=False*)
Return bulk code

get_id()
Force the object saveing to get an id

get_meta()

reload()

save(*bulk=False*, *id=None*, *parent=None*, *routing=None*, *force=False*)
Save the object and returns id

class pyes.models.**ListBulker**(*conn*, *bulk_size=400*, *raise_on_bulk_item_failure=False*)
A bulker that store data in a list

add(*content*)

flush_bulk(*forced=False*)

class pyes.models.**SortedDict**(*data=None*)
A dictionary that keeps its keys in the order in which they're inserted.

Taken from django

clear()

copy()
Returns a copy of this object.

insert(*index*, *key*, *value*)
Inserts the key, value pair before the item with the given index.

items()

iterkeys()

itervalues()

keys()

pop(*k*, **args*)

popitem()

setdefault(*key*, *default*)

update(*dict_*)

value_for_index(*index*)
Returns the value of the item at the given zero-based index.

values()

pyes.query

class pyes.query.**BoolQuery**(*must=None*, *must_not=None*, *should=None*, *boost=None*, *minimum_number_should_match=1*, *disable_coord=None*, ***kwargs*)
A boolean combination of other queries.

BoolQuery maps to Lucene **BooleanQuery**. It is built using one or more boolean clauses, each clause with a typed occurrence. The occurrence types are:

Occur	Description
must	The clause (query) must appear in matching documents.
should	The clause (query) should appear in the matching document. A boolean query with no **must** clauses, one or more **should** clauses must match a document. The minimum number of should clauses to match can be set using **minimum_number_should_match** parameter.
must_not	The clause (query) must not appear in the matching documents. Note that it is not possible to search on documents that only consists of a **must_not** clauses.

The bool query also supports **disable_coord** parameter (defaults to **false**).

add_must(*queries*)
Add a query to the "must" clause of the query.

The Query object will be returned, so calls to this can be chained.

add_must_not(*queries*)
Add a query to the "must_not" clause of the query.

The Query object will be returned, so calls to this can be chained.

add_should(*queries*)
Add a query to the "should" clause of the query.

The Query object will be returned, so calls to this can be chained.

is_empty()

class pyes.query.**ConstantScoreQuery**(*filter=None*, *boost=1.0*, ***kwargs*)
Returns a constant score for all documents matching a filter.

Multiple filters may be supplied by passing a sequence or iterator as the filter parameter. If multiple filters are supplied, documents must match all of them to be matched by this query.

add(*filter_or_query*)
Add a filter, or a list of filters, to the query.

If a sequence of filters is supplied, they are all added, and will be combined with an ANDFilter.

If a sequence of queries is supplied, they are all added, and will be combined with an BooleanQuery(must)
.

is_empty()
Returns True if the query is empty.

class pyes.query.**CustomScoreQuery**(*query=None*, *script=None*, *params=None*, *lang=None*, ***kwargs*)

add_param(*name*, *value*)

class pyes.query.**DisMaxQuery**(*query=None*, *tie_breaker=0.0*, *boost=1.0*, *queries=None*, ***kwargs*)

add(*query*)

class pyes.query.**FieldParameter**(*field*, *query*, *default_operator='OR'*, *analyzer=None*, *allow_leading_wildcard=True*, *lowercase_expanded_terms=True*, *enable_position_increments=True*, *fuzzy_prefix_length=0*, *fuzzy_min_sim=0.5*, *phrase_slop=0*, *boost=1.0*)

serialize()

class pyes.query.**FilterQuery**(*filters=None*, ***kwargs*)

add(*filterquery*)

class pyes.query.**FilteredQuery**(*query*, *filter*, ***kwargs*)
FilteredQuery allows for results to be filtered using the various filter classes.

Example:

t = TermFilter('name', 'john') q = FilteredQuery(MatchAllQuery(), t) results = conn.search(q)

class pyes.query.**FunctionScoreQuery**(*functions=None*, *query=None*, *filter=None*, *max_boost=None*, *boost=None*, *score_mode=None*, *boost_mode=None*, *params=None*)
The functoin_score query exists since 0.90.4. It replaces CustomScoreQuery and some other.

class BoostFunction(*boost_factor*, *filter=None*)
Boost by a factor

serialize()

class FunctionScoreQuery.**BoostModes**
Some helper object to show the possibility of boost_mode

AVG = 'avg'

MAX = 'max'

MIN = 'min'

MULTIPLY = 'multiply'

REPLACE = 'replace'

SUM = 'sum'

class FunctionScoreQuery.**DecayFunction**(*decay_function*, *field*, *origin*, *scale*, *decay=None*, *offset=None*, *filter=None*)

class FunctionScoreQuery.**FunctionScoreFunction**

serialize()
Serialize the function to a structure using the query DSL.

class FunctionScoreQuery.**RandomFunction**(*seed*, *filter=None*)
Is a random boost based on a seed value

class FunctionScoreQuery.**ScoreModes**
Some helper object to show the possibility of score_mode

AVG = 'avg'

FIRST = 'first'

MAX = 'max'

MIN = 'min'

MULTIPLY = 'multiply'

SUM = 'sum'

class FunctionScoreQuery.**ScriptScoreFunction**(*script=None*, *params=None*, *lang=None*, *filter=None*)
Scripting function with params and a script. Also possible to switch the script language

class pyes.query.**FuzzyLikeThisFieldQuery**(*field*, *like_text*, *ignore_tf=False*, *max_query_terms=25*, *boost=1.0*, *min_similarity=0.5*, ***kwargs*)

class pyes.query.**FuzzyLikeThisQuery**(*fields*, *like_text*, *ignore_tf=False*, *max_query_terms=25*, *min_similarity=0.5*, *prefix_length=0*, *boost=1.0*, ***kwargs*)

class pyes.query.**FuzzyQuery**(*field*, *value*, *boost=None*, *min_similarity=0.5*, *prefix_length=0*, ***kwargs*)
A fuzzy based query that uses similarity based on Levenshtein (edit distance) algorithm.

Note Warning: this query is not very scalable with its default prefix length of 0 - in this case, every term will be enumerated and cause an edit score calculation. Here is a simple example:

class pyes.query.**HasChildQuery**(*type*, *query*, *_scope=None*, ***kwargs*)

class pyes.query.**HasParentQuery**(*type*, *query*, *_scope=None*, ***kwargs*)

class pyes.query.**HasQuery**(*type*, *query*, *_scope=None*, ***kwargs*)

class pyes.query.**IdsQuery**(*values*, *type=None*, ***kwargs*)

class pyes.query.**MatchAllQuery**(*boost=None*, ***kwargs*)
Query used to match all

Example:

q = MatchAllQuery() results = conn.search(q)

class pyes.query.**MatchQuery**(*field*, *text*, *type='boolean'*, *slop=0*, *fuzziness=None*, *prefix_length=0*, *max_expansions=2147483647*, *operator='or'*, *analyzer=None*, *boost=1.0*, *minimum_should_match=None*, *cutoff_frequency=None*, ***kwargs*)
Replaces TextQuery

class pyes.query.**MoreLikeThisFieldQuery**(*field*, *like_text*, *percent_terms_to_match=0.3*, *min_term_freq=2*, *max_query_terms=25*, *stop_words=None*, *min_doc_freq=5*, *max_doc_freq=None*, *min_word_len=0*, *max_word_len=0*, *boost_terms=1*, *boost=1.0*, ***kwargs*)

class pyes.query.**MoreLikeThisQuery**(*fields*, *like_text*, *percent_terms_to_match=0.3*, *min_term_freq=2*, *max_query_terms=25*, *stop_words=None*, *min_doc_freq=5*, *max_doc_freq=None*, *min_word_len=0*, *max_word_len=0*, *boost_terms=1*, *boost=1.0*, ***kwargs*)

class pyes.query.**MultiMatchQuery**(*fields*, *text*, *type='boolean'*, *slop=0*, *fuzziness=None*, *prefix_length=0*, *max_expansions=2147483647*, *rewrite=None*, *operator='or'*, *analyzer=None*, *use_dis_max=True*, *minimum_should_match=None*, ***kwargs*)

A family of match queries that accept text/numerics/dates, analyzes it, and constructs a query out of it. Replaces TextQuery.

Examples:

q = MatchQuery('book_title', 'elasticsearch') results = conn.search(q)

q = MatchQuery('book_title', 'elasticsearch python', type='phrase') results = conn.search(q)

class pyes.query.**NestedQuery**(*path*, *query*, *_scope=None*, *score_mode='avg'*, ***kwargs*)

Nested query allows to query nested objects / docs (see nested mapping). The query is executed against the nested objects / docs as if they were indexed as separate docs (they are, internally) and resulting in the root parent doc (or parent nested mapping).

The query path points to the nested object path, and the query (or filter) includes the query that will run on the nested docs matching the direct path, and joining with the root parent docs.

The score_mode allows to set how inner children matching affects scoring of parent. It defaults to avg, but can be total, max and none.

Multi level nesting is automatically supported, and detected, resulting in an inner nested query to automatically match the relevant nesting level (and not root) if it exists within another nested query.

class pyes.query.**PercolatorQuery**(*doc*, *query=None*, ***kwargs*)

A percolator query is used to determine which registered PercolatorDoc's match the document supplied.

search(***kwargs*)

Disable this as it is not allowed in percolator queries.

serialize()

Serialize the query to a structure using the query DSL.

class pyes.query.**PrefixQuery**(*field=None*, *prefix=None*, *boost=None*, ***kwargs*)

add(*field*, *prefix*, *boost=None*)

class pyes.query.**Query**(**args*, ***kwargs*)

Base class for all queries.

search(***kwargs*)

Return this query wrapped in a Search object.

Any keyword arguments supplied to this call will be passed to the Search object.

serialize()

Serialize the query to a structure using the query DSL.

class pyes.query.**QueryStringQuery**(*query*, *default_field=None*, *search_fields=None*, *default_operator='OR'*, *analyzer=None*, *allow_leading_wildcard=True*, *lowercase_expanded_terms=True*, *enable_position_increments=True*, *fuzzy_prefix_length=0*, *fuzzy_min_sim=0.5*, *phrase_slop=0*, *boost=1.0*, *analyze_wildcard=False*, *use_dis_max=True*, *tie_breaker=0*, *clean_text=False*, *minimum_should_match=None*, ***kwargs*)

Query to match values on all fields for a given string

Example:

q = QueryStringQuery('elasticsearch') results = conn.search(q)

class pyes.query.**RangeQuery**(*qrange=None*, ***kwargs*)

add(*qrange*)

class pyes.query.**RegexTermQuery**(*field=None*, *value=None*, *boost=None*, ***kwargs*)

class pyes.query.**RescoreQuery**(*query*, *window_size=None*, *query_weight=None*, *rescore_query_weight=None*, ***kwargs*)

A rescore query is used to rescore top results from another query.

serialize()

Serialize the query to a structure using the query DSL.

class pyes.query.**Search**(*query=None*, *filter=None*, *fields=None*, *start=None*, *size=None*, *highlight=None*, *sort=None*, *explain=False*, *facet=None*, *agg=None*, *rescore=None*, *window_size=None*, *version=None*, *track_scores=None*, *script_fields=None*, *index_boost=None*, *min_score=None*, *stats=None*, *bulk_read=None*, *partial_fields=None*, *_source=None*, *timeout=None*)

A search to be performed.

This contains a query, and has additional parameters which are used to control how the search works, what it should return, etc.

The rescore parameter takes a Search object created from a RescoreQuery.

Example:

q = QueryStringQuery('elasticsearch') s = Search(q, fields=['title', 'author'], start=100, size=50) results = conn.search(s)

add_highlight(*field*, *fragment_size=None*, *number_of_fragments=None*, *fragment_offset=None*, *type=None*)

Add a highlight field.

The Search object will be returned, so calls to this can be chained.

add_index_boost(*index*, *boost*)

Add a boost on an index.

The Search object will be returned, so calls to this can be chained.

get_agg_factory()

Returns the agg factory

get_facet_factory()

Returns the facet factory

highlight

script_fields

serialize()
Serialize the search to a structure as passed for a search body.

class pyes.query.**SimpleQueryStringQuery**(*query*, *default_field=None*, *search_fields=None*, *default_operator='OR'*, *analyzer=None*, *allow_leading_wildcard=True*, *lowercase_expanded_terms=True*, *enable_position_increments=True*, *fuzzy_prefix_length=0*, *fuzzy_min_sim=0.5*, *phrase_slop=0*, *boost=1.0*, *analyze_wildcard=False*, *use_dis_max=True*, *tie_breaker=0*, *clean_text=False*, *minimum_should_match=None*, ***kwargs*)
A query that uses the SimpleQueryParser to parse its context. Unlike the regular query_string query, the simple_query_string query will never throw an exception, and discards invalid parts of the query.

Example:

q = SimpleQueryStringQuery('elasticsearch') results = conn.search(q)

class pyes.query.**SpanFirstQuery**(*field=None*, *value=None*, *end=3*, ***kwargs*)

class pyes.query.**SpanMultiQuery**(*query*, ***kwargs*)
This query allows you to wrap multi term queries (fuzzy, prefix, wildcard, range).

The query element is either of type WildcardQuery, FuzzyQuery, PrefixQuery or RangeQuery. A boost can also be associated with the element query

class pyes.query.**SpanNearQuery**(*clauses=None*, *slop=1*, *in_order=None*, *collect_payloads=None*, ***kwargs*)
Matches spans which are near one another. One can specify _slop_, the maximum number of intervening unmatched positions, as well as whether matches are required to be in-order.

The clauses element is a list of one or more other span type queries and the slop controls the maximum number of intervening unmatched positions permitted.

class pyes.query.**SpanNotQuery**(*include*, *exclude*, ***kwargs*)
Removes matches which overlap with another span query.

The include and exclude clauses can be any span type query. The include clause is the span query whose matches are filtered, and the exclude clause is the span query whose matches must not overlap those returned.

class pyes.query.**SpanOrQuery**(*clauses=None*, ***kwargs*)
Matches the union of its span clauses.

The clauses element is a list of one or more other span type queries.

class pyes.query.**SpanTermQuery**(*field=None*, *value=None*, *boost=None*, ***kwargs*)

class pyes.query.**Suggest**(*fields=None*)

add(*text*, *name*, *field*, *type='term'*, *size=None*, *params=None*)
Set the suggester of given type.

Parameters

- **text** – text
- **name** – name of suggest
- **field** – field to be used
- **type** – type of suggester to add, available types are: completion, phrase, term
- **size** – number of phrases

- **params** – dict of additional parameters to pass to the suggester

Returns None

add_completion(*text*, *name*, *field*, *size=None*, *params=None*)
Add completion-type suggestion:

http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/search-suggesters-completion.html

Parameters

- **text** – text for searching using autocompletion
- **name** – name for the suggester
- **field** – document's field to be autocompleted
- **size** – (optional) size of the autocomplete list

Returns

add_phrase(*text*, *name*, *field*, *size=None*, *params=None*)

add_term(*text*, *name*, *field*, *size=None*, *params=None*)

is_valid()

serialize()

class pyes.query.**TermQuery**(*field=None*, *value=None*, *boost=None*, ***kwargs*)
Match documents that have fields that contain a term (not analyzed).

A boost may be supplied.

Example:

q = TermQuery('name', 'john') results = conn.search(q)

With boost:

q = TermQuery('name', 'john', boost=0.75) results = conn.search(q)

add(*field*, *value*, *boost=None*)

class pyes.query.**TermsQuery**(**args*, ***kwargs*)

add(*field*, *value*, *minimum_match=1*, *boost=None*)

class pyes.query.**TextQuery**(*field*, *text*, *type='boolean'*, *slop=0*, *fuzziness=None*, *prefix_length=0*, *max_expansions=2147483647*, *operator='or'*, *analyzer=None*, *boost=1.0*, *minimum_should_match=None*, *cutoff_frequency=None*, ***kwargs*)
A new family of text queries that accept text, analyzes it, and constructs a query out of it.

Examples:

q = TextQuery('book_title', 'elasticsearch') results = conn.search(q)

q = TextQuery('book_title', 'elasticsearch python', operator='and') results = conn.search(q)

add_query(*field*, *text*, *type='boolean'*, *slop=0*, *fuzziness=None*, *prefix_length=0*, *max_expansions=2147483647*, *operator='or'*, *analyzer=None*, *boost=1.0*, *minimum_should_match=None*, *cutoff_frequency=None*)

class pyes.query.**TopChildrenQuery**(*type*, *score='max'*, *factor=5*, *incremental_factor=2*, ***kwargs*)

class pyes.query.**WildcardQuery**(*field=None*, *value=None*, *boost=None*, ***kwargs*)

pyes.query.**is_a_spanquery**(*obj*)
Returns if the object is a span query

pyes.queryset

The main QuerySet implementation. This provides the public API for the ORM.

Taken from django one and from django-elasticsearch.

class pyes.queryset.**ESModel**(*index*, *type*, *es_url=None*, *es_kwargs={}*)

class pyes.queryset.**QuerySet**(*model=None*, *using=None*, *index=None*, *type=None*, *es_url=None*, *es_kwargs={}*)
Represents a lazy database lookup for a set of objects.

agg(**args*, ***kwargs*)

aggregate(**args*, ***kwargs*)
Returns a dictionary containing the calculations (aggregation) over the current queryset

If args is present the expression is passed as a kwarg using the Aggregate object's default alias.

aggs

all()
Returns a new QuerySet that is a copy of the current one. This allows a QuerySet to proxy for a model manager in some cases.

annotate(**args*, ***kwargs*)
Return a query set in which the returned objects have been annotated with data aggregated from related fields.

bulk_create(*objs*, *batch_size=None*)
Inserts each of the instances into the database. This does *not* call save() on each of the instances, does not send any pre/post save signals, and does not set the primary key attribute if it is an autoincrement field.

complex_filter(*filter_obj*)
Returns a new QuerySet instance with filter_obj added to the filters.

filter_obj can be a Q object (or anything with an add_to_query() method) or a dictionary of keyword lookup arguments.

This exists to support framework features such as 'limit_choices_to', and usually it will be more natural to use other methods.

count()
Performs a SELECT COUNT() and returns the number of records as an integer.

If the QuerySet is already fully cached this simply returns the length of the cached results set to avoid multiple SELECT COUNT(*) calls.

create(***kwargs*)
Creates a new object with the given kwargs, saving it to the database and returning the created object.

dates(*field_name*, *kind*, *order='ASC'*)
Returns a list of datetime objects representing all available dates for the given field_name, scoped to 'kind'.

defer(**fields*)
Defers the loading of data for certain fields until they are accessed. The set of fields to defer is added to any existing set of deferred fields. The only exception to this is if None is passed in as the only parameter, in which case all deferrals are removed (None acts as a reset option).

delete()
Deletes the records in the current QuerySet.

distinct(**field_names*)
Returns a new QuerySet instance that will select only distinct results.

evaluated()
Lets check if the queryset was already evaluated without accessing private methods / attributes

exclude(**args*, ***kwargs*)
Returns a new QuerySet instance with NOT (args) ANDed to the existing set.

exists()

facet(**args*, ***kwargs*)

facets

filter(**args*, ***kwargs*)
Returns a new QuerySet instance with the args ANDed to the existing set.

classmethod from_qs(*qs*, ***kwargs*)
Creates a new queryset using class *cls* using *qs'* data.

Parameters

- **qs** – The query set to clone
- **kwargs** – The kwargs to pass to _clone method

get(**args*, ***kwargs*)
Performs the query and returns a single object matching the given keyword arguments.

get_or_create(***kwargs*)
Looks up an object with the given kwargs, creating one if necessary. Returns a tuple of (object, created), where created is a boolean specifying whether an object was created.

in_bulk(*id_list*)
Returns a dictionary mapping each of the given IDs to the object with that ID.

index

iterator()
An iterator over the results from applying this QuerySet to the database.

latest(*field_name=None*)
Returns the latest object, according to the model's 'get_latest_by' option or optional given field_name.

none()
Returns an empty QuerySet.

only(**fields*)
Essentially, the opposite of defer. Only the fields passed into this method and that are not already specified as deferred are loaded immediately when the queryset is evaluated.

order_by(**field_names*)
Returns a new QuerySet instance with the ordering changed.

ordered
Returns True if the QuerySet is ordered – i.e. has an order_by() clause or a default ordering on the model.

reverse()
Reverses the ordering of the QuerySet.

size(*number*)

start(*number*)

type

update(***kwargs*)
Updates all elements in the current QuerySet, setting all the given fields to the appropriate values.

using(*alias*)
Selects which database this QuerySet should excecute its query against.

value_annotation = True

values(**fields*)

values_list(**fields*, ***kwargs*)

pyes.queryset.**generate_model**(*index*, *doc_type*, *es_url=None*, *es_kwargs={}*)

pyes.queryset.**get_es_connection**(*es_url*, *es_kwargs*)

pyes.rivers

class pyes.rivers.**CouchDBRiver**(*host='localhost'*, *port=5984*, *db='mydb'*, *filter=None*, *filter_params=None*, *script=None*, *user=None*, *password=None*, ***kwargs*)

serialize()

type = 'couchdb'

class pyes.rivers.**JDBCRiver**(*dbhost='localhost'*, *dbport=5432*, *dbtype='postgresql'*, *dbname=None*, *dbuser=None*, *dbpassword=None*, *poll_time='5s'*, *sql=''*, *name=None*, *params=None*, ***kwargs*)

type = 'jdbc'

class pyes.rivers.**MongoDBRiver**(*servers*, *db*, *collection*, *index_name*, *mapping_type*, *gridfs=False*, *options=None*, *bulk_size=1000*, *filter=None*, ***kwargs*)

type = 'mongodb'

class pyes.rivers.**RabbitMQRiver**(*host='localhost'*, *port=5672*, *user='guest'*, *password='guest'*, *vhost='/'*, *queue='es'*, *exchange='es'*, *routing_key='es'*, *exchange_declare=True*, *exchange_type='direct'*, *exchange_durable=True*, *queue_declare=True*, *queue_durable=True*, *queue_auto_delete=False*, *queue_bind=True*, ***kwargs*)

type = 'rabbitmq'

class pyes.rivers.**River**(*index_name=None*, *index_type=None*, *bulk_size=100*, *bulk_timeout=None*)

serialize()

class pyes.rivers.**TwitterRiver**(*user=None*, *password=None*, ***kwargs*)

type = 'twitter'

pyes.scriptfields

class pyes.scriptfields.**ScriptField**(*script*, *lang='mvel'*, *params=None*, *ignore_failure=False*)

class pyes.scriptfields.**ScriptFields**(*name=None*, *script=None*, *lang=None*, *params=None*, *ignore_failure=False*)

This object create the script_fields definition

add_field(*name*, *script*, *lang=None*, *params=None*, *ignore_failure=False*)

Add a field to script_fields

add_parameter(*field_name*, *param_name*, *param_value*)

Add a parameter to a field into script_fields

The ScriptFields object will be returned, so calls to this can be chained.

serialize()

pyes.utils

pyes.utils.**clean_string**(*text*)

Remove Lucene reserved characters from query string

class pyes.utils.**ESRange**(*field*, *from_value=None*, *to_value=None*, *include_lower=None*, *include_upper=None*, ***kwargs*)

negate()

Reverse the range

serialize()

class pyes.utils.**ESRangeOp**(*field*, *op1*, *value1*, *op2=None*, *value2=None*)

pyes.utils.**string_b64encode**(*s*)

This function is useful to convert a string to a valid id to be used in ES. You can use it to generate an ID for urls or some texts

pyes.utils.**string_b64decode**(*s*)

pyes.utils.**make_path**(**path_components*)

Smash together the path components. Empty components will be ignored.

pyes.utils.**make_id**(*value*)

Build a string id from a value :param value: a text value :return: a string

CHAPTER 4

Change history

Contents

0.19.1

News

- Create Manager to manage API action grouped as Elasticsearch.
- This allows to simplify ES object and to move grouped functionality in manager. We are following the Elastic-Search
- grouping of actions. For now we are adding:
  - Indices Manager: to manage index operation
  - Cluster Manager: to manage index operation
- Renamed field_name in name in ScriptFields
- Got docs building on readthedocs.org (Wraithan - Chris McDonald)
- Added model and scan to search.
- So one can pass custom object to be created
- Added document exists call, to check is a document exists.

Deprecated

Using manager, a lot of es methods are refactored in the managers. This is the list of moved methods:

- .aliases -> .indices.aliases
- .status -> .indices.status
- .create_index -> .indices.create_index
- .create_index_if_missing -> .indices.create_index_if_missing
- .delete_index -> .indices.delete_index
- .exists_index -> .indices.exists_index
- .delete_index_if_exists -> .indices.delete_index_if_exists
- .get_indices -> .indices.get_indices
- .get_closed_indices -> .indices.get_closed_indices
- .get_alias -> .indices.get_alias
- .change_aliases -> .indices.change_aliases
- .add_alias -> .indices.add_alias
- .delete_alias -> .indices.delete_alias
- .set_alias -> .indices.set_alias
- .close_index -> .indices.close_index
- .open_index -> .indices.open_index
- .flush -> .indices.flush
- .refresh -> .indices.refresh
- .optimize -> .indices.optimize

- .analyze -> .indices.analyze
- .gateway_snapshot -> .indices.gateway_snapshot
- .put_mapping -> .indices.put_mapping
- .get_mapping -> .indices.get_mapping
- .cluster_health -> .cluster.cluster_health
- .cluster_state -> .cluster.state
- .cluster_nodes -> .cluster.nodes_info
- .cluster_stats -> .cluster.node_stats
- .index_stats -> .indices.stats
- .delete_mapping -> .indices.delete_mapping
- .get_settings -> .indices.get_settings
- .update_settings -> .indices.update_settings

Fixes

- Fixed ResultSet slicing.
- Moved tests outside pyes code dir. Update references. Upgraded test elasticsearch to 0.19.9.
- Added documentation links.
- Renamed scroll_timeout in scroll.
- Renamed field_name in name in ScriptFields.
- Added routing to delete document call.
- Removed minimum_number_should_match parameter.It is not supported by ElasticSearch and causes errors when using a BoolFilter. (Jernej Kos)
- Improved speed json conversion of datetime values
- Added boost argument to TextQuery. (Jernej Kos)
- Go back to urllib3 instead of requests. (gsakkis)
- Enhance Twitter River class. (thanks @dendright)
- Add OAuth authentication and filtering abilities to Twitter River. (Jack Riches)
- HasChildFilter expects a Query. (gsakkis)
- Fixed _parent being pulled from _meta rather than the instance itself. (merrellb)
- Add support of all_terms to TermFacet. (mouad)

0.19.0

- Use default_indices instead of hardcoding ['_all'] (gsakkis)
- Complete rewrite of connection_http (gsakkis)
- Don't collect info on creation of ES object (patricksmith)

- Add interval to histogram facet. (vrachil)
- Improved connection string construction and added more flexibility. (ferhatsb)
- Fixed pickling DotDict.
- Fixed a bug in Decoder.
- Added execution to TermsFilter. Fixed missing _name attribute in serialized object
- Added _cache and _cache_key parameters to filters.
- Added scope, filter and global parameters to facets. closes #119
- Use a single global ConnectionPool instead of initializing it on every execute call. (gsakkis)
- Allow partial_fields to be passed in the Search class. (tehmaze)
- Propagated parameters to bulker.
- Support params for analyze. (akheron)
- Added LimitFilter.
- Fixed support for query as dict in Search object.
- Added ListBulker implementation and create_bulker method.
- Moved imports to absolute ones.
- Removed inused urllib3 files and added timeout to connection_http.
- Add NotFilter as facet filter (junckritter)
- Add terms facet filter

0.18.7-rc1

- Tested against 0.18.7, with all tests passing
- Added support for index_stats

0.17.0

- API BREAKING: Added new searcher iterator API. (To use the old code rename ".search" in ".search_raw")
- API BREAKING: renamed indexes in indices. To be complaint to ES documentation.
- Tests refactory.
- Add model object to objetify a dict.

0.16.0

- Updated documentation.
- Added TextQuery and some clean up of code.
- Added percolator (matterkkila).
- Added date_histogram facet (zebuline).

- Added script fields to Search object, also add "fields" to TermFacet (aguereca).
- Added analyze_wildcard param to StringQuery (available for ES 0.16.0) (zebuline).
- Add ScriptFields object used as parameter script_fields of Search object (aguereca).
- Add IdsQuery, IdsFilter and delete_by_query (aguereca).
- Bulk delete (acdha).

0.15.0

- Only require simplejson for python < 2.6 (matterkkila)
- Added basic version support to ES.index and Search (merrellb)
- Added scan method to ES. This is only supported on ES Master (pre 0.16) (merrellb)
- Added GeoPointField to mapping types (merrellb)
- Disable thrift in setup.py.
- Added missing _routing property in ObjectField
- Added ExistsFilter
- Improved HasChildren
- Add min_similarity and prefix_length to flt.
- Added _scope to HasChildQuery. (andreiz)
- Added parent/child document in test indexing. Added _scope to HasChildFilter.
- Added MissingFilter as a subclass of TermFilter
- Fixed error in checking TermsQuery (merrellb)
- If an analyzer is set on a field, the returned mapping will have an analyzer
- Add a specific error subtype for mapper parsing exceptions (rboulton)
- Add support for Float numeric field mappings (rboulton)
- ES.get() now accepts "fields" as well as other keyword arguments (eg "routing") (rboulton)
- Allow dump_curl to be passed a filehandle (or still a filename), don't for filenames to be in /tmp, and add a basic test of it.
- Add alias handling (rboulton)
- Add ElasticSearchIllegalArgumentException - used for example when writing to an alias which refers to more than one index. (rboulton)
- Handle errors produced by deleting a missing document, and add a test for it. (rboulton)
- Split Query object into a Search object, for the search specific parts, and a Query base class. Allow ES.search() to take a query or a search object. Make some of the methods of Query base classes chainable, where that is an obviously reasonable thing to do. (rboulton)

0.14.0

- Added delete of mapping type.
- Embedded urllib3 to be buildout safe and for users sake.
- Some code cleanup.
- Added reindex by query (usable only with my elasticsearch git branch).
- Added contrib with mailman indexing.
- Autodetect if django is available and added related functions.
- Code cleanup and PEP8.
- Reactivated the morelikethis query.
- Fixed river support plus unittest. (Tavis Aitken)
- Added autorefresh to sync search and write.
- Added QueryFilter.
- Forced name attribute in multifield declaration.
- Added is_empty to ConstantScoreQuery and fixed some bad behaviour.
- Added CustomScoreQuery.
- Added parent/children indexing.
- Added dump commands in a script file "curl" way.
- Added a lot of fix from Richard Boulton.

0.13.1

- Added jython support (HTTP only for now).

0.13.0

- API Changes: errors -> exceptions.
- Splitting of query/filters.
- Added open/close of index.
- Added the number of retries if server is down.
- Refactory Range query. (Andrei)
- Improved HTTP connection timeout/retries. (Sandymahalo)
- Cleanup some imports. (Sandymahalo)

0.12.1

- Added collecting server info.
- Version 0.12 or above requirement.
- Fixed attachment plugin.
- Updated bulk insert to use new api.
- Added facet support (except geotypes).
- Added river support.
- Cleanup some method.
- Added default_indexes variable.
- Added datetime deserialization.
- Improved performance and memory usage in bulk insert replacing list with StringIO.
- Initial propagation of elasticsearch exception to python.

0.12.0

- Added http transport, added autodetect of transport, updated thrift interface.

0.10.3

- Added bulk insert, explain and facet.

0.10.2

- Added new geo query type.

0.10.1

- Added new connection pool system based on pycassa one.

0.10.0

- Initial working version.

CHAPTER 5

Interesting Links

ElasticSearch

- IRC channel `#elasticsearch` (Freenode)
- ElasticSearch: Search Engine for the cloud.

CHAPTER 6

ES Reference

Reference:

Setup

This section includes information on how to setup *elasticsearch* and get it running. If you haven't already, download it, and then check the *installation* docs.

Configuration

elasticsearch configuration files can be found under **ES_HOME/config** folder. The folder comes with two files, the **elasticsearch.yml** for configuring ElasticSearch different *modules*, and **logging.yml** for configuring the ElasticSearch logging.

Settings

The configuration format is YAML. Here is an example of changing the address all network based modules will use to bind and publish to:

```
network :
    host : 10.0.0.4
```

In production use, you will almost certainly want to change paths for data and log files:

```
path:
  logs: /var/log/elasticsearch
  data: /var/data/elasticsearch
```

Also, don't forget to give your production cluster a name, which is used to discover and auto-join other nodes:

```
cluster:
  name: <NAME OF YOUR CLUSTER>
```

Internally, all settings are collapsed into namespaced" settings. For example, the above gets collapsed into **network.host**. This means that its easy to support other configuration formats, for example, "JSON. If JSON is a preferred configuration format, simply rename the **elasticsearch.yml** file to **elasticsearch.json** and add:

```
{
    "network" : {
        "host" : "10.0.0.4"
    }
}
```

It also means that its easy to provide the settings externally either using the **ES_JAVA_OPTS** or as parameters to the **elasticsearch** command, for example:

```
$ elasticsearch -f -Des.network.host=10.0.0.4
```

Another option is to set **es.default.** prefix instead of **es.** prefix, which means the default setting will be used only if not explicitly set in the configuration file.

Another option is to use the **${...}** notation within the configuration file which will resolve to an environment setting, for example:

```
{
    "network" : {
        "host" : "${ES_NET_HOST}"
    }
}
```

The location of the configuration file can be set externally using a system property:

```
$ elasticsearch -f -Des.config=/path/to/config/file
```

Index Settings

Indices created within the cluster can provide their own settings. For example, the following creates an index with memory based storage instead of the default file system based one (the format can be either YAML or JSON):

```
$ curl -XPUT http://localhost:9200/kimchy/ -d \
'
index :
    store:
        type: memory
'
```

Index level settings can be set on the node level as well, for example, within the **elasticsearch.yml** file, the following can be set:

```
index :
    store:
        type: memory
```

This means that every index that gets created on the specific node started with the mentioned configuration will store the index in memory unless the index explicitly sets it. In other words, any index level settings override what is set in the node configuration. Of course, the above can also be set as a "collapsed" setting, for example:

```
$ elasticsearch -f -Des.index.store.type=memory
```

All of the index level configuration can be found within each *index module*.

Logging

ElasticSearch uses an internal logging abstraction and comes, out of the box, with log4j. _. It tries to simplify log4j configuration by us‘YAML <http://www.yaml.org/>‘_ rg/ to configure it, and the logging configuration file is **config/logging.yml** file.

Dir Layout

The directory layout of an installation is as follows:

Type	Description	Default Location	Setting
home	Home of elasticsearch installation		**path.home**
bin	Binary scripts including **elasticsearch** to start a node	**{path.home}/bin**	
conf	Configuration files including **elasticsearch.yml**	**{path.home}/config**	**path.conf**
data	The location of the data files of each index / shard allocated on the node. Can hold multiple locations.	**{path.home}/data**	**path.data**
work	Temporal files that are used by different nodes.	**{path.home}/work**	**path.work**
logs	Log files location	**{path.home}/logs**	**path.logs**

The multiple data locations allows to stripe it. The striping is simple, placing whole files in one of the locations, and deciding where to place the file based on the location with greatest free space. Note, there is no multiple copies of the same data, in that, its similar to RAID 0. Though simple, it should provide a good solution for people that don’t want to mess with raids and the like. Here is how it is configured:

path.data: /mnt/first,/mnt/second

Or the in an array format:

path.data: [”/mnt/first”, “/mnt/second”]

Installation

After downloading the latest release and extracting it, *elasticsearch* can be started using:

```
$ bin/elasticsearch
```

Under Unix system, the command will start the process in the background. To run it in the foreground, add the -f switch to it:

```
$ bin/elasticsearch -f
```

ElasticSearch is built using Java, and requires Java 6 in order to run. The version of Java that will be used can be set by setting the **JAVA_HOME** environment variable.

Environment Variables

Within the scripts, ElasticSearch comes with built in **JAVA_OPTS** passed to the JVM started. The most important setting for that is the **-Xmx** to control the maximum allowed memory for the process, and **-Xms** to control the minimum allocated memory for the process (in general, the more memory allocated to the process, the better).

Most times it is better to leave the default **JAVA_OPTS** as they are, and use the **ES_JAVA_OPTS** environment variable in order to set / change JVM settings or arguments.

The **ES_HEAP_SIZE** environment variable allows to set the heap memory that will be allocated to elasticsearch java process. It will allocate the same value to both min and max values, though those can be set explicitly (not recommended) by setting **ES_MIN_MEM** (defaults to **256m**), and **ES_MAX_MEM** (defaults to **1gb**).

It is recommended to set the min and max memory to the same value, and enable **mlockall** see later.

UNIX

There are added features when using the **elasticsearch** shell script. The first, which was explained earlier, is the ability to easily run the process either in the foreground or the background.

Another feature is the ability to pass **-X** and **-D** directly to the script. When set, both override anything set using either **JAVA_OPTS** or **ES_JAVA_OPTS**. For example:

```
$ bin/elasticsearch -f -Xmx2g -Xms2g -Des.index.storage.type=memory
```

Important Configurations

File Descriptors

Make sure to increase the number of open files descriptors on the machine (or for the user running elasticsearch). Setting it to 32k or even 64k is recommended.

In order to test how many open files the process can open, start it with **-Des.max-open-files** set to **true**. This will print the number of open files the process can open on startup.

Memory Settings

> There is an option to use mlockall to try and lock the process address space so it won't be swapped. For this to work, the **bootstrap.mlockall** should be set to **true** and it is recommended to set both the min and max memory allocation to be the same.

In order to see if this works or not, set the **common.jna** logging to DEBUG level. A solution to "Unknown mlockall error 0" can be to set **ulimit -l unlimited**.

Note, this is experimental feature, and might cause the JVM or shell session to exit if failing to allocate the memory (because not enough memory is available on the machine).

Running As a Service

It should be simple to wrap the **elasticsearch** script in an **init.d** or the like. But, elasticsearch also supports running it using the Java Service Wrapper.

ElasticSearch can be run as a service using the **elasticsearch** script located under **bin/service** location. The repo for it is located here. The script accepts a single parameter with the following values:

Parameter	Description
console	Run the elasticsearch in the foreground.
start	Run elasticsearch int the background.
stop	Stops elasticsearch if its running.
install	Install elasticsearch to run on system startup (init.d / service).
remove	Removes elasticsearch from system startup (init.d / service).

The service uses Java Service Wrapper which is a small native wrapper around the Java virtual machine which also monitors it.

Note, passing JVM level configuration (such as -X parameters) should be set within the **elasticsearch.conf** file.

The **ES_MIN_MEM** and **ES_MAX_MEM** environment variables to set the minimum and maximum memory allocation for the JVM (set in mega bytes). It defaults to **256** and **1024** respectively.

Api

This section describes the REST APIs *elasticsearch* provides (mainly) using JSON. The API is exposed using *HTTP*, *thrift*, *memcached*.

Nodes

Most cluster level APIs allow to specify which nodes to execute on (for example, getting the node stats for a node). Nodes can be identified in the APIs either using their internal node id, the node name, address, custom attributes, or just the **_local** node receiving the request. For example, here are some sample executions of nodes info:

```
# Local
curl localhost:9200/_cluster/nodes/_local
# Address
curl localhost:9200/_cluster/nodes/10.0.0.3,10.0.0.4
curl localhost:9200/_cluster/nodes/10.0.0.*
# Names
curl localhost:9200/_cluster/nodes/node_name_goes_here
curl localhost:9200/_cluster/nodes/node_name_goes_*
# Attributes (set something like node.rack: 2 in the config)
curl localhost:9200/_cluster/nodes/rack:2
curl localhost:9200/_cluster/nodes/ra*:2
curl localhost:9200/_cluster/nodes/ra*:2*
```

Options

Pretty Results

When appending **?pretty=true** to any request made, the JSON returned will be pretty formatted (use it for debugging only!).

Parameters

Rest parameters (when using HTTP, map to HTTP URL parameters) follow the convention of using underscore casing.

Boolean Values

All REST APIs parameters (both request parameters and JSON body) support providing boolean "false" as the values: **false**, **0** and **off**. All other values are considered "true". Note, this is not related to fields within a document indexed treated as boolean fields.

Number Values

All REST APIs support providing numbered parameters as **string** on top of supporting the native JSON number types.

Result Casing

All REST APIs accept the **case** parameter. When set to **camelCase**, all field names in the result will be returned in camel casing, otherwise, underscore casing will be used. Note, this does not apply to the source document indexed.

JSONP

All REST APIs accept a **callback** parameter resulting in a JSONP result.

Request body in query string

For libraries that don't accept a request body for non-POST requests, you can pass the request body as the **source** query string parameter instead.

Admin Cluster Health

The cluster health API allows to get a very simple status on the health of the cluster.

```
$ curl -XGET 'http://localhost:9200/_cluster/health?pretty=true'
{
  "cluster_name" : "testcluster",
  "status" : "green",
  "timed_out" : false,
  "number_of_nodes" : 2,
  "number_of_data_nodes" : 2,
  "active_primary_shards" : 5,
  "active_shards" : 10,
  "relocating_shards" : 0,
  "initializing_shards" : 0,
  "unassigned_shards" : 0
}
```

The API can also be executed against one or more indices to get just the specified indices health:

```
$ curl -XGET 'http://localhost:9200/_cluster/health/test1,test2'
```

The cluster health is status is: **green**, **yellow** or **red**. On the shard level, a **red** status indicates that the specific shard is not allocated in the cluster, **yellow** means that the primary shard is allocated but replicas are not, and **green** means that all shards are allocated. The index level status is controlled by the worst shard status. The cluster status is controlled by the worst index status.

One of the main benefits of the API is the ability to wait until the cluster reaches a certain high water-mark health level. For example, the following will wait till the cluster reaches the **yellow** level for 50 seconds (if it reaches the **green** or **yellow** status beforehand, it will return):

```
$ curl -XGET 'http://localhost:9200/_cluster/health?wait_for_status=yellow&timeout=50s
↪'
```

Request Parameters

The cluster health API accepts the following request parameters:

Name	Description
level	Can be one of **cluster**, **indices** or **shards**. Controls the details level of the health information returned. Defaults to **cluster**.
wait_for_status	One of **green**, **yellow** or **red**. Will wait (until the timeout provided) until the status of the cluster changes to the one provided. By default, will not wait for any status.
wait_for_relocating_shards	A number controlling to how many relocating shards to wait for. Usually will be **0** to indicate to wait till all relocation have happened. Defaults to not to wait.
wait_for_nodes	The request waits until the specified number **N** of nodes is available. It also accepts **>=N**, **<=N**, **>N** and **<N**. Alternatively, it is possible to use **ge(N)**, **le(N)**, **gt(N)** and **lt(N)** notation.
timeout	A time based parameter controlling how long to wait if one of the wait_for_XXX are provided. Defaults to **30s**.

The following is example of getting cluster health at the **shards** level:

```
$ curl -XGET 'http://localhost:9200/_cluster/health/twitter?level=shards'
```

Admin Cluster Nodes Info

The cluster nodes info API allows to retrieve one or more (or all) of the cluster nodes information.

```
curl -XGET 'http://localhost:9200/_cluster/nodes'
curl -XGET 'http://localhost:9200/_cluster/nodes/nodeId1,nodeId2'

# Shorter Format
curl -XGET 'http://localhost:9200/_nodes'
curl -XGET 'http://localhost:9200/_nodes/nodeId1,nodeId2'
```

The first command retrieves information of all the nodes in the cluster. The second command selectively retrieves nodes information of only **nodeId1** and **nodeId2**. All the nodes selective options are explained *here*.

By default, it just returns the attributes and core settings for a node. It also allows to get information on **settings**, **os**, **process**, **jvm**, **thread_pool**, **network**, **transport** and **http**:

```
curl -XGET 'http://localhost:9200/_nodes?os=true&process=true'
curl -XGET 'http://localhost:9200/_nodes/10.0.0.1/?os=true&process=true'

# Or, specific type endpoint:

curl -XGET 'http://localhost:9200/_nodes/process'
curl -XGET 'http://localhost:9200/_nodes/10.0.0.1/process'
```

Admin Cluster Nodes Shutdown

The nodes shutdown API allows to shutdown one or more (or all) nodes in the cluster. Here is an example of shutting the **_local** node the request is directed to:

```
$ curl -XPOST 'http://localhost:9200/_cluster/nodes/_local/_shutdown'
```

Specific node(s) can be shutdown as well using their respective node ids (or other selective options as explained *here*:

```
$ curl -XPOST 'http://localhost:9200/_cluster/nodes/nodeId1,nodeId2/_shutdown'
```

The master (of the cluster) can also be shutdown using:

```
$ curl -XPOST 'http://localhost:9200/_cluster/nodes/_master/_shutdown'
```

Finally, all nodes can be shutdown using one of the options below:

```
$ curl -XPOST 'http://localhost:9200/_shutdown'

$ curl -XPOST 'http://localhost:9200/_cluster/nodes/_shutdown'

$ curl -XPOST 'http://localhost:9200/_cluster/nodes/_all/_shutdown'
```

Delay

By default, the shutdown will be executed after a 1 seconds delay (**1s**). The delay can be customized by setting the **delay** parameter in a time value format. For example:

```
$ curl -XPOST 'http://localhost:9200/_cluster/nodes/_local/_shutdown?delay=10s'
```

Disable Shutdown

The shutdown API can be disabled by setting **action.disable_shutdown** in the node configuration.

Admin Cluster Nodes Stats

The cluster nodes stats API allows to retrieve one or more (or all) of the cluster nodes statistics.

```
curl -XGET 'http://localhost:9200/_cluster/nodes/stats'
curl -XGET 'http://localhost:9200/_cluster/nodes/nodeId1,nodeId2/stats'

# simplified
curl -XGET 'http://localhost:9200/_nodes/stats'
curl -XGET 'http://localhost:9200/_nodes/nodeId1,nodeId2/stats'
```

The first command retrieves stats of all the nodes in the cluster. The second command selectively retrieves nodes stats of only **nodeId1** and **nodeId2**. All the nodes selective options are explained *here*.

By default, **indices** stats are returned. With options for **indices**, **os**, **process**, **jvm**, **network**, **transport**, **http**, **fs**, and **thread_pool**. For example:

```
# return indices and os
curl -XGET 'http://localhost:9200/_nodes/stats?os=true'
# return just os and process
curl -XGET 'http://localhost:9200/_nodes/stats?clear=true&os=true&process=true'
# specific type endpoint
curl -XGET 'http://localhost:9200/_nodes/process/stats'
curl -XGET 'http://localhost:9200/_nodes/10.0.0.1/process/stats'
# or, if you like the other way
curl -XGET 'http://localhost:9200/_nodes/stats/process'
curl -XGET 'http://localhost:9200/_nodes/10.0.01/stats/process'
```

The **all** flag can be set to return all the stats.

Admin Cluster State

The cluster state API allows to get a comprehensive state information of the whole cluster.

```
$ curl -XGET 'http://localhost:9200/_cluster/state'
```

Response Filters

It is possible to filter the cluster state response using the following REST parameters:

Parameter	Description
filter_nodes	Set to **true** to filter out the **nodes** part of the response.
filter_routing_table	Set to **true** to filter out the **routing_table** part of the response.
filter_metadata	Set to **true** to filter out the **metadata** part of the response.
filter_blocks	Set to **true** to filter out the **blocks** part of the response.
filter_indices	When not filtering metadata, a comma separated list of indices to include in the response.

Example follows:

```
$ curl -XGET 'http://localhost:9200/_cluster/state?filter_nodes=true'
```

Admin Cluster Update Settings

Allow to update cluster wide specific settings. Settings updated can either be persistent (applied cross restarts) or transient (will not survive a full cluster restart). Here is an example:

```
curl -XPUT localhost:9200/_cluster/settings -d '{
    "persistent" : {
        "discovery.zen.minimum_master_nodes" : 2
    }
}'
```

Or:

```
curl -XPUT localhost:9200/_cluster/settings -d '{
    "transient" : {
        "discovery.zen.minimum_master_nodes" : 2
    }
}'
```

There is a specific list of settings that can be updated, those include:

- **cluster.blocks.read_only**: Have the whole cluster read only (indices do not accept write operations), metadata is not allowed to be modified (create or delete indices).
- **discovery.zen.minimum_master_nodes**
- **indices.recovery.concurrent_streams**
- **cluster.routing.allocation.node_initial_primaries_recoveries**, **cluster.routing.allocation.node_concurrent_recoveries**
- **cluster.routing.allocation.cluster_concurrent_rebalance**
- **cluster.routing.allocation.awareness.attributes**
- **cluster.routing.allocation.awareness.force.***
- **cluster.routing.allocation.disable_allocation**
- **cluster.routing.allocation.disable_replica_allocation**
- **cluster.routing.allocation.include.***
- **cluster.routing.allocation.exclude.***
- **indices.cache.filter.size**
- **indices.ttl.interval**
- **indices.recovery.file_chunk_size**, **indices.recovery.translog_ops**, **indices.recovery.translog_size**, **indices.recovery.compress**, **indices.recovery.concurrent_streams**, **indices.recovery.max_size_per_sec**.

Logger values can also be updated by setting **logger.** prefix. More settings will be allowed to be updated.

Cluster wide settings can be returned using **curl -XGET localhost:9200/_cluster/settings**.

Admin Indices Aliases

APIs in elasticsearch accept an index name when working against a specific index, and several indices when applicable. The index aliases API allow to alias an index with a name, with all APIs automatically converting the alias name to the actual index name. An alias can also be mapped to more than one index, and when specifying it, the alias will automatically expand to the aliases indices. An alias can also be associated with a filter that will automatically be applied when searching, and a routing values.

Here is a sample of associating the alias **alias1** with index **test1**:

```
curl -XPOST 'http://localhost:9200/_aliases' -d '
{
    "actions" : [
        { "add" : { "index" : "test1", "alias" : "alias1" } }
    ]
}'
```

An alias can also be removed, for example:

```
curl -XPOST 'http://localhost:9200/_aliases' -d '
{
    "actions" : [
        { "remove" : { "index" : "test1", "alias" : "alias1" } }
    ]
}'
```

Renaming an index is a simple **remove** then **add** operation within the same API:

```
curl -XPOST 'http://localhost:9200/_aliases' -d '
{
    "actions" : [
        { "remove" : { "index" : "test1", "alias" : "alias1" } },
        { "add" : { "index" : "test1", "alias" : "alias2" } }
    ]
}'
```

Associating an alias with more then one index are simply several **add** actions:

```
curl -XPOST 'http://localhost:9200/_aliases' -d '
{
    "actions" : [
        { "add" : { "index" : "test1", "alias" : "alias1" } },
        { "add" : { "index" : "test2", "alias" : "alias1" } }
    ]
}'
```

It is an error to index to an alias which points to more than one index.

Filtered Aliases

Aliases with filters provide an easy way to create different "views" of the same index. The filter can be defined using Query DSL and is applied to all Search, Count, Delete By Query and More Like This operations with this alias. Here is an example:

```
curl -XPOST 'http://localhost:9200/_aliases' -d '
{
    "actions" : [
        {
            "add" : {
                 "index" : "test1",
                 "alias" : "alias2",
                 "filter" : { "term" : { "user" : "kimchy" } }
            }
        }
    ]
}'
```

Routing

Allow to associate routing values with aliases. This feature can be used together with filtering aliases in order to avoid unnecessary shard operations.

The following command creates a new alias **alias1** that points to index **test**. After **alias1** is created, all operations with this alias are automatically modified to use value **1** for routing:

```
curl -XPOST 'http://localhost:9200/_aliases' -d '
{
    "actions" : [
        {
            "add" : {
                 "index" : "test",
```

```
                "alias" : "alias1",
                "routing" : "1"
            }
        }
    ]
}'
```

It's also possible to specify different routing values for searching and indexing operations:

```
curl -XPOST 'http://localhost:9200/_aliases' -d '
{
    "actions" : [
        {
            "add" : {
                "index" : "test",
                "alias" : "alias2",
                "search_routing" : "1,2",
                "index_routing" : "2"
            }
        }
    ]
}'
```

As shown in the example above, search routing may contain several values separated by comma. Index routing can contain only a single value.

If an operation that uses routing alias also has a routing parameter, an intersection of both alias routing and routing specified in the parameter is used. For example the following command will use "2" as a routing value:

```
curl -XGET 'http://localhost:9200/alias2/_search?q=user:kimchy&routing=2,3'
```

Retrieving existing aliases

Aliases can be retrieved using the get aliases API, which can either return all indices with all aliases, or just for specific indices:

```
curl -XGET 'localhost:9200/test/_aliases'
curl -XGET 'localhost:9200/test1,test2/_aliases'
curl -XGET 'localhost:9200/_aliases'
```

Admin Indices Analyze

Performs the analysis process on a text and return the tokens breakdown of the text.

Can be used without specifying an index against one of the many built in analyzers:

```
curl -XGET 'localhost:9200/_analyze?analyzer=standard' -d 'this is a test'
```

Or by building a custom transient analyzer out of tokenizers and filters:

```
curl -XGET 'localhost:9200/_analyze?tokenizer=keyword&filters=lowercase' -d 'this is␣
↪a test'
```

It can also run against a specific index:

```
curl -XGET 'localhost:9200/test/_analyze?text=this+is+a+test'
```

The above will run an analysis on the "this is a test" text, using the default index analyzer associated with the **test** index. An **analyzer** can also be provided to use a different analyzer:

```
curl -XGET 'localhost:9200/test/_analyze?analyzer=whitespace' -d 'this is a test'
```

Also, the analyzer can be derived based on a field mapping, for example:

```
curl -XGET 'localhost:9200/test/_analyze?field=obj1.field1' -d 'this is a test'
```

Will cause the analysis to happen based on the analyzer configure in the mapping for **obj1.field1** (and if not, the default index analyzer).

Also, the text can be provided as part of the request body, and not as a parameter.

Format

By default, the format the tokens are returned in are in json and its called **detailed**. The **text** format value provides the analyzed data in a text stream that is a bit more readable.

Admin Indices Clearcache

The clear cache API allows to clear either all caches or specific cached associated with one ore more indices.

```
$ curl -XPOST 'http://localhost:9200/twitter/_cache/clear'
```

The API, by default, will clear all cached. Specific caches can cleaned explicitly by setting **filter**, **field_data** or **bloom** to **true**.

All caches relating to a specific field(s) can also be cleared by specifying **fields** parameter with a comma delimited list of the relevant fields.

Multi Index

The flush API can be applied to more than one index with a single call, or even on **_all** the indices.

```
$ curl -XPOST 'http://localhost:9200/kimchy,elasticsearch/_cache/clear'

$ curl -XPOST 'http://localhost:9200/_cache/clear'
```

Admin Indices Create Index

The create index API allows to instantiate an index. ElasticSearch provides support for multiple indices, including executing operations across several indices. Each index created can have specific settings associated with it.

```
$ curl -XPUT 'http://localhost:9200/twitter/'

$ curl -XPUT 'http://localhost:9200/twitter/' -d '
index :
    number_of_shards : 3
```

```
    number_of_replicas : 2
'
```

The above second example curl shows how an index called **twitter** can be created with specific settings for it using YAML. In this case, creating an index with 3 shards, each with 2 replicas. The index settings can also defined with JSON:

```
$ curl -XPUT 'http://localhost:9200/twitter/' -d '{
    "settings" : {
        "index" : {
            "number_of_shards" : 3,
            "number_of_replicas" : 2
        }
    }
}'
```

or more simplified

```
$ curl -XPUT 'http://localhost:9200/twitter/' -d '{
    "settings" : {
        "number_of_shards" : 3,
        "number_of_replicas" : 2
    }
}'
```

Note you do not have to explicitly specify **index** section inside **settings** section.

Mappings

The create index API allows to provide a set of one or more mappings:

```
curl -XPOST localhost:9200/test -d '{
    "settings" : {
        "number_of_shards" : 1
    },
    "mappings" : {
        "type1" : {
            "_source" : { "enabled" : false },
            "properties" : {
                "field1" : { "type" : "string", "index" : "not_analyzed" }
            }
        }
    }
}'
```

Index Settings

For more information regarding all the different index level settings that can be set when creating an index, please check the *index modules* section.

Admin Indices Delete Index

The delete index API allows to delete an existing index.

```
$ curl -XDELETE 'http://localhost:9200/twitter/'
```

The above example deletes an index called **twitter**.

The delete index API can also be applied to more than one index, or on **_all** indices (be careful!). All indices will also be deleted when no specific index is provided. In order to disable allowing to delete all indices, set **action.disable_delete_all_indices** setting in the config to **true**.

Admin Indices Delete Mapping

Allow to delete a mapping (type) along with its data. The REST endpoint is **/{index}/{type}** with **DELETE** method.

Note, most times, it make more sense to reindex the data into a fresh index compared to delete large chunks of it.

Admin Indices Flush

The flush API allows to flush one or more indices through an API. The flush process of an index basically frees memory from the index by flushing data to the index storage and clearing the internal transaction log. By default, ElasticSearch uses memory heuristics in order to automatically trigger flush operations as required in order to clear memory.

```
$ curl -XPOST 'http://localhost:9200/twitter/_flush'
```

Request Parameters

The flush API accepts the following request parameters:

Name	Description
refresh	Should a refresh be performed after the flush. Defaults to **false**.

Multi Index

The flush API can be applied to more than one index with a single call, or even on **_all** the indices.

```
$ curl -XPOST 'http://localhost:9200/kimchy,elasticsearch/_flush'

$ curl -XPOST 'http://localhost:9200/_flush'
```

Admin Indices Gateway Snapshot

The gateway snapshot API allows to explicitly perform a snapshot through the gateway of one or more indices (backup them). By default, each index gateway periodically snapshot changes, though it can be disabled and be controlled completely through this API.

Note, this API only applies when using shared storage gateway implementation, and does not apply when using the (default) local gateway.

```
$ curl -XPOST 'http://localhost:9200/twitter/_gateway/snapshot'
```

Multi Index

The gateway snapshot API can be applied to more than one index with a single call, or even on **_all** the indices.

```
$ curl -XPOST 'http://localhost:9200/kimchy,elasticsearch/_gateway/snapshot'

$ curl -XPOST 'http://localhost:9200/_gateway/snapshot'
```

Admin Indices Get Mapping

The get mapping API allows to retrieve mapping definition of index or index/type.

```
$ curl -XGET 'http://localhost:9200/twitter/tweet/_mapping'
```

Multiple Indices and Types

The get mapping API can be used to get more than one index or type mapping with a single call. General usage of the API follows the following syntax: **host:port/{index}/{type}/_mapping** where both **{index}** and **{type}** can stand for comma-separated list of names. To get mappings for all indices you can use **_all** for **{index}**. The following are some examples:

```
$ curl -XGET 'http://localhost:9200/twitter,kimchy/_mapping'

$ curl -XGET 'http://localhost:9200/_all/tweet,book/_mapping'
```

If you want to get mappings of all indices and types then the following two examples are equivalent:

```
$ curl -XGET 'http://localhost:9200/_all/_mapping'

$ curl -XGET 'http://localhost:9200/_mapping'
```

Admin Indices Indices Exists

Used to check if the index (indices) exists or not. For example:

```
curl -XHEAD 'http://localhost:9200/twitter'
```

The HTTP status code indicates if it exists or not. A **404** means its not found, and **200** means its there.

Admin Indices Open Close

The open and close index APIs allow to close an index, and later on opening it. A closed index has almost no overhead on the cluster (except for maintaining its metadata), and is blocked for read/write operations. A closed index can be opened which will then go through the normal recovery process.

The REST endpoint is **/{index}/_close** and **/{index}/_open**. For example:

```
curl -XPOST 'localhost:9200/my_index/_close'

curl -XPOST 'localhost:9200/my_index/_open'
```

Admin Indices Optimize

The optimize API allows to optimize one or more indices through an API. The optimize process basically optimizes the index for faster search operations (and relates to the number of segments a lucene index within each shard). The optimize operation allows to optimize the number of segments to optimize to.

```
$ curl -XPOST 'http://localhost:9200/twitter/_optimize'
```

Request Parameters

The optimize API accepts the following request parameters:

Name	Description
max_num_segments	The number of segments to optimize to. To fully optimize the index, set it to **1**. Defaults to simply checking if a merge needs to execute, and if so, executes it.
only_expunge_deletes	Should the optimize process only expunge segments with deletes in it. In Lucene, a document is not deleted from a segment, just marked as deleted. During a merge process of segments, a new segment is created that does have those deletes. This flag allow to only merge segments that have deletes. Defaults to **false**.
refresh	Should a refresh be performed after the optimize. Defaults to **true**.
flush	Should a flush be performed after the optimize. Defaults to **true**.
wait_for_merge	Should the request wait for the merge to end. Defaults to **true**. Note, a merge can potentially be a very heavy operation, so it might make sense to run it set to **false**.

Multi Index

The optimize API can be applied to more than one index with a single call, or even on **_all** the indices.

```
$ curl -XPOST 'http://localhost:9200/kimchy,elasticsearch/_optimize'

$ curl -XPOST 'http://localhost:9200/_optimize'
```

Admin Indices Put Mapping

The put mapping API allows to register specific mapping definition for a specific type.

```
$ curl -XPUT 'http://localhost:9200/twitter/tweet/_mapping' -d '
{
    "tweet" : {
        "properties" : {
            "message" : {"type" : "string", "store" : "yes"}
        }
    }
}
'
```

The above example creates a mapping called **tweet** within the **twitter** index. The mapping simply defines that the **message** field should be stored (by default, fields are not stored, just indexed) so we can retrieve it later on using selective loading.

More information on how to define type mappings can be found in the *mapping* section.

Merging & Conflicts

When an existing mapping already exists under the given type, the two mapping definitions, the one already defined, and the new ones are merged. The **ignore_conflicts** parameters can be used to control if conflicts should be ignored or not, by default, it is set to **false** which means conflicts are *not* ignored.

The definition of conflict is really dependent on the type merged, but in general, if a different core type is defined, it is considered as a conflict. New mapping definitions can be added to object types, and core type mapping can be upgraded to **multi_field** type.

Multi Index

The put mapping API can be applied to more than one index with a single call, or even on **_all** the indices.

```
$ curl -XPUT 'http://localhost:9200/kimchy,elasticsearch/tweet/_mapping' -d '
{
    "tweet" : {
        "properties" : {
            "message" : {"type" : "string", "store" : "yes"}
        }
    }
}
'
```

Admin Indices Refresh

The refresh API allows to explicitly refresh one or more index, making all operations performed since the last refresh available for search. The (near) real-time capabilities depends on the index engine used. For example, the robin one requires refresh to be called, but by default a refresh is scheduled periodically.

```
$ curl -XPOST 'http://localhost:9200/twitter/_refresh'
```

Multi Index

The refresh API can be applied to more than one index with a single call, or even on **_all** the indices.

```
$ curl -XPOST 'http://localhost:9200/kimchy,elasticsearch/_refresh'

$ curl -XPOST 'http://localhost:9200/_refresh'
```

Admin Indices Segments

Provide low level segments information that a Lucene index (shard level) is built with. Allows to be used to provide more information on the state of a shard and an index, possibly optimization information, data "wasted" on deletes, and so on.

Endpoints include segments for a specific index, several indices, or all:

```
curl -XGET 'http://localhost:9200/test/_segments'
curl -XGET 'http://localhost:9200/test1,test2/_segments'
curl -XGET 'http://localhost:9200/_segments'
```

Admin Indices Stats

Indices level stats provide statistics on different operations happening on an index. The API provides statistics on the index level scope (though most stats can also be retrieved using node level scope).

The following returns high level aggregation and index level stats for all indices:

```
curl localhost:9200/_stats
```

Specific index stats can be retrieved using:

```
curl localhost:9200/index1,index2/_stats
```

By default, **docs**, **store**, and **indexing**, **get**, and **search** stats are returned, other stats can be enabled as well:

- **docs**: The number of docs / deleted docs (docs not yet merged out). Note, affected by refreshing the index.
- **store**: The size of the index.
- **indexing**: Indexing statistics, can be combined with a comma separated list of **types** to provide document type level stats.
- **get**: Get statistics, including missing stats.
- **search**: Search statistics, including custom grouping using the **groups** parameter (search operations can be associated with one or more groups).
- **merge**: merge stats.
- **flush**: flush stats.
- **refresh**: refresh stats.
- **clear**: Clears all the flags (first).

Here are some samples:

```
# Get back stats for merge and refresh on top of the defaults
curl 'localhost:9200/_stats?merge=true&refresh=true'
# Get back stats just for flush
curl 'localhost:9200/_stats?clear=true&flush=true'
# Get back stats for type1 and type2 documents for the my_index index
curl 'localhost:9200/my_index/_stats?clear=true&indexing=true&types=type1,type2
```

The stats returned are aggregated on the index level, with **primaries** and **total** aggregations. In order to get back shard level stats, set the **level** parameter to **shards**.

Note, as shards move around the cluster, their stats will be cleared as they are created on other nodes. On the other hand, even though a shard "left" a node, that node will still retain the stats that shard contributed to.

Specific stats endpoints

Instead of using flags to indicate which stats to return, specific REST endpoints can be used, for example:

```
# Merge stats across all indices
curl localhost:9200/_stats/merge
# Merge stats for the my_index index
curl localhost:9200/my_index/_stats/merge
# Indexing stats for my_index
curl localhost:9200/my_index/_stats/indexing
```

```
# Indexing stats for my_index for my_type1 and my_type2
curl localhost:9200/my_index/_stats/indexing/my_type1,my_type2
```

Admin Indices Status

The indices status API allows to get a comprehensive status information of one or more indices.

```
curl -XGET 'http://localhost:9200/twitter/_status'
```

In order to see the recovery status of shards, pass **recovery** flag and set it to **true**. For snapshot status, pass the **snapshot** flag and set it to **true**.

Multi Index

The status API can be applied to more than one index with a single call, or even on **_all** the indices.

```
curl -XGET 'http://localhost:9200/kimchy,elasticsearch/_status'

curl -XGET 'http://localhost:9200/_status'
```

Admin Indices Templates

Index templates allow to define templates that will automatically be applied to new indices created. The templates include both settings and mappings, and a simple pattern template that controls if the template will be applied to the index created. For example:

```
curl -XPUT localhost:9200/_template/template_1 -d '
{
    "template" : "te*",
    "settings" : {
        "number_of_shards" : 1
    },
    "mappings" : {
        "type1" : {
            "_source" : { "enabled" : false }
        }
    }
}
'
```

Defines a template named template_1, with a template pattern of **te***. The settings and mappings will be applied to any index name that matches the **te*** template.

Deleting a Template

Index templates are identified by a name (in the above case **template_1**) and can be delete as well:

```
curl -XDELETE localhost:9200/_template/template_1
```

GETting a Template

Index templates are identified by a name (in the above case **template_1**) and can be retrieved using the following:

```
curl -XGET localhost:9200/_template/template_1
```

To get list of all index templates you can use *Cluster State* API and check for the metadata/templates section of the response.

Multiple Template Matching

Multiple index templates can potentially match an index, in this case, both the settings and mappings are merged into the final configuration of the index. The order of the merging can be controlled using the **order** parameter, with lower order being applied first, and higher orders overriding them. For example:

```
curl -XPUT localhost:9200/_template/template_1 -d '
{
    "template" : "*",
    "order" : 0,
    "settings" : {
        "number_of_shards" : 1
    },
    "mappings" : {
        "type1" : {
            "_source" : { "enabled" : false }
        }
    }
}
'

curl -XPUT localhost:9200/_template/template_2 -d '
{
    "template" : "te*",
    "order" : 1,
    "settings" : {
        "number_of_shards" : 1
    },
    "mappings" : {
        "type1" : {
            "_source" : { "enabled" : true }
        }
    }
}
'
```

The above will disable storing the **_source** on all **type1** types, but for indices of that start with **te***, source will still be enabled. Note, for mappings, the merging is "deep", meaning that specific object/property based mappings can easily be added/overridden on higher order templates, with lower order templates providing the basis.

Config

Index templates can also be placed within the config location (**path.config**) under the **templates** directory (note, make sure to place them on all master eligible nodes). For example, a file called **template_1.json** can be placed under **config/templates** and it will be added if it matches an index. Here is a sample of a the mentioned file:

```
{
    "template_1" : {
        "template" : "*",
        "settings" : {
            "index.number_of_shards" : 2
        },
        "mappings" : {
            "_default_" : {
                "_source" : {
                    "enabled" : false
                }
            },
            "type1" : {
                "_all" : {
                    "enabled" : false
                }
            }
        }
    }
}
```

Admin Indices Get Settings

The get settings API allows to retrieve settings of index/indices:

```
$ curl -XGET 'http://localhost:9200/twitter/_settings'
```

Admin Indices Update Settings

Change specific index level settings in real time.

The REST endpoint is **/_settings** (to update all indices) or **{index}/_settings** to update one (or more) indices settings. The body of the request includes the updated settings, for example:

```
{
    "index" : {
        "number_of_replicas" : 4
    }
}
```

The above will change the number of replicas to 4 from the current number of replicas. Here is a curl example:

```
curl -XPUT 'localhost:9200/my_index/_settings' -d '
{
    "index" : {
        "number_of_replicas" : 4
    }
}
'
```

Below is the list of settings that can be changed using the update settings API:

Setting	Description
index.number_of_replicas	The number of replicas each shard has.
index.auto_expand_replicas	Set to an actual value (like **0-all**) or **false** to disable it.
index.blocks.read_only	Set to **true** to have the index read only. **false** to allow writes and metadata changes.
index.blocks.read	Set to **true** to disable read operations against the index.
index.blocks.write	Set to **true** to disable write operations against the index.
index.blocks.metadata	Set to **true** to disable metadata operations against the index.
index.refresh_interval	The async refresh interval of a shard.
index.term_index_interval	The Lucene index term interval. Only applies to newly created docs.
index.term_index_divisor	The Lucene reader term index divisor.
index.translog.flush_threshold_ops	When to flush based on operations.
index.translog.flush_threshold_size	When to flush based on translog (bytes) size.
index.translog.flush_threshold_period	When to flush based on a period of not flushing.
index.translog.disable_flush	Disables flushing. Note, should be set for a short interval and then enabled.
index.cache.filter.max_size	The maximum size of filter cache (per segment in shard). Set to **-1** to disable.
index.cache.filter.expire	The expire after access time for filter cache. Set to **-1** to disable.
index.gateway.snapshot_interval	The gateway snapshot interval (only applies to shared gateways).
"merge policy":/guide/reference/index-modules/merge.html	All the settings for the merge policy currently configured. A different merge policy can't be set.
index.routing.allocation.include.*	Controls which nodes the index will be allowed to be allocated on.
index.routing.allocation.exclude.*	Controls which nodes the index will not be allowed ot be allocated on.
index.routing.allocation.total_shards_per_node	Controls the total number of shards allowed to be allocated on a single node. Default is unbounded.
index.recovery.initial_shards	When using local gateway a particular shard is recovered only if there can be allocated quorum shards in the cluster. It can be set to **quorum** (default), **quorum-1** (or **half**), **full** and **full-1**. Number values are also supported, e.g. **1**.
index.gc_deletes	
index.ttl.disable_purge	Disables temporarily the purge of expired docs.

Bulk Indexing Usage

For example, the update settings API can be used to dynamically change the index from being more performant for bulk indexing, and then move it to more real time indexing state. Before the bulk indexing is started, use:

```
curl -XPUT localhost:9200/test/_settings -d '{
    "index" : {
        "refresh_interval" : "-1"
    }
}'
```

(Another optimization option is to start the index without any replicas, and only later adding them, but that really depends on the use case).

Then, once bulk indexing is done, the settings can be updated (back to the defaults for example):

```
curl -XPUT localhost:9200/test/_settings -d '{
    "index" : {
        "refresh_interval" : "1s"
    }
}'
```

And, an optimize should be called:

```
curl -XPOST 'http://localhost:9200/test/_optimize?max_num_segments=5'
```

Bulk

The bulk API makes it possible to perform many index/delete operations in a single API call. This can greatly increase the indexing speed. The REST API endpoint is **/_bulk**, and it expects the following JSON structure:

```
action_and_meta_data\n
optional_source\n
action_and_meta_data\n
optional_source\n
....
action_and_meta_data\n
optional_source\n
```

NOTE: the final line of data must end with a newline character **n**.

If you're providing text file input to **curl**, you *must* **–data-binary** instead of plain **-d**. The latter doesn't preserve newlines. Example:

```
$ cat requests
{ "index" : { "_index" : "test", "_type" : "type1", "_id" : "1" } }
{ "field1" : "value1" }
$ curl -s -XPOST localhost:9200/_bulk --data-binary **requests; echo
{"took":7,"items":[{"create":{"_index":"test","_type":"type1","_id":"1","_version":1,
↪"ok":true}}]}
```

Because this format uses literal **n**'s as delimiters, please be sure that the JSON actions and sources are not pretty printed. Here is an example of a correct sequence of bulk commands:

```
{ "index" : { "_index" : "test", "_type" : "type1", "_id" : "1" } }
{ "field1" : "value1" }
{ "delete" : { "_index" : "test", "_type" : "type1", "_id" : "2" } }
{ "create" : { "_index" : "test", "_type" : "type1", "_id" : "3" } }
{ "field1" : "value3" }
```

The endpoints are **/_bulk**, **/{index}/_bulk**, and **{index}/type/_bulk**. When the index or the index/type are provided, they will be used by default on bulk items that don't provide them explicitly.

A note on the format. The idea here is to make processing of this as fast as possible. As some of the actions will be redirected to other shards on other nodes, only **action_meta_data** is parsed on the receiving node side.

Client libraries using this protocol should try and strive to do something similar on the client side, and reduce buffering as much as possible.

The response to a bulk action is a large JSON structure with the individual results of each action that was performed. The failure of a single action does not affect the remaining actions.

There is no "correct" number of actions to perform in a single bulk call. You should experiment with different settings to find the optimum size for your particular workload.

If using the HTTP API, make sure that the client does not send HTTP chunks, as this will slow things down.

Versioning

Each bulk item can include the version value using the **_version/version** field. It automatically follows the behavior of the index / delete operation based on the **_version** mapping. It also support the **version_type/_version_type** when using **external** versioning.

Routing

Each bulk item can include the routing value using the **_routing/routing** field. It automatically follows the behavior of the index / delete operation based on the **_routing** mapping.

Percolator

Each bulk index action can include a percolate value using the **_percolate/percolate** field.

Parent

Each bulk item can include the parent value using the **_parent/parent** field. It automatically follows the behavior of the index / delete operation based on the **_parent / _routing** mapping.

Timestamp

Each bulk item can include the timestamp value using the **_timestamp/timestamp** field. It automatically follows the behavior of the index operation based on the **_timestamp** mapping.

TTL

Each bulk item can include the ttl value using the **_ttl/ttl** field. It automatically follows the behavior of the index operation based on the **_ttl** mapping.

Write Consistency

When making bulk calls, you can require a minimum number of active shards in the partition through the **consistency** parameter. The values allowed are **one**, **quorum**, and **all**. It defaults to the node level setting of **action.write_consistency**, which in turn defaults to **quorum**.

For example, in a N shards with 2 replicas index, there will have to be at least 2 active shards within the relevant partition (**quorum**) for the operation to succeed. In a N shards with 1 replica scenario, there will need to be a single shard active (in this case, **one** and **quorum** is the same).

Refresh

The **refresh** parameter can be set to **true** in order to refresh the relevant shards immediately after the bulk operation has occurred and make it searchable, instead of waiting for the normal refresh interval to expire. Setting it to **true** can trigger additional load, and may slow down indexing.

Count

The count API allows to easily execute a query and get the number of matches for that query. It can be executed across one or more indices and across one or more types. The query can either be provided using a simple query string as a parameter, or using the *Query DSL* defined within the request body. Here is an example:

```
$ curl -XGET 'http://localhost:9200/twitter/tweet/_count?q=user:kimchy'

$ curl -XGET 'http://localhost:9200/twitter/tweet/_count' -d '
{
    "term" : { "user" : "kimchy" }
}'
```

Both examples above do the same thing, which is count the number of tweets from the twitter index for a certain user. The result is:

```
{
    "count" : 1,
    "_shards" : {
        "total" : 5,
        "successful" : 5,
        "failed" : 0
    }
}
```

The query is optional, and when not provided, it will use **match_all** to count all the docs.

Multiple Indices and Types

The count API can be applied to multiple types within an index, and across multiple indices. For example, we can count all documents across all types that match a **user** field with value **kimchy**:

```
$ curl -XGET 'http://localhost:9200/_count?q=user:kimchy'
```

We can also count within specific types:

```
$ curl -XGET 'http://localhost:9200/twitter/tweet,user/_count?q=user:kimchy'
```

We can also count all tweets with a certain tag across several indices (for example, when each user has his own index):

```
$ curl -XGET 'http://localhost:9200/kimchy,elasticsearch/_count?q=tag:wow'
```

Or even count across all indices:

```
$ curl -XGET 'http://localhost:9200/_count?q=tag:wow'
```

Request Parameters

When executing count using the query parameter **q**, the query passed is a query string using Lucene query parser. There are additional parameters that can be passed:

Name	Description
df	The default field to use when no field prefix is defined within the query.
analyzer	The analyzer name to be used when analyzing the query string.
default_operator	The default operator to be used, can be **AND** or **OR**. Defaults to **OR**.

Request Body

The count can use the *Query DSL* within its body in order to express the query that should be executed. The body content can also be passed as a REST parameter named **source**.

Note Both HTTP GET and HTTP POST can be used to execute count with body. Since not all clients support GET with body, POST is allowed as well.

Distributed

The count operation is broadcast across all shards. For each shard id group, a replica is chosen and executed against it. This means that replicas increase the scalability of count.

Routing

The routing value (a comma separated list of the routing values) can be specified to control which shards the count request will be executed on.

Delete By Query

The delete by query API allows to delete documents from one or more indices and one or more types based on a query. The query can either be provided using a simple query string as a parameter, or using the *Query DSL* defined within the request body. Here is an example:

```
$ curl -XDELETE 'http://localhost:9200/twitter/tweet/_query?q=user:kimchy'

$ curl -XDELETE 'http://localhost:9200/twitter/tweet/_query' -d '{
    "term" : { "user" : "kimchy" }
}
'
```

Both above examples end up doing the same thing, which is delete all tweets from the twitter index for a certain user. The result of the commands is:

```
{
    "ok" : true,
    "_indices" : {
        "twitter" : {
            "_shards" : {
                "total" : 5,
                "successful" : 5,
                "failed" : 0
```

```
            }
        }
    }
}
```

Note, delete by query bypasses versioning support. Also, it is not recommended to delete "large chunks of the data in an index", many times, its better to simply reindex into a new index.

Multiple Indices and Types

The delete by query API can be applies to multiple types within an index, and across multiple indices. For example, we can delete all documents across all types within the twitter index:

```
$ curl -XDELETE 'http://localhost:9200/twitter/_query?q=user:kimchy'
```

We can also delete within specific types:

```
$ curl -XDELETE 'http://localhost:9200/twitter/tweet,user/_query?q=user:kimchy'
```

We can also delete all tweets with a certain tag across several indices (for example, when each user has his own index):

```
$ curl -XDELETE 'http://localhost:9200/kimchy,elasticsearch/_query?q=tag:wow'
```

Or even delete across all indices:

```
$ curl -XDELETE 'http://localhost:9200/_all/_query?q=tag:wow'
```

Request Parameters

When executing a delete by query using the query parameter **q**, the query passed is a query string using Lucene query parser. There are additional parameters that can be passed:

Name	Description
df	The default field to use when no field prefix is defined within the query.
analyzer	The analyzer name to be used when analyzing the query string.
default_operator	The default operator to be used, can be **AND** or **OR**. Defaults to **OR**.

Request Body

The delete by query can use the *Query DSL* within its body in order to express the query that should be executed and delete all documents. The body content can also be passed as a REST parameter named **source**.

Distributed

The delete by query API is broadcast across all primary shards, and from there, replicated across all shards replicas.

Routing

The routing value (a comma separated list of the routing values) can be specified to control which shards the delete by query request will be executed on.

Replication Type

The replication of the operation can be done in an asynchronous manner to the replicas (the operation will return once it has be executed on the primary shard). The **replication** parameter can be set to **async** (defaults to **sync**) in order to enable it.

Write Consistency

Control if the operation will be allowed to execute based on the number of active shards within that partition (replication group). The values allowed are **one**, **quorum**, and **all**. The parameter to set it is **consistency**, and it defaults to the node level setting of **action.write_consistency** which in turn defaults to **quorum**.

For example, in a N shards with 2 replicas index, there will have to be at least 2 active shards within the relevant partition (**quorum**) for the operation to succeed. In a N shards with 1 replica scenario, there will need to be a single shard active (in this case, **one** and **quorum** is the same).

Delete

The delete API allows to delete a typed JSON document from a specific index based on its id. The following example deletes the JSON document from an index called twitter, under a type called tweet, with id valued 1:

```
$ curl -XDELETE 'http://localhost:9200/twitter/tweet/1'
```

The result of the above delete operation is:

```
{
    "ok" : true,
    "_index" : "twitter",
    "_type" : "tweet",
    "_id" : "1",
    "found" : true
}
```

Versioning

Each document indexed is versioned. When deleting a document, the **version** can be specified to make sure the relevant document we are trying to delete is actually being deleted and it has not changed in the meantime.

Routing

When indexing using the ability to control the routing, in order to delete a document, the routing value should also be provided. For example:

```
$ curl -XDELETE 'http://localhost:9200/twitter/tweet/1?routing=kimchy'
```

The above will delete a tweet with id 1, but will be routed based on the user. Note, issuing a delete without the correct routing, will cause the document to not be deleted.

Many times, the routing value is not known when deleting a document. For those cases, when specifying the **_routing** mapping as **required**, and no routing value is specified, the delete will be broadcasted automatically to all shards.

Parent

The **parent** parameter can be set, which will basically be the same as setting the routing parameter.

Note that deleting a parent document does not automatically delete its children. One way of deleting all child documents given a parent's id is to perform a *delete by query* on the child index with the automatically generated (and indexed) field ==_parent==, which is in the format ==parent_type#parent_id==.

Automatic for the Index

The delete operation automatically creates an index if it has not been created before (check out the *create index API* for manually creating an index), and also automatically creates a dynamic type mapping for the specific type if it has not been created before (check out the *put mapping* API for manually creating type mapping).

Distributed

The delete operation gets hashed into a specific shard id. It then gets redirected into the primary shard within that id group, and replicated (if needed) to shard replicas within that id group.

Replication Type

The replication of the operation can be done in an asynchronous manner to the replicas (the operation will return once it has be executed on the primary shard). The **replication** parameter can be set to **async** (defaults to **sync**) in order to enable it.

Write Consistency

Control if the operation will be allowed to execute based on the number of active shards within that partition (replication group). The values allowed are **one**, **quorum**, and **all**. The parameter to set it is **consistency**, and it defaults to the node level setting of **action.write_consistency** which in turn defaults to **quorum**.

For example, in a N shards with 2 replicas index, there will have to be at least 2 active shards within the relevant partition (**quorum**) for the operation to succeed. In a N shards with 1 replica scenario, there will need to be a single shard active (in this case, **one** and **quorum** is the same).

Refresh

The **refresh** parameter can be set to **true** in order to refresh the relevant shard after the delete operation has occurred and make it searchable. Setting it to **true** should be done after careful thought and verification that this does not cause a heavy load on the system (and slows down indexing).

Get

The get API allows to get a typed JSON document from the index based on its id. The following example gets a JSON document from an index called twitter, under a type called tweet, with id valued 1:

```
curl -XGET 'http://localhost:9200/twitter/tweet/1'
```

The result of the above get operation is:

```
{
    "_index" : "twitter",
    "_type" : "tweet",
    "_id" : "1",
    "_source" : {
        "user" : "kimchy",
        "postDate" : "2009-11-15T14:12:12",
        "message" : "trying out Elastic Search"
    }
}
```

The above result includes the **_index**, **_type**, and **_id** of the document we wish to retrieve, including the actual source of the document that was indexed.

The API also allows to check for the existance of a document using **HEAD**, for example:

```
curl -XHEAD 'http://localhost:9200/twitter/tweet/1'
```

Realtime

By default, the get API is realtime, and is not affected by the refresh rate of the index (when data will become visible for search).

In order to disable realtime GET, one can either set **realtime** parameter to **false**, or globally default it to by setting the **action.get.realtime** to **false** in the node configuration.

When getting a document, one can specify **fields** to fetch from it. They will, when possible, be fetched as stored fields (fields mapped as stored in the mapping). When using realtime GET, there is no notion of stored fields (at least for a period of time, basically, until the next flush), so they will be extracted from the source itself (note, even if source is not enabled). It is a good practice to assume that the fields will be loaded from source when using realtime GET, even if the fields are stored.

Optional Type

The get API allows for **_type** to be optional. Set it to **_all** in order to fetch the first document matching the id across all types.

Fields

The get operation allows to specify a set of fields that will be returned (by default, its the **_source** field) by passing the **fields** parameter. For example:

```
curl -XGET 'http://localhost:9200/twitter/tweet/1?fields=title,content'
```

The returned fields will either be loaded if they are stored, or fetched from the **_source** (parsed and extracted). It also supports sub objects extraction from _source, like **obj1.obj2**.

Routing

When indexing using the ability to control the routing, in order to get a document, the routing value should also be provided. For example:

```
curl -XGET 'http://localhost:9200/twitter/tweet/1?routing=kimchy'
```

The above will get a tweet with id 1, but will be routed based on the user. Note, issuing a get without the correct routing, will cause the document not to be fetched.

Preference

Controls a **preference** of which shard replicas to execute the get request on. By default, the operation is randomized between the each shard replicas.

The **preference** can be set to:

- **_primary**: The operation will go and be executed only on the primary shards.
- **_local**: The operation will prefer to be executed on a local allocated shard is possible.
- Custom (string) value: A custom value will be used to guarantee that the same shards will be used for the same custom value. This can help with "jumping values" when hitting different shards in different refresh states. A sample value can be something like the web session id, or the user name.

Refresh

The **refresh** parameter can be set to **true** in order to refresh the relevant shard before the get operation and make it searchable. Setting it to **true** should be done after careful thought and verification that this does not cause a heavy load on the system (and slows down indexing).

Distributed

The get operation gets hashed into a specific shard id. It then gets redirected to one of the replicas within that shard id and returns the result. The replicas are the primary shard and its replicas within that shard id group. This means that the more replicas we will have, the better GET scaling we will have.

Index

The index API adds or updates a typed JSON document in a specific index, making it searchable. The following example inserts the JSON document into the "twitter" index, under a type called "tweet" with an id of 1:

```
$ curl -XPUT 'http://localhost:9200/twitter/tweet/1' -d '{
    "user" : "kimchy",
    "post_date" : "2009-11-15T14:12:12",
    "message" : "trying out Elastic Search"
}'
```

The result of the above index operation is:

```
{
    "ok" : true,
    "_index" : "twitter",
    "_type" : "tweet",
    "_id" : "1"
}
```

Automatic Index Creation

The index operation automatically creates an index if it has not been created before (check out the *create index API* for manually creating an index), and also automatically creates a dynamic type mapping for the specific type if one has not yet been created (check out the *put mapping* API for manually creating a type mapping).

The mapping itself is very flexible and is schema-free. New fields and objects will automatically be added to the mapping definition of the type specified. Check out the *mapping* section for more information on mapping definitions.

Though explained on the *mapping* section, its important to note that the format of the JSON document can also include the type (very handy when using JSON mappers), for example:

```
$ curl -XPUT 'http://localhost:9200/twitter/tweet/1' -d '{
    "tweet" : {
        "user" : "kimchy",
        "post_date" : "2009-11-15T14:12:12",
        "message" : "trying out Elastic Search"
    }
}'
```

Automatic index creation can be disabled by setting **action.auto_create_index** to **false** in the config file of all nodes. Automatic mapping creation can be disabled by setting **index.mapper.dynamic** to **false** in the config files of all nodes (or on the specific index settings).

Versioning

Each indexed document is given a version number. The associated **version** number is returned as part of the response to the index API request. The index API optionally allows for optimistic concurrency control when the **version** parameter is specified. This will control the version of the document the operation is intended to be executed against. A good example of a use case for versioning is performing a transactional read-then-update. Specifying a **version** from the document initially read ensures no changes have happened in the meantime (when reading in order to update, it is recommended to set **preference** to **_primary**). For example:

```
curl -XPUT 'localhost:9200/twitter/tweet/1?version=2' -d '{
    "message" : "elasticsearch now has versioning support, double cool!"
}'
```

NOTE: versioning is completely real time, and is not affected by the near real time aspects of search operations. If no version is provided, then the operation is executed without any version checks.

By default, internal versioning is used that starts at 1 and increments with each update. Optionally, the version number can be supplemented with an external value (for example, if maintained in a database). To enable this functionality, **version_type** should be set to **external**. The value provided must be a numeric, long value greater than 0, and less than around 9.2e+18. When using the external version type, instead of checking for a matching version number, the system checks to see if the version number passed to the index request is greater than the version of the currently stored document. If true, the document will be indexed and the new version number used. If the value provided is less than or equal to the stored document's version number, a version conflict will occur and the index operation will fail.

A nice side effect is that there is no need to maintain strict ordering of async indexing operations executed a result of changes to a source database, as long as version numbers from the source database are used. Even the simple case of updating the elasticsearch index using data from a database is simplified if external versioning is used, as only the latest version will be used if the index operations are out of order for whatever reason.

Operation Type

The index operation also accepts an **op_type** that can be used to force a **create** operation, allowing for "put-if-absent" behavior. When **create** is used, the index operation will fail if a document by that id already exists in the index.

Here is an example of using the **op_type** parameter:

```
$ curl -XPUT 'http://localhost:9200/twitter/tweet/1?op_type=create' -d '{
    "user" : "kimchy",
    "post_date" : "2009-11-15T14:12:12",
    "message" : "trying out Elastic Search"
}'
```

Another option to specify **create** is to use the following uri:

```
$ curl -XPUT 'http://localhost:9200/twitter/tweet/1/_create' -d '{
    "user" : "kimchy",
    "post_date" : "2009-11-15T14:12:12",
    "message" : "trying out Elastic Search"
}'
```

Automatic ID Generation

The index operation can be executed without specifying the id. In such a case, an id will be generated automatically. In addition, the **op_type** will automatically be set to **create**. Here is an example (note the *POST* used instead of *PUT*):

```
$ curl -XPOST 'http://localhost:9200/twitter/tweet/' -d '{
    "user" : "kimchy",
    "post_date" : "2009-11-15T14:12:12",
    "message" : "trying out Elastic Search"
}'
```

The result of the above index operation is:

```
{
    "ok" : true,
    "_index" : "twitter",
    "_type" : "tweet",
    "_id" : "6a8ca01c-7896-48e9-81cc-9f70661fcb32"
}
```

Routing

By default, shard placement — or **routing** — is controlled by using a hash of the document's id value. For more explicit control, the value fed into the hash function used by the router can be directly specified on a per-operation basis using the **routing** parameter. For example:

```
$ curl -XPOST 'http://localhost:9200/twitter/tweet?routing=kimchy' -d '{
    "user" : "kimchy",
    "post_date" : "2009-11-15T14:12:12",
    "message" : "trying out Elastic Search"
}'
```

In the example above, the "tweet" document is routed to a shard based on the **routing** parameter provided: "kimchy".

When setting up explicit mapping, the **_routing** field can be optionally used to direct the index operation to extract the routing value from the document itself. This does come at the (very minimal) cost of an additional document parsing pass. If the **_routing** mapping is defined, and set to be **required**, the index operation will fail if no routing value is provided or extracted.

Parents & Children

A child document can be indexed by specifying it's parent when indexing. For example:

```
$ curl -XPUT localhost:9200/blogs/blog_tag/1122?parent=1111 -d '{
    "tag" : "something"
}'
```

When indexing a child document, the routing value is automatically set to be the same as it's parent, unless the routing value is explicitly specified using the **routing** parameter.

Timestamp

A document can be indexed with a **timestamp** associated with it. The **timestamp** value of a document can be set using the **timestamp** parameter. For example:

```
$ curl -XPUT localhost:9200/twitter/tweet/1?timestamp=2009-11-15T14%3A12%3A12 -d '{
    "user" : "kimchy",
    "message" : "trying out Elastic Search",
}'
```

If the **timestamp** value is not provided externally or in the **_source**, the **timestamp** will be automatically set to the date the document was processed by the indexing chain. More information can be found on the *_timestamp mapping page*.

TTL

A document can be indexed with a **ttl** (time to live) associated with it. Expired documents will be expunged automatically. The expiration date that will be set for a document with a provided **ttl** is relative to the **timestamp** of the document, meaning it can be based on the time of indexing or on any time provided. The provided **ttl** must be strictly positive and can be a number (in milliseconds) or any valid time value as shown in the following examples:

```
curl -XPUT 'http://localhost:9200/twitter/tweet/1?ttl=86400000' -d '{
    "user": "kimchy",
    "message": "Trying out elasticsearch, so far so good?"
}'
```

```
curl -XPUT 'http://localhost:9200/twitter/tweet/1?ttl=1d' -d '{
    "user": "kimchy",
    "message": "Trying out elasticsearch, so far so good?"
}'
```

```
curl -XPUT 'http://localhost:9200/twitter/tweet/1' -d '{
    "_ttl": "1d",
    "user": "kimchy",
    "message": "Trying out elasticsearch, so far so good?"
}'
```

More information can be found on the *_ttl mapping page*. Percolate =========

Percolation can be performed at index time by passing the **percolate** parameter. Setting it to * will cause all percolation queries registered against the index to be checked against the provided document, for example:

```
curl -XPUT localhost:9200/test/type1/1?percolate=* -d '{
    "field1" : "value1"
}'
```

To filter out which percolator queries will be executed, pass the query string syntax to the **percolate** parameter:

```
curl -XPUT localhost:9200/test/type1/1?percolate=color:green -d '{
    "field1" : "value1",
    "field2" : "value2"
}'
```

NOTE: In a distributed cluster, percolation during the index operation is performed on the primary shard, as soon as the index operation completes. The operation executes on the primary while the replicas are updating, concurrently. Percolation during the index operation somewhat cuts down on parsing overhead, as the parse tree for the document is simply re-used for percolation.

Distributed

The index operation is directed to the primary shard based on it's route (see the Routing section above) and performed on the actual node containing this shard. After the primary shard completes the operation, if needed, the update is distributed to applicable replicas.

Write Consistency

To prevent writes from taking place on the "wrong" side of a network partition, by default, index operations only succeed if a quorum (>replicas/2+1) of active shards are available. This default can be overridden on a node-by-node basis using the **action.write_consistency** setting. To alter this behavior per-operation, the **consistency** request parameter can be used.

Valid write consistency values are **one**, **quorum**, and **all**.

Asynchronous Replication

By default, the index operation only returns after all shards within the replication group have indexed the document (sync replication). To enable asynchronous replication, causing the replication process to take place in the background, set the **replication** parameter to **async**. When asynchronous replication is used, the index operation will return as soon as the operation succeeds on the primary shard.

Refresh

To refresh the index immediately after the operation occurs, so that the document appears in search results immediately, the **refresh** parameter can be set to **true**. Setting this option to **true** should *ONLY* be done after careful thought and verification that it does not lead to poor performance, both from an indexing and a search standpoint. Note, getting a document using the get API is completely realtime.

Timeout

The primary shard assigned to perform the index operation might not be available when the index operation is executed. Some reasons for this might be that the primary shard is currently recovering from a gateway or undergoing relocation. By default, the index operation will wait on the primary shard to become available for up to 1 minute before failing and responding with an error. The **timeout** parameter can be used to explicitly specify how long it waits. Here is an example of setting it to 5 minutes:

```
$ curl -XPUT 'http://localhost:9200/twitter/tweet/1?timeout=5m' -d '{
    "user" : "kimchy",
    "post_date" : "2009-11-15T14:12:12",
    "message" : "trying out Elastic Search"
}'
```

More Like This

The more like this (mlt) API allows to get documents that are "like" a specified document. Here is an example:

```
$ curl -XGET 'http://localhost:9200/twitter/tweet/1/_mlt?mlt_fields=tag,content&min_
↪doc_freq=1'
```

The API simply results in executing a search request with *moreLikeThis* query (http parameters match the parameters to the **more_like_this** query). This means that the body of the request can optionally include all the request body options in the *search API* (facets, from/to and so on).

Rest parameters relating to search are also allowed, including **search_type**, **search_indices**, **search_types**, **search_scroll**, **search_size** and **search_from**.

When no **mlt_fields** are specified, all the fields of the document will be used in the **more_like_this** query generated.

Multi Get

Multi GET API allows to get multiple documents based on an index, type (optional) and id (and possibly routing). The response includes a **doc** array with all the documents each element similar in structure to the get API. Here is an example:

```
curl 'localhost:9200/_mget' -d '{
    "docs" : [
        {
            "_index" : "test",
            "_type" : "type",
            "_id" : "1"
        },
        {
            "_index" : "test",
            "_type" : "type",
            "_id" : "2"
        }
    ]
}'
```

The **mget** endpoint can also be used against an index (in which case its not required in the body):

```
curl 'localhost:9200/test/_mget' -d '{
    "docs" : [
        {
            "_type" : "type",
            "_id" : "1"
        },
        {
            "_type" : "type",
            "_id" : "2"
        }
    ]
}'
```

And type:

```
curl 'localhost:9200/test/type/_mget' -d '{
    "docs" : [
        {
            "_id" : "1"
        },
        {
            "_id" : "2"
        }
    ]
}'
```

In which case, the **ids** element can directly be used to simplify the request:

```
curl 'localhost:9200/test/type/_mget' -d '{
    "ids" : ["1", "2"]
}'
```

Fields

Specific fields can be specified to be retrieved per document to get. For example:

```
curl 'localhost:9200/_mget' -d '{
    "docs" : [
        {
            "_index" : "test",
            "_type" : "type",
            "_id" : "1",
            "fields" : ["field1", "field2"]
        },
        {
            "_index" : "test",
            "_type" : "type",
            "_id" : "2",
            "fields" : ["field3", "field4"]
        }
    ]
}'
```

Multi Index

Many of elasticearch APIs support operating across several indices. The format usually includes an index name, or a comma separated index names. It also supports using aliases, or a comma delimited list of aliases and indexes for the APIs.

Using the APIs against all indices is simple by usually either using the **_all** index, or simply omitting the index.

For multi index APIs (like search), there is also support for using wildcards (since **0.19.8**) in order to resolve indices and aliases. For example, if we have an indices **test1**, **test2** and **test3**, we can simply specify **test*** and automatically operate on all of them. The syntax also support the ability to add (**+**) and remove (**-**), for example: **+test*,-test3**.

Multi Search

The multi search API allows to execute several search requests within the same API. The endpoint for it is **_msearch** (available from **0.19** onwards).

The format of the request is similar to the bulk API format, and the structure is as follows (the structure is specifically optimized to reduce parsing if a specific search ends up redirected to another node):

```
header\n
body\n
header\n
body\n
```

The header part includes which index / indices to search on, optional (mapping) types to search on, the **search_type**, **preference**, and **routing**. The body includes the typical search body request (including the **query**, **facets**, **from**, **size**, and so on). Here is an example:

```
$ cat requests
{"index" : "test"}
{"query" : {"match_all" : {}}, "from" : 0, "size" : 10}
{"index" : "test", "search_type" : "count"}
{"query" : {"match_all" : {}}}
{}
{"query" : {"match_all" : {}}}

{"query" : {"match_all" : {}}}
{"search_type" : "count"}
{"query" : {"match_all" : {}}}

$ curl -XGET localhost:9200/_msearch --data-binary **requests; echo
```

Note, the above includes an example of an empty header (can also be just without any content) which is supported as well.

The response returns a **responses** array, which includes the search response for each search request matching its order in the original multi search request. If there was a complete failure for that specific search request, an object with **error** message will be returned in place of the actual search response.

The endpoint allows to also search against an index/indices and type/types in the URI itself, in which case it will be used as the default unless explicitly defined otherwise in the header. For example:

```
$ cat requests
{}
{"query" : {"match_all" : {}}, "from" : 0, "size" : 10}
{}
```

```
{"query" : {"match_all" : {}}}
{"index" : "test2"}
{"query" : {"match_all" : {}}}

$ curl -XGET localhost:9200/test/_msearch --data-binary **requests; echo
```

The above will execute the search against the **test** index for all the requests that don't define an index, and the last one will be executed against the **test2** index.

The **search_type** can be set in a similar manner to globally apply to all search requests.

Percolate

The percolator allows to register queries against an index, and then send **percolate** requests which include a doc, and getting back the queries that match on that doc out of the set of registered queries.

Think of it as the reverse operation of indexing and then searching. Instead of sending docs, indexing them, and then running queries. One sends queries, registers them, and then sends docs and finds out which queries match that doc.

As an example, a user can register an interest (a query) on all tweets that contain the word "elasticsearch". For every tweet, one can percolate the tweet against all registered user queries, and find out which ones matched.

Here is a quick sample, first, lets create a **test** index:

```
curl -XPUT localhost:9200/test
```

Next, we will register a percolator query with a specific name called **kuku** against the **test** index:

```
curl -XPUT localhost:9200/_percolator/test/kuku -d '{
    "query" : {
        "term" : {
            "field1" : "value1"
        }
    }
}'
```

And now, we can percolate a document and see which queries match on it (note, its not really indexed!):

```
curl -XGET localhost:9200/test/type1/_percolate -d '{
    "doc" : {
        "field1" : "value1"
    }
}'
```

And the matches are part of the response:

```
{"ok":true, "matches":["kuku"]}
```

You can unregister the previous percolator query with the same API you use to delete any document in an index:

```
curl -XDELETE localhost:9200/_percolator/test/kuku
```

Filtering Executed Queries

Since the registered percolator queries are just docs in an index, one can filter the queries that will be used to percolate a doc. For example, we can add a **color** field to the registered query:

```
curl -XPUT localhost:9200/_percolator/test/kuku -d '{
    "color" : "blue",
    "query" : {
        "term" : {
            "field1" : "value1"
        }
    }
}'
```

And then, we can percolate a doc that only matches on blue colors:

```
curl -XGET localhost:9200/test/type1/_percolate -d '{
    "doc" : {
        "field1" : "value1"
    },
    "query" : {
        "term" : {
            "color" : "blue"
        }
    }
}'
```

How it Works

The **_percolator** which holds the repository of registered queries is just a another index. The query is registered under a concrete index that exists (or will exist). That index name is represented as the type in the **_percolator** index (a bit confusing, I know...).

The fact that the queries are stored as docs in another index (**_percolator**) gives us both the persistency nature of it, and the ability to filter out queries to execute using another query.

The **_percolator** index uses the **index.auto_expand_replica** setting to make sure that each data node will have access locally to the registered queries, allowing for fast query executing to filter out queries to run against a percolated doc.

The percolate API uses the whole number of shards as percolating processing "engines", both primaries and replicas. In our above case, if the **test** index has 2 shards with 1 replica, 4 shards will round robing in handing percolate requests. (dynamically) increasing the number of replicas will increase the number of percolation power.

Note, percolate request will prefer to be executed locally, and will not try and round robin across shards if a shard exists locally on a node that received a request (for example, from HTTP). Its important to do some roundrobin in the client code among nodes (in any case its recommended). If this behavior is not desired, the **prefer_local** parameter can be set to **false** to disable it.

Search

The search API allows to execute a search query and get back search hits that match the query. It can be executed across *indices and types*. The query can either be provided using a simple *query string as a parameter*, or using a *request body*.

Routing

When executing a search, it will be broadcasted to all the index/indices shards (round robin between replicas). Which shards will be searched on can be controlled by providing the **routing** parameter. For example, when indexing tweets, the routing value can be the user name:

```
$ curl -XPOST 'http://localhost:9200/twitter/tweet?routing=kimchy' -d '{
    "user" : "kimchy",
    "postDate" : "2009-11-15T14:12:12",
    "message" : "trying out Elastic Search"
}
'
```

In such a case, if we want to search only on the tweets for a specific user, we can specify it as the routing, resulting in the search hitting only the relevant shard:

```
$ curl -XGET 'http://localhost:9200/twitter/tweet/_search?routing=kimchy' -d '{
    "query": {
        "filtered" : {
            "query" : {
                "query_string" : {
                    "query" : "some query string here"
                }
            },
            "filter" : {
                "term" : { "user" : "kimchy" }
            }
        }
    }
}
'
```

The routing parameter can be multi valued represented as a comma separated string. This will result in hitting the relevant shards where the routing values match to.

Stats Groups

A search can be associated with stats groups, which maintains a statistics aggregation per group. It can later be retrieved using the indices stats API specifically. For example, here is a search body request that associate the request with two different groups:

```
{
    "query" : {
        "match_all" : {}
    },
    "stats" : ["group1", "group2"]
}
```

Explain

Enables explanation for each hit on how its score was computed.

```
{
    "explain": true,
    "query" : {
        "term" : { "user" : "kimchy" }
    }
}
```

Facets

The usual purpose of a full-text search engine is to return a small number of documents matching your query.

Facets provide aggregated data based on a search query. In the simplest case, a *terms facet* can return **facet counts** for various **facet values** for a specific **field**. ElasticSearch supports more facet implementations, such as *statistical* or *date histogram* facets.

The field used for facet calculations **must** be of type numeric, date/time or be analyzed as a single token — see the *_Mapping_* guide for details on the analysis process.

You can give the facet a custom **name** and return multiple facets in one request.

Let's try it out with a simple example. Suppose we have a number of articles with a field called **tags**, preferably analyzed with the *keyword* analyzer. The facet aggregation will return counts for the most popular tags across the documents matching your query — or across all documents in the index.

We will store some example data first:

```
curl -X DELETE "http://localhost:9200/articles"
curl -X POST "http://localhost:9200/articles/article" -d '{"title" : "One",   "tags"␣
↪: ["foo"]}'
curl -X POST "http://localhost:9200/articles/article" -d '{"title" : "Two",   "tags"␣
↪: ["foo", "bar"]}'
curl -X POST "http://localhost:9200/articles/article" -d '{"title" : "Three", "tags"␣
↪: ["foo", "bar", "baz"]}'
```

Now, let's query the index for articles beginning with letter "T" and retrieve a *_terms facet_* for the **tags** field. We will name the facet simply: **tags**.

```
curl -X POST "http://localhost:9200/articles/_search?pretty=true" -d '
  {
    "query" : { "query_string" : {"query" : "T*"} },
    "facets" : {
      "tags" : { "terms" : {"field" : "tags"} }
    }
  }
'
```

This request will return articles "Two" and "Three" (because they match our query), as well as the **tags** facet:

```
"facets" : {
  "tags" : {
    "_type" : "terms",
    "missing" : 0,
    "total": 5,
    "other": 0,
    "terms" : [ {
      "term" : "foo",
      "count" : 2
    }, {
      "term" : "bar",
      "count" : 2
    }, {
      "term" : "baz",
      "count" : 1
    } ]
  }
}
```

In the **terms** array, relevant **terms** and **counts** are returned. You'll probably want to display these to your users. The facet also returns the number of documents which have no value for the field (**missing**), the number of facet values not included in the returned facets (**other**), and the total number of tokens in the facet (**total**).

Notice, that the counts are scoped to the current query: **foo** is counted only twice (not three times), **bar** is counted twice and **baz** once.

That's because the primary purpose of facets is to enable _faceted navigation_, allowing the user to refine her query based on the insight from the facet, ie. restrict the search to a specific category, price or date range. See the example of faceted navigation at **LinkedIn** below:

!/guide/images/linkedin-faceted-search.png(Faceted Search at LinkedIn)!

Facets can be used, however, for other purposes: computing histograms, statistical aggregations, and more.

Scope

As we have already mentioned, facet computation is restricted to the scope of the current query, called **main**, by default. Facets can be computed within the **global** scope as well, in which case it will return values computed acrosss all documents in the index:

```
{
    "facets" : {
        "<FACET NAME>" : {
            "<FACET TYPE>" : { ... },
            "global" : true
        }
    }
}
```

There's one **important distinction** to keep in mind. While search **queries** restrict both the returned documents and facet counts, search **filters** restrict only returned documents — but **not** facet counts.

If you need to restrict both the documents and facets, and you're not willing or able to use a query, you may use a **facet filter**.

Facet Filter

All facets can be configured with an additional filter (explained in the *Query DSL* section), which **will** reduce the documents they use for computing results. An example with a **term** filter:

```
{
    "facets" : {
        "<FACET NAME>" : {
            "<FACET TYPE>" : {
                ...
            },
            "facet_filter" : {
                "term" : { "user" : "kimchy"}
            }
        }
    }
}
```

Note that this is different from a facet of the *filter* type.

Facets with the nested types

Nested mapping allows for better support for "inner" documents faceting, especially when it comes to multi valued key and value facets (like histograms, or term stats).

What is it good for? First of all, this is the only way to use facets on nested documents once they are used (possibly for other reasons). But, there is also facet specific reason why nested documents can be used, and that's the fact that facets working on different key and value field (like term_stats, or histogram) can now support cases where both are multi valued properly.

For example, lets use the following mapping:

```
{
    "type1" : {
        "properties" : {
            "obj1" : {
                "type" : "nested"
            }
        }
    }
}
```

And, here is a sample data:

```
{
    "obj1" : [
        {
            "name" : "blue",
            "count" : 4
        },
        {
            "name" : "green",
            "count" : 6
        }
    ]
}
```

Nested Query Facets

Any **nested** query allows to specify a **_scope** associated with it. Any **facet** allows for a scope to be defined on it controlling the scope it will execute against. For example, the following **facet1** terms stats facet will only run on documents matching the nested query associated with **my_scope**:

```
{
    "query": {
        "nested": {
            "_scope": "my_scope",
            "path": "obj1",
            "score_mode": "avg",
            "query": {
                "bool": {
                    "must": [
                        {"text": {"obj1.name": "blue"}},
                        {"range": {"obj1.count": {"gt": 3}}}
                    ]
                }
```

```
            }
        }
    },
    "facets": {
        "facet1": {
            "terms_stats": {
                "key_field": "obj1.name",
                "value_field": "obj1.count"
            },
            "scope": "my_scope"
        }
    }
}
```

All Nested Matching Root Documents

Another option is to run the facet on all the nested documents matching the root objects that the main query will end up producing. For example:

```
{
    "query": {
        "match_all": {}
    },
    "facets": {
        "facet1": {
            "terms_stats": {
                "key_field" : "name",
                "value_field": "count"
            },
            "nested": "obj1"
        }
    }
}
```

The **nested** element provides the path to the nested document (can be a multi level nested docs) that will be used.

Date Histogram Facet

A specific histogram facet that can work with **date** field types enhancing it over the regular *histogram facet*. Here is a quick example:

```
{
    "query" : {
        "match_all" : {}
    },
    "facets" : {
        "histo1" : {
            "date_histogram" : {
                "field" : "field_name",
                "interval" : "day"
            }
        }
    }
}
```

Interval

The **interval** allows to set the interval at which buckets will be created for each hit. It allows for the constant values of **year**, **month**, **week**, **day**, **hour**, **minute**.

The specific constant values also support setting rounding by appending : to it, and then the rounding value. For example: day:ceiling. The values are:

- **floor**: (the default), rounds to the lowest whole unit of this field.
- **ceiling**: Rounds to the highest whole unit of this field.
- **half_floor**: Round to the nearest whole unit of this field. If the given millisecond value is closer to the floor or is exactly halfway, this function behaves like floor. If the millisecond value is closer to the ceiling, this function behaves like ceiling.
- **half_ceiling**: Round to the nearest whole unit of this field. If the given millisecond value is closer to the floor, this function behaves like floor. If the millisecond value is closer to the ceiling or is exactly halfway, this function behaves like ceiling.
- **half_even**: Round to the nearest whole unit of this field. If the given millisecond value is closer to the floor, this function behaves like floor. If the millisecond value is closer to the ceiling, this function behaves like ceiling. If the millisecond value is exactly halfway between the floor and ceiling, the ceiling is chosen over the floor only if it makes this field's value even.

It also support time setting like **1.5h** (up to **w** for weeks).

Time Zone

By default, times are stored as UTC milliseconds since the epoch. Thus, all computation and "bucketing" / "rounding" is done on UTC. It is possible to provide a time zone (both pre rounding, and post rounding) value, which will cause all computations to take the relevant zone into account. The time returned for each bucket/entry is milliseconds since the epoch of the provided time zone.

The parameters are **pre_zone** (pre rounding based on interval) and **post_zone** (post rounding based on interval). The **time_zone** parameter simply sets the **pre_zone** parameter. By default, those are set to **UTC**.

The zone value accepts either a numeric value for the hours offset, for example: **'time_zone" : -2**. It also accepts a format of hours and minutes, like **"time_zone" : "-02:30"**. Another option is to provide a time zone accepted as one of the values listed "here <http://joda-time.sourceforge.net/timezones.html>'_.

Lets take an example. For **2012-04-01T04:15:30Z**, with a **pre_zone** of **-08:00**. For **day** interval, the actual time by applying the time zone and rounding falls under **2012-03-31**, so the returned value will be (in millis) of **2012-03-31T00:00:00Z** (UTC). For **hour** interval, applying the time zone results in **2012-03-31T20:15:30**, rounding it results in **2012-03-31T20:00:00**, but, we want to return it in UTC (**post_zone** is not set), so we convert it back to UTC: **2012-04-01T04:00:00Z**. Note, we are consistent in the results, returning the rounded value in UTC.

post_zone simply takes the result, and adds the relevant offset.

Sometimes, we want to apply the same conversion to UTC we did above for **hour** also for **day** (and up) intervals. We can set **pre_zone_adjust_large_interval** to **true**, which will apply the same conversion done for **hour** interval in the example, to **day** and above intervals (it can be set regardless of the interval, but only kick in when using **day** and higher intervals).

Factor

The date histogram works on numeric values (since time is stored in milliseconds since the epoch in UTC). But, sometimes, systems will store a different resolution (like seconds since UTC) in a numeric field. The **factor** parameter

can be used to change the value in the field to milliseconds to actual do the relevant rounding, and then be applied again to get to the original unit. For example, when storing in a numeric field seconds resolution, the **factor** can be set to **1000**.

Pre / Post Offset

Specific offsets can be provided for pre rounding and post rounding. The **pre_offset** for pre rounding, and **post_offset** for post rounding. The format is the date time format (**1h**, **1d**, ...).

Value Field

The date_histogram facet allows to use a different key (of type date) which controls the bucketing, with a different value field which will then return the total and mean for that field values of the hits within the relevant bucket. For example:

```
{
    "query" : {
        "match_all" : {}
    },
    "facets" : {
        "histo1" : {
            "date_histogram" : {
                "key_field" : "timestamp",
                "value_field" : "price",
                "interval" : "day"
            }
        }
    }
}
```

Script Value Field

A script can be used to compute the value that will then be used to compute the total and mean for a bucket. For example:

```
{
    "query" : {
        "match_all" : {}
    },
    "facets" : {
        "histo1" : {
            "date_histogram" : {
                "key_field" : "timestamp",
                "value_script" : "doc['price'].value * 2",
                "interval" : "day"
            }
        }
    }
}
```

Filter Facet

A filter facet (not to be confused with a *facet filter* allows you to return a count of the hits matching the filter. The filter itself can be expressed using the *Query DSL*. For example:

```
{
    "facets" : {
        "wow_facet" : {
            "filter" : {
                "term" : { "tag" : "wow" }
            }
        }
    }
}
```

Note, filter facet filters are faster than query facet when using native filters (non query wrapper ones).

Geo Distance Facet

The geo_distance facet is a facet providing information for ranges of distances from a provided geo_point including count of the number of hits that fall within each range, and aggregation information (like total).

Assuming the following sample doc:

```
{
    "pin" : {
        "location" : {
            "lat" : 40.12,
            "lon" : -71.34
        }
    }
}
```

Here is an example that create a **geo_distance** facet from a **pin.location** of 40,-70, and a set of ranges:

```
{
    "query" : {
        "match_all" : {}
    },
    "facets" : {
        "geo1" : {
            "geo_distance" : {
                "pin.location" : {
                    "lat" : 40,
                    "lon" : -70
                },
                "ranges" : [
                    { "to" : 10 },
                    { "from" : 10, "to" : 20 },
                    { "from" : 20, "to" : 100 },
                    { "from" : 100 }
                ]
            }
        }
    }
}
```

Accepted Formats

In much the same way the geo_point type can accept different representation of the geo point, the filter can accept it as well:

Lat Lon As Properties

```
{
    "query" : {
        "match_all" : {}
    },
    "facets" : {
        "geo1" : {
            "geo_distance" : {
                "pin.location" : {
                    "lat" : 40,
                    "lon" : -70
                },
                "ranges" : [
                    { "to" : 10 },
                    { "from" : 10, "to" : 20 },
                    { "from" : 20, "to" : 100 },
                    { "from" : 100 }
                ]
            }
        }
    }
}
```

Lat Lon As Array

Format in **[lon, lat]**, note, the order of lon/lat here in order to conform with GeoJSON.

```
{
    "query" : {
        "match_all" : {}
    },
    "facets" : {
        "geo1" : {
            "geo_distance" : {
                "pin.location" : [40, -70],
                "ranges" : [
                    { "to" : 10 },
                    { "from" : 10, "to" : 20 },
                    { "from" : 20, "to" : 100 },
                    { "from" : 100 }
                ]
            }
        }
    }
}
```

Lat Lon As String

Format in **lat,lon**.

```
{
    "query" : {
        "match_all" : {}
    },
    "facets" : {
        "geo1" : {
            "geo_distance" : {
                "pin.location" : "40, -70",
                "ranges" : [
                    { "to" : 10 },
                    { "from" : 10, "to" : 20 },
                    { "from" : 20, "to" : 100 },
                    { "from" : 100 }
                ]
            }
        }
    }
}
```

Geohash

```
{
    "query" : {
        "match_all" : {}
    },
    "facets" : {
        "geo1" : {
            "geo_distance" : {
                "pin.location" : "drm3btev3e86",
                "ranges" : [
                    { "to" : 10 },
                    { "from" : 10, "to" : 20 },
                    { "from" : 20, "to" : 100 },
                    { "from" : 100 }
                ]
            }
        }
    }
}
```

Ranges

When a **to** or **from** are not set, they are assumed to be unbounded. Ranges are allowed to overlap, basically, each range is treated by itself.

Options

Option	Description
unit	The unit the ranges are provided in. Defaults to **km**. Can also be **mi** or **miles**.
distance_type	How to compute the distance. Can either be **arc** (better precision) or **plane** (faster). Defaults to **arc**.

Value Options

On top of the count of hits falling within each range, aggregated data can be provided (total) as well. By default, the aggregated data will simply use the distance calculated, but the value can be extracted either using a different numeric field, or a script. Here is an example of using a different numeric field:

```
{
    "query" : {
        "match_all" : {}
    },
    "facets" : {
        "geo1" : {
            "geo_distance" : {
                "pin.location" : "drm3btev3e86",
                "value_field" : "num1",
                "ranges" : [
                    { "to" : 10 },
                    { "from" : 10, "to" : 20 },
                    { "from" : 20, "to" : 100 },
                    { "from" : 100 }
                ]
            }
        }
    }
}
```

And here is an example of using a script:

```
{
    "query" : {
        "match_all" : {}
    },
    "facets" : {
        "geo1" : {
            "geo_distance" : {
                "pin.location" : "drm3btev3e86",
                "value_script" : "doc['num1'].value * factor",
                "params" : {
                    "factor" : 5
                }
                "ranges" : [
                    { "to" : 10 },
                    { "from" : 10, "to" : 20 },
                    { "from" : 20, "to" : 100 },
                    { "from" : 100 }
                ]
            }
        }
```

```
    }
}
```

Note the params option, allowing to pass parameters to the script (resulting in faster script execution instead of providing the values within the script each time).

geo_point Type

The facet *requires* the **geo_point** type to be set on the relevant field.

Multi Location Per Document

The facet can work with multiple locations per document.

Histogram Facet

The histogram facet works with numeric data by building a histogram across intervals of the field values. Each value is "rounded" into an interval (or placed in a bucket), and statistics are provided per interval/bucket (count and total). Here is a simple example:

```
{
    "query" : {
        "match_all" : {}
    },
    "facets" : {
        "histo1" : {
            "histogram" : {
                "field" : "field_name",
                "interval" : 100
            }
        }
    }
}
```

The above example will run a histogram facet on the **field_name** filed, with an **interval** of **100** (so, for example, a value of **1055** will be placed within the **1000** bucket).

The interval can also be provided as a time based interval (using the time format). This mainly make sense when working on date fields or field that represent absolute milliseconds, here is an example:

```
{
    "query" : {
        "match_all" : {}
    },
    "facets" : {
        "histo1" : {
            "histogram" : {
                "field" : "field_name",
                "time_interval" : "1.5h"
            }
        }
    }
}
```

Key and Value

The histogram facet allows to use a different key and value. The key is used to place the hit/document within the appropriate bucket, and the value is used to compute statistical data (for example, total). Here is an example:

```
{
    "query" : {
        "match_all" : {}
    },
    "facets" : {
        "histo1" : {
            "histogram" : {
                "key_field" : "key_field_name",
                "value_field" : "value_field_name",
                "interval" : 100
            }
        }
    }
}
```

Script Key and Value

Sometimes, some munging of both the key and the value are needed. In the key case, before it is rounded into a bucket, and for the value, when the statistical data is computed per bucket *scripts* can be used. Here is an example:

```
{
    "query" : {
        "match_all" : {}
    },
    "facets" : {
        "histo1" : {
            "histogram" : {
                "key_script" : "doc['date'].date.minuteOfHour",
                "value_script" : "doc['num1'].value",
            }
        }
    }
}
```

In the above sample, we can use a date type field called **date** to get the minute of hour from it, and the total will be computed based on another field **num1**. Note, in this case, no **interval** was provided, so the bucket will be based directly on the **key_script** (no rounding).

Parameters can also be provided to the different scripts (preferable if the script is the same, with different values for a specific parameter, like "factor"):

```
{
    "query" : {
        "match_all" : {}
    },
    "facets" : {
        "histo1" : {
            "histogram" : {
                "key_script" : "doc['date'].date.minuteOfHour * factor1",
                "value_script" : "doc['num1'].value + factor2",
                "params" : {
```

```
                    "factor1" : 2,
                    "factor2" : 3
                }
            }
        }
    }
}
```

Memory Considerations

In order to implement the histogram facet, the relevant field values are loaded into memory from the index. This means that per shard, there should be enough memory to contain them. Since by default, dynamic introduced types are **long** and **double**, one option to reduce the memory footprint is to explicitly set the types for the relevant fields to either **short**, **integer**, or **float** when possible.

Query Facet

A facet query allows to return a count of the hits matching the facet query. The query itself can be expressed using the Query DSL. For example:

```
{
    "facets" : {
        "wow_facet" : {
            "query" : {
                "term" : { "tag" : "wow" }
            }
        }
    }
}
```

Range Facet

range facet allow to specify a set of ranges and get both the number of docs (count) that fall within each range, and aggregated data either based on the field, or using another field. Here is a simple example:

```
{
    "query" : {
        "match_all" : {}
    },
    "facets" : {
        "range1" : {
            "range" : {
                "field" : "field_name",
                "ranges" : [
                    { "to" : 50 },
                    { "from" : 20, "to" : 70 },
                    { "from" : 70, "to" : 120 },
                    { "from" : 150 }
                ]
            }
        }
    }
}
```

Another option which is a bit more DSL enabled is to provide the ranges on the actual field name, for example:

```
{
    "query" : {
        "match_all" : {}
    },
    "facets" : {
        "range1" : {
            "range" : {
                "my_field" : [
                    { "to" : 50 },
                    { "from" : 20, "to" : 70 },
                    { "from" : 70, "to" : 120 },
                    { "from" : 150 }
                ]
            }
        }
    }
}
```

Key and Value

The **range** facet allow to use a different field to check if it doc falls within a range, and another field to compute aggregated data per range (like total). For example:

```
{
    "query" : {
        "match_all" : {}
    },
    "facets" : {
        "range1" : {
            "range" : {
                "key_field" : "field_name",
                "value_field" : "another_field_name",
                "ranges" : [
                    { "to" : 50 },
                    { "from" : 20, "to" : 70 },
                    { "from" : 70, "to" : 120 },
                    { "from" : 150 }
                ]
            }
        }
    }
}
```

Script Key and Value

Sometimes, some munging of both the key and the value are needed. In the key case, before it is checked if it falls within a range, and for the value, when the statistical data is computed per range scripts can be used. Here is an example:

```
{
    "query" : {
```

```
        "match_all" : {}
    },
    "facets" : {
        "range1" : {
            "range" : {
                "key_script" : "doc['date'].date.minuteOfHour",
                "value_script" : "doc['num1'].value",
                "ranges" : [
                    { "to" : 50 },
                    { "from" : 20, "to" : 70 },
                    { "from" : 70, "to" : 120 },
                    { "from" : 150 }
                ]
            }
        }
    }
}
```

Date Ranges

The range facet support also providing the range as string formatted dates.

Statistical Facet

Statistical facet allows to compute statistical data on a numeric fields. The statistical data include count, total, sum of squares, mean (average), minimum, maximum, variance, and standard deviation. Here is an example:

```
{
    "query" : {
        "match_all" : {}
    },
    "facets" : {
        "stat1" : {
            "statistical" : {
                "field" : "num1"
            }
        }
    }
}
```

Script field

When using **field**, the numeric value of the field is used to compute the statistical information. Sometimes, several fields values represent the statistics we want to compute, or some sort of mathematical evaluation. The script field allows to define a *script* to evaluate, with its value used to compute the statistical information. For example:

```
{
    "query" : {
        "match_all" : {}
    },
    "facets" : {
        "stat1" : {
            "statistical" : {
```

```
                "script" : "doc['num1'].value + doc['num2'].value"
            }
        }
    }
}
```

Parameters can also be provided to the different scripts (preferable if the script is the same, with different values for a specific parameter, like "factor"):

```
{
    "query" : {
        "match_all" : {}
    },
    "facets" : {
        "stat1" : {
            "statistical" : {
                "script" : "(doc['num1'].value + doc['num2'].value) * factor",
                "params" : {
                    "factor" : 5
                }
            }
        }
    }
}
```

Multi Field

The statistical facet can be executed against more than one field, returning the aggregation result across those fields. For example:

```
{
    "query" : {
        "match_all" : {}
    },
    "facets" : {
        "stat1" : {
            "statistical" : {
                "fields" : ["num1", "num2"]
            }
        }
    }
}
```

Memory Considerations

In order to implement the histogram facet, the relevant field values are loaded into memory from the index. This means that per shard, there should be enough memory to contain them. Since by default, dynamic introduced types are **long** and **double**, one option to reduce the memory footprint is to explicitly set the types for the relevant fields to either **short**, **integer**, or **float** when possible.

Terms Facet

Allow to specify field facets that return the N most frequent terms. For example:

```
{
    "query" : {
        "match_all" : {   }
    },
    "facets" : {
        "tag" : {
            "terms" : {
                "field" : "tag",
                "size" : 10
            }
        }
    }
}
```

Note It is preferred to have the terms facet executed on a non analyzed field, or a field without a large number of terms it breaks to.

Ordering

Allow to control the ordering of the terms facets, to be ordered by **count**, **term**, **reverse_count** or **reverse_term**. The default is **count**. Here is an example:

```
{
    "query" : {
        "match_all" : {   }
    },
    "facets" : {
        "tag" : {
            "terms" : {
                "field" : "tag",
                "size" : 10,
                "order" : "term"
            }
        }
    }
}
```

All Terms

Allow to get all the terms in the terms facet, ones that do not match a hit, will have a count of 0. Note, this should not be used with fields that have many terms.

```
{
    "query" : {
        "match_all" : {   }
    },
    "facets" : {
        "tag" : {
            "terms" : {
                "field" : "tag",
                "all_terms" : true
            }
        }
    }
}
```

Excluding Terms

It is possible to specify a set of terms that should be excluded from the terms facet request result:

```
{
    "query" : {
        "match_all" : { }
    },
    "facets" : {
        "tag" : {
            "terms" : {
                "field" : "tag",
                "exclude" : ["term1", "term2"]
            }
        }
    }
}
```

Regex Patterns

The terms API allows to define regex expression that will control which terms will be included in the faceted list, here is an example:

```
{
    "query" : {
        "match_all" : { }
    },
    "facets" : {
        "tag" : {
            "terms" : {
                "field" : "tag",
                "regex" : "_regex expression here_"
                "regex_flags" : "DOTALL"
            }
        }
    }
}
```

Check Java Pattern API for more details about **regex_flags** options.

Term Scripts

Allow to define a script for terms facet to process the actual term that will be used in the term facet collection, and also optionally control its inclusion or not.

The script can either return a boolean value, with **true** to include it in the facet collection, and **false** to exclude it from the facet collection.

Another option is for the script to return a **string** controlling the term that will be used to count against. The script execution will include the term variable which is the current field term used.

For example:

```
{
    "query" : {
        "match_all" : {   }
    },
    "facets" : {
        "tag" : {
            "terms" : {
                "field" : "tag",
                "size" : 10,
                "script" : "term + 'aaa'"
            }
        }
    }
}
```

And using the boolean feature:

```
{
    "query" : {
        "match_all" : {   }
    },
    "facets" : {
        "tag" : {
            "terms" : {
                "field" : "tag",
                "size" : 10,
                "script" : "term == 'aaa' ? true : false"
            }
        }
    }
}
```

Multi Fields

The term facet can be executed against more than one field, returning the aggregation result across those fields. For example:

```
{
    "query" : {
        "match_all" : {   }
    },
    "facets" : {
        "tag" : {
            "terms" : {
                "fields" : ["tag1", "tag2"],
                "size" : 10
            }
        }
    }
}
```

Script Field

A script that provides the actual terms that will be processed for a given doc. A **script_field** (or **script** which will be used when no **field** or **fields** are provided) can be set to provide it.

As an example, a search request (that is quite "heavy") can be executed and use either **_source** itself or **_fields** (for stored fields) without needing to load the terms to memory (at the expense of much slower execution of the search, and causing more IO load):

```
{
    "query" : {
        "match_all" : {   }
    },
    "facets" : {
        "my_facet" : {
            "terms" : {
                "script_field" : "_source.my_field",
                "size" : 10
            },
        }
    }
}
```

Or:

```
{
    "query" : {
        "match_all" : {   }
    },
    "facets" : {
        "my_facet" : {
            "terms" : {
                "script_field" : "_fields['my_field']",
                "size" : 10
            },
        }
    }
}
```

Note also, that the above will use the whole field value as a single term.

_index

The term facet allows to specify a special field name called **_index**. This will return a facet count of hits per **_index** the search was executed on (relevant when a search request spans more than one index).

Memory Considerations

Term facet causes the relevant field values to be loaded into memory. This means that per shard, there should be enough memory to contain them. It is advisable to explicitly set the fields to be **not_analyzed** or make sure the number of unique tokens a field can have is not large.

Terms Stats Facet

The **terms_stats** facet combines both the *terms* and *statistical* allowing to compute stats computed on a field, per term value driven by another field. For example:

```
{
    "query" : {
        "match_all" : {   }
    },
    "facets" : {
        "tag_price_stats" : {
            "terms_stats" : {
                "key_field" : "tag",
                "value_field" : "price"
            }
        }
    }
}
```

The **size** parameter controls how many facet entries will be returned. It defaults to **10**. Setting it to 0 will return all terms matching the hits (be careful not to return too many results).

Ordering is done by setting **order**, with possible values of **term**, **reverse_term**, **count**, **reverse_count**, **total**, **reverse_total**, **min**, **reverse_min**, **max**, **reverse_max**, **mean**, **reverse_mean**. Defaults to **count**.

The value computed can also be a script, using the **value_script** instead of **value_field**, in which case the **lang** can control its language, and **params** allow to provide custom parameters (as in other scripted components).

Note, the terms stats can work with mutli valued key fields, or multi valued value fields, but not when both are multi valued (as ordering is not maintained).

Fields

Allows to selectively load specific fields for each document represented by a search hit. Defaults to load the internal **_source** field.

```
{
    "fields" : ["user", "postDate"],
    "query" : {
        "term" : { "user" : "kimchy" }
    }
}
```

The fields will automatically load stored fields (**store** mapping set to **yes**), or, if not stored, will load the **_source** and extract it from it (allowing to return nested document object).

***** can be used to load all stored fields from the document.

An empty array will cause only the **_id** and **_type** for each hit to be returned, for example:

```
{
    "fields" : [],
    "query" : {
        "term" : { "user" : "kimchy" }
    }
}
```

Script fields can also be automatically detected and used as fields, so things like **_source.obj1.obj2** can be used, though not recommended, as **obj1.obj2** will work as well.

Partial

When loading data from **_source**, partial fields can be used to use wildcards to control what part of the **_source** will be loaded based on **include** and **explude** patterns. For example:

```
{
    "query" : {
        "match_all" : {}
    },
    "partial_fields" : {
        "partial1" : {
            "include" : "obj1.obj2.*",
        }
    }
}
```

And one that will also exclude **obj1.obj3**:

```
{
    "query" : {
        "match_all" : {}
    },
    "partial_fields" : {
        "partial1" : {
            "include" : "obj1.obj2.*",
            "exclude" : "obj1.obj3.*"
        }
    }
}
```

Both **include** and **exclude** support multiple patterns:

```
{
    "query" : {
        "match_all" : {}
    },
    "partial_fields" : {
        "partial1" : {
            "include" : ["obj1.obj2.*", "obj1.obj4.*"],
            "exclude" : "obj1.obj3.*"
        }
    }
}
```

Filter

When doing things like facet navigation, sometimes only the hits are needed to be filtered by the chosen facet, and all the facets should continue to be calculated based on the original query. The **filter** element within the search request can be used to accomplish it.

Note, this is different compared to creating a **filtered** query with the filter, since this will cause the facets to only process the filtered results.

For example, lets create two tweets, with two different tags:

```
curl -XPUT 'localhost:9200/twitter/tweet/1' -d '
{
    "message" : "something blue",
    "tag" : "blue"
}
'

curl -XPUT 'localhost:9200/twitter/tweet/2' -d '
{
    "message" : "something green",
    "tag" : "green"
}
'

curl -XPOST 'localhost:9200/_refresh'
```

We can now search for something, and have a terms facet.

```
curl -XPOST 'localhost:9200/twitter/_search?pretty=true' -d '
{
    "query" : {
        "term" : { "message" : "something" }
    },
    "facets" : {
        "tag" : {
            "terms" : { "field" : "tag" }
        }
    }
}
'
```

We get two hits, and the relevant facets with a count of 1 for both **green** and **blue**. Now, lets say the **green** facet is chosen, we can simply add a filter for it:

```
curl -XPOST 'localhost:9200/twitter/_search?pretty=true' -d '
{
    "query" : {
        "term" : { "message" : "something" }
    },
    "filter" : {
        "term" : { "tag" : "green" }
    },
    "facets" : {
        "tag" : {
            "terms" : { "field" : "tag" }
        }
    }
}
'
```

And now, we get only 1 hit back, but the facets remain the same.

Note, if additional filters is required on specific facets, they can be added as a **facet_filter** to the relevant facets.

From Size

Though can be set as request parameters, they can also be set within the search body. **from** defaults to **0**, and **size** defaults to **10**.

```
{
    "from" : 0, "size" : 10,
    "query" : {
        "term" : { "user" : "kimchy" }
    }
}
```

Highlighting

Allow to highlight search results on one or more fields. The implementation uses the either lucene **fast-vector-highlighter** or **highlighter**. The search request body:

```
{
    "query" : {...},
    "highlight" : {
        "fields" : {
            "content" : {}
        }
    }
}
```

In the above case, the **content** field will be highlighted for each search hit (there will be another element in each search hit, called **highlight**, which includes the highlighted fields and the highlighted fragments).

In order to perform highlighting, the actual content of the field is required. If the field in question is stored (has **store** set to **yes** in the mapping), it will be used, otherwise, the actual **_source** will be loaded and the relevant field will be extracted from it.

If no **term_vector** information is provided (by setting it to **with_positions_offsets** in the mapping), then the plain highlighter will be used. If it is provided, then the fast vector highlighter will be used. When term vectors are available, highlighting will be performed faster at the cost of bigger index size.

Here is an example of setting the **content** field to allow for highlighting using the fast vector highlighter on it (this will cause the index to be bigger):

```
{
    "type_name" : {
        "content" : {"term_vector" : "with_positions_offsets"}
    }
}
```

Highlighting Tags

By default, the highlighting will wrap highlighted text in **** and ****. This can be controlled by setting **pre_tags** and **post_tags**, for example:

```
{
    "query" : {...},
    "highlight" : {
        "pre_tags" : ["<tag1>", "<tag2>"],
```

```
        "post_tags" : ["</tag1>", "</tag2>"],
        "fields" : {
            "_all" : {}
        }
    }
}
```

There can be a single tag or more, and the "importance" is ordered. There are also built in "tag" schemas, with currently a single schema called **styled** with **pre_tags** of:

```
<em class="hlt1">, <em class="hlt2">, <em class="hlt3">,
<em class="hlt4">, <em class="hlt5">, <em class="hlt6">,
<em class="hlt7">, <em class="hlt8">, <em class="hlt9">,
<em class="hlt10">
```

And post tag of ****. If you think of more nice to have built in tag schemas, just send an email to the mailing list or open an issue. Here is an example of switching tag schemas:

```
{
    "query" : {...},
    "highlight" : {
        "tags_schema" : "styled",
        "fields" : {
            "content" : {}
        }
    }
}
```

An **encoder** parameter can be used to define how highlighted text will be encoded. It can be either **default** (no encoding) or **html** (will escape html, if you use html highlighting tags).

Highlighted Fragments

Each field highlighted can control the size of the highlighted fragment in characters (defaults to **100**), and the maximum number of fragments to return (defaults to **5**). For example:

```
{
    "query" : {...},
    "highlight" : {
        "fields" : {
            "content" : {"fragment_size" : 150, "number_of_fragments" : 3}
        }
    }
}
```

On top of this it is possible to specify that highlighted fragments are order by score:

```
{
    "query" : {...},
    "highlight" : {
        "order" : "score",
        "fields" : {
            "content" : {"fragment_size" : 150, "number_of_fragments" : 3}
        }
    }
}
```

Note the score of text fragment in this case is calculated by Lucene highlighting framework. For implementation details you can check **ScoreOrderFragmentsBuilder.java** class.

If the **number_of_fragments** value is set to 0 then no fragments are produced, instead the whole content of the field is returned, and of course it is highlighted. This can be very handy if short texts (like document title or address) need to be highlighted but no fragmentation is required. Note that **fragment_size** is ignored in this case.

```
{
    "query" : {...},
    "highlight" : {
        "fields" : {
            "_all" : {},
            "bio.title" : {"number_of_fragments" : 0}
        }
    }
}
```

When using **fast-vector-highlighter** one can use **fragment_offset** parameter to conrol the margin to start highlighting from.

Global Settings

Highlighting settings can be set on a global level and then overridden at the field level.

```
{
    "query" : {...},
    "highlight" : {
        "number_of_fragments" : 3,
        "fragment_size" : 150,
        "tag_schema" : "styled",
        "fields" : {
            "_all" : { "pre_tags" : ["<em>"], "post_tags" : ["</em>"] },
            "bio.title" : { "number_of_fragments" : 0 },
            "bio.author" : { "number_of_fragments" : 0 },
            "bio.content" : { "number_of_fragments" : 5, "order" : "score" }
        }
    }
}
```

Require Field Match

require_field_match can be set to **true** which will cause a field to be highlighted only if a query matched that field. false means that terms are highlighted on all requested fields regardless if the query matches specifically on them.

Boundary Characters

When highlighting a field that is mapped with term vectors, **boundary_chars** can be configured to define what constitutes a boundary for highlighting. Its a single string with each boundary character defined in it. It defaults to **.,!? tn**.

The **boundary_max_size** allows to control how far to look for boundary characters, and defaults to **20**.

Index Boost

Allows to configure different boost level per index when searching across more than one indices. This is very handy when hits coming from one index matter more than hits coming from another index (think social graph where each user has an index).

```
{
    "indices_boost" : {
        "index1" : 1.4,
        "index2" : 1.3
    }
}
```

Indices Types

The search API can be applied to multiple types within an index, and across multiple indices with support for the *multi index syntax*. For example, we can search on all documents across all types within the twitter index:

```
$ curl -XGET 'http://localhost:9200/twitter/_search?q=user:kimchy'
```

We can also search within specific types:

```
$ curl -XGET 'http://localhost:9200/twitter/tweet,user/_search?q=user:kimchy'
```

We can also search all tweets with a certain tag across several indices (for example, when each user has his own index):

```
$ curl -XGET 'http://localhost:9200/kimchy,elasticsearch/tweet/_search?q=tag:wow'
```

Or we can search all tweets across all available indices using **_all** placeholder:

```
$ curl - XGET 'http://localhost:9200/_all/tweet/_search?q=tag:wow'
```

Or even search across all indices and all types:

```
$ curl -XGET 'http://localhost:9200/_search?q=tag:wow'
```

Named Filters

Each filter can accept a _name in its top level definition, for example:

```
{
    "filtered" : {
        "query" : {
            "term" : { "name.first" : "shay" }
        },
        "filter" : {
            "terms" : {
                "name.last" : ["banon", "kimchy"],
                "_name" : "test"
            }
        }
    }
}
```

The search response will include for each hit the **matched_filters** it matched on (note, this feature make sense for **or / bool** filters).

Note, the query filter had to be enhanced in order to support this. In order to set a name, the **fquery** filter should be used, which wraps a query (just so there will be a place to set a name for it), for example:

```
{
    "filtered" : {
        "query" : {
            "term" : { "name.first" : "shay" }
        },
        "filter" : {
            "fquery" : {
                "query" : {
                    "term" : { "name.last" : "banon" }
                },
                "_name" : "test"
            }
        }
    }
}
```

Min Score

Allows to filter out documents based on a minimum score:

```
{
    "min_score": 0.5,
    "query" : {
        "term" : { "user" : "kimchy" }
    }
}
```

Note, most times, this does not make much sense, but is provided for advance use cases.

Query

The query element within the search request body allows to define a query using the *Query DSL*.

```
{
    "query" : {
        "term" : { "user" : "kimchy" }
    }
}
```

Preference

Controls a **preference** of which shard replicas to execute the search request on. By default, the operation is randomized between the each shard replicas.

The **preference** can be set to:

- **_primary**: The operation will go and be executed only on the primary shards.

- **_primary_first**: The operation will go and be executed on the primary shard, and if not available (failover), will execute on other shards.
- **_local**: The operation will prefer to be executed on a local allocated shard is possible.
- **_only_node:xyz**: Restricts the search to execute only on a node with the provided node id (**xyz** in this case).
- Custom (string) value: A custom value will be used to guarantee that the same shards will be used for the same custom value. This can help with "jumping values" when hitting different shards in different refresh states. A sample value can be something like the web session id, or the user name.

Request Body

The search request can be executed with a search DSL, which includes the *Query DSL*, within its body. Here is an example:

```
$ curl -XGET 'http://localhost:9200/twitter/tweet/_search' -d '{
    "query" : {
        "term" : { "user" : "kimchy" }
    }
}
'
```

And here is a sample response:

```
{
    "_shards":{
        "total" : 5,
        "successful" : 5,
        "failed" : 0
    },
    "hits":{
        "total" : 1,
        "hits" : [
            {
                "_index" : "twitter",
                "_type" : "tweet",
                "_id" : "1",
                "_source" : {
                    "user" : "kimchy",
                    "postDate" : "2009-11-15T14:12:12",
                    "message" : "trying out Elastic Search"
                }
            }
        ]
    }
}
```

Parameters

Name	Description
time-out	A search timeout, bounding the search request to be executed within the specified time value and bail with the hits accumulated up to that point when expired. Defaults to no timeout.
from	The starting from index of the hits to return. Defaults to **0**.
size	The number of hits to return. Defaults to **10**.
search_type	The type of the search operation to perform. Can be **dfs_query_then_fetch**, **dfs_query_and_fetch**, **query_then_fetch**, **query_and_fetch**. Defaults to **query_then_fetch**.

Out of the above, the **search_type** is the one that can not be passed within the search request body, and in order to set it, it must be passed as a request REST parameter.

The rest of the search request should be passed within the body itself. The body content can also be passed as a REST parameter named **source**.

Note Both HTTP GET and HTTP POST can be used to execute search with body. Since not all clients support GET with body, POST is allowed as well.

Script Fields

Allows to return a *script evaluation* (based on different fields) for each hit, for example:

```
{
    "query" : {
        ...
    },
    "script_fields" : {
        "test1" : {
            "script" : "doc['my_field_name'].value * 2"
        },
        "test2" : {
            "script" : "doc['my_field_name'].value * factor",
            "params" : {
                "factor"  : 2.0
            }
        }
    }
}
```

Script fields can work on fields that are not store (**my_field_name** in the above case), and allow to return custom values to be returned (the evaluated value of the script).

Script fields can also access the actual **_source** document indexed and extract specific elements to be returned from it (can be an "object" type). Here is an example:

```
{
    "query" : {
        ...
    },
    "script_fields" : {
        "test1" : {
            "script" : "_source.obj1.obj2"
        }
    }
}
```

Note the **_source** keyword here to navigate the json like model.

Its important to understand the difference between **doc['my_field'].value** and **_source.my_field**. The first, using the doc keyword, will cause the terms for that field to be loaded to memory (cached), which will result in faster execution, but more memory consumption. Also, the **doc[...]** notation only allows for simple valued fields (can't return a json object from it) and make sense only on non analyzed or single term based fields.

The **_source** on the other hand causes the source to be loaded, parsed, and then only the relevant part of the json is returned.

Scroll

A search request can be scrolled by specifying the **scroll** parameter. The **scroll** parameter is a time value parameter (for example: **scroll=5m**), indicating for how long the nodes that participate in the search will maintain relevant resources in order to continue and support it. This is very similar in its idea to opening a cursor against a database.

A **scroll_id** is returned from the first search request (and from continuous) scroll requests. The **scroll_id** should be used when scrolling (along with the **scroll** parameter, to stop the scroll from expiring). The scroll id can also be passed as part of the search request body.

Note: the **scroll_id** changes for each scroll request and only the most recent one should be used.

```
$ curl -XGET 'http://localhost:9200/twitter/tweet/_search?scroll=5m' -d '{
    "query": {
        "query_string" : {
            "query" : "some query string here"
        }
    }
}
'
```

```
$ curl -XGET 'http://localhost:9200/_search/scroll?scroll=5m&scroll_
↪id=c2Nhbjs2OzM0NDg1ODpzRlBLc0FXNlNyNm5JWUc1'
```

Note Scrolling is not intended for real time user requests, it is intended for cases like scrolling over large portions of data that exists within elasticsearch to reindex it for example.

For more information on scrolling, see the *scan* search type.

Search Type

There are different execution paths that can be done when executing a distributed search. The distributed search operation needs to be scattered to all the relevant shards and then all the results are gathered back. When doing scatter/gather type execution, there are several ways to do that, specifically with search engines.

One of the questions when executing a distributed search is how much results to retrieve from each shard. For example, if we have 10 shards, the 1st shard might hold the most relevant results from 0 till 10, with other shards results ranking below it. For this reason, when executing a request, we will need to get results from 0 till 10 from all shards, sort them, and then return the results if we want to insure correct results.

Another question, which relates to search engine, is the fact that each shard stands on its own. When a query is executed on a specific shard, it does not take into account term frequencies and other search engine information from the other shards. If we want to support accurate ranking, we would need to first execute the query against all shards and gather the relevant term frequencies, and then, based on it, execute the query.

Also, because of the need to sort the results, getting back a large document set, or even scrolling it, while maintaing the correct sorting behavior can be a very expensive operation. For large result set scrolling without sorting, the **scan** search type (explained below) is also available.

ElasticSearch is very flexible and allows to control the type of search to execute on a *per search request* basis. The type can be configured by setting the *search_type* parameter in the query string. The types are:

Query And Fetch

Parameter value: *query_and_fetch*.

The most naive (and possibly fastest) implementation is to simply execute the query on all relevant shards and return the results. Each shard returns **size** results. Since each shard already returns **size** hits, this type actually returns **size** times **number of shards** results back to the caller.

Query Then Fetch

Parameter value: *query_then_fetch*.

The query is executed against all shards, but only enough information is returned (*not the document content*). The results are then sorted and ranked, and based on it, *only the relevant shards* are asked for the actual document content. The return number of hits is exactly as specified in **size**, since they are the only ones that are fetched. This is very handy when the index has a lot of shards (not replicas, shard id groups).

Dfs, Query And Fetch

Parameter value: *dfs_query_and_fetch*.

Same as "Query And Fetch", except for an initial scatter phase which goes and computes the distributed term frequencies for more accurate scoring.

Dfs, Query Then Fetch

Parameter value: *dfs_query_then_fetch*.

Same as "Query Then Fetch", except for an initial scatter phase which goes and computes the distributed term frequencies for more accurate scoring.

Count

Parameter value: *count*.

A special search type that returns the count that matched the search request without any docs (represented in **total_hits**), and possibly, including facets as well. In general, this is preferable to the **count** API as it provides more options.

Scan

Parameter value: *scan*.

The **scan** search type allows to efficiently scroll a large result set. Its used first by executing a search request with scrolling and a query:

```
curl -XGET 'localhost:9200/_search?search_type=scan&scroll=10m&size=50' -d '
{
    "query" : {
        "match_all" : {}
    }
}
'
```

The **scroll** parameter control the keep alive time of the scrolling request and initiates the scrolling process. The timeout applies per round trip (i.e. between the previous scan scroll request, to the next).

The response will include no hits, with two important results, the **total_hits** will include the total hits that match the query, and the **scroll_id** that allows to start the scroll process. From this stage, the **_search/scroll** endpoint should be used to scroll the hits, feeding the next scroll request with the previous search result **scroll_id**. For example:

```
curl -XGET 'localhost:9200/_search/scroll?scroll=10m' -d
↪'c2NhbjsxOjBLMzdpWEtqU2IyZHlmVURPeFJOZnc7MzowSzM3aVhLalNiMmR5ZlVET3hSTmZ3OzU6MEszN2lYS2pTYjJkeWZVR
↪'
```

Scroll requests will include a number of hits equal to the size multiplied by the number of primary shards.

The "breaking" condition out of a scroll is when no hits has been returned. The total_hits will be maintained between scroll requests.

Note, scan search type does not support sorting (either on score or a field) or faceting.

Sort

Allows to add one or more sort on specific fields. Each sort can be reversed as well. The sort is defined on a per field level, with special field name for **_score** to sort by score.

```
{
    "sort" : [
        { "post_date" : {"order" : "asc"} },
        "user",
        { "name" : "desc" },
        { "age" : "desc" },
        "_score"
    ],
    "query" : {
        "term" : { "user" : "kimchy" }
    }
}
```

If the JSON parser support ordering without an array, the sort request can also be structured as follows:

```
{
    "sort" : {
        { "post_date" : {"order" : "asc"} },
        "user" : { },
        "_score" : { }
    },
    "query" : {
        "term" : { "user" : "kimchy" }
    }
}
```

Sort Values

The sort values for each document returned are also returned as part of the response.

Missing Values

Numeric fields support specific handling for missing fields in a doc. The **missing** value can be **_last**, **_first**, or a custom value (that will be used for missing docs as the sort value). For example:

```
{
    "sort" : [
        { "price" : {"missing" : "_last"} },
    ],
    "query" : {
        "term" : { "user" : "kimchy" }
    }
}
```

Ignoring Unmapped Fields

By default, the search request will fail if there is no mapping associated with a field. The **ignore_unmapped** option allows to ignore fields that have no mapping and not sort by them. Here is an example of how it can be used:

```
{
    "sort" : [
        { "price" : {"ignore_unmapped" : true} },
    ],
    "query" : {
        "term" : { "user" : "kimchy" }
    }
}
```

Geo Distance Sorting

Allow to sort by **_geo_distance**. Here is an example:

```
{
    "sort" : [
        {
            "_geo_distance" : {
                "pin.location" : [-70, 40],
                "order" : "asc",
                "unit" : "km"
            }
        }
    ],
    "query" : {
        "term" : { "user" : "kimchy" }
    }
}
```

The following formats are supported in providing the coordinates:

Lat Lon as Properties

```
{
    "sort" : [
        {
            "_geo_distance" : {
                "pin.location" : {
                    "lat" : 40,
                    "lon", -70
                }
                "order" : "asc",
                "unit" : "km"
            }
        }
    ],
    "query" : {
        "term" : { "user" : "kimchy" }
    }
}
```

Lat Lon as String

Format in **lat,lon**.

```
{
    "sort" : [
        {
            "_geo_distance" : {
                "pin.location" : "-70,40",
                "order" : "asc",
                "unit" : "km"
            }
        }
    ],
    "query" : {
        "term" : { "user" : "kimchy" }
    }
}
```

Geohash

```
{
    "sort" : [
        {
            "_geo_distance" : {
                "pin.location" : "drm3btev3e86",
                "order" : "asc",
                "unit" : "km"
            }
        }
    ],
    "query" : {
        "term" : { "user" : "kimchy" }
```

```
    }
}
```

Lat Lon as Array

Format in **[lon, lat]**, note, the order of lon/lat here in order to conform with GeoJSON.

```
{
    "sort" : [
        {
            "_geo_distance" : {
                "pin.location" : [-70, 40],
                "order" : "asc",
                "unit" : "km"
            }
        }
    ],
    "query" : {
        "term" : { "user" : "kimchy" }
    }
}
```

Script Based Sorting

Allow to sort based on custom scripts, here is an example:

```
{
    "query" : {
        ....
    },
    "sort" : {
        "_script" : {
            "script" : "doc['field_name'].value * factor",
            "type" : "number",
            "params" : {
                "factor" : 1.1
            },
            "order" : "asc"
        }
    }
}
```

Note, it is recommended, for single custom based script based sorting, to use **custom_score** query instead as sorting based on score is faster.

Track Scores

When sorting on a field, scores are not computed. By setting **track_scores** to true, scores will still be computed and tracked.

```
{
    "track_scores": true,
    "sort" : [
```

```
        { "post_date" : {"reverse" : true} },
        { "name" : "desc" },
        { "age" : "desc" }
    ],
    "query" : {
        "term" : { "user" : "kimchy" }
    }
}
```

Memory Considerations

When sorting, the relevant sorted field values are loaded into memory. This means that per shard, there should be enough memory to contain them. For string based types, the field sorted on should not be analyzed / tokenized. For numeric types, if possible, it is recommended to explicitly set the type to six_hun types (like **short**, **integer** and **float**).

Uri Request

A search request can be executed purely using a URI by providing request parameters. Not all search options are exposed when executing a search using this mode, but it can be handy for quick "curl tests". Here is an example:

```
$ curl -XGET 'http://localhost:9200/twitter/tweet/_search?q=user:kimchy'
```

And here is a sample response:

```
{
    "_shards":{
        "total" : 5,
        "successful" : 5,
        "failed" : 0
    },
    "hits":{
        "total" : 1,
        "hits" : [
            {
                "_index" : "twitter",
                "_type" : "tweet",
                "_id" : "1",
                "_source" : {
                    "user" : "kimchy",
                    "postDate" : "2009-11-15T14:12:12",
                    "message" : "trying out Elastic Search"
                }
            }
        ]
    }
}
```

Parameters

The parameters allowed in the URI are:

Name	Description
q	The query string (maps to the **query_string** query).
df	The default field to use when no field prefix is defined within the query.
analyzer	The analyzer name to be used when analyzing the query string.
default_operator	The default operator to be used, can be **AND** or **OR**. Defaults to **OR**.
explain	For each hit, contain an explanation of how scoring of the hits was computed.
fields	The selective fields of the document to return for each hit (either retried from the index if stored, or from the **_source** if not), comma delimited. Defaults to the internal **_source** field. Not specifying any value will cause no fields to return.
sort	Sorting to perform. Can either be in the form of **fieldName**, or **fieldName:asc/fieldName:desc**. The fieldName can either be an actual field within the document, or the special **_score** name to indicate sorting based on scores. There can be several **sort** parameters (order is important).
track_scores	When sorting, set to **true** in order to still track scores and return them as part of each hit.
timeout	A search timeout, bounding the search request to be executed within the specified time value and bail with the hits accumulated up to that point when expired. Defaults to no timeout.
from	The starting from index of the hits to return. Defaults to **0**.
size	The number of hits to return. Defaults to **10**.
search_type	The type of the search operation to perform. Can be **dfs_query_then_fetch**, **dfs_query_and_fetch**, **query_then_fetch**, **query_and_fetch**. Defaults to **query_then_fetch**.
lowercase_expanded_terms	Should terms be automatically lowercased or not. Defaults to **true**.
analyze_wildcard	Should wildcard and prefix queries be analyzed or not. Defaults to **false**.

Version

Returns a version for each search hit.

```
{
    "version": true,
    "query" : {
        "term" : { "user" : "kimchy" }
    }
}
```

Update

The update API allows to update a document based on a script provided. The operation gets the document (collocated with the shard) from the index, runs the script (with optional script language and parameters), and index back the result (also allows to delete, or ignore the operation). It uses versioning to make sure no updates have happened during the "get" and "reindex". (available from **0.19** onwards).

Note, this operation still means full reindex of the document, it just removes some network roundtrips and reduces chances of version conflicts between the get and the index. The **_source** field need to be enabled for this feature to work.

For example, lets index a simple doc:

```
curl -XPUT localhost:9200/test/type1/1 -d '{
    "counter" : 1,
    "tags" : ["red"]
}'
```

Now, we can execute a script that would increment the counter:

```
curl -XPOST 'localhost:9200/test/type1/1/_update' -d '{
    "script" : "ctx._source.counter += count",
    "params" : {
        "count" : 4
    }
}'
```

We can also add a tag to the list of tags (note, if the tag exists, it will still add it, since its a list):

```
curl -XPOST 'localhost:9200/test/type1/1/_update' -d '{
    "script" : "ctx._source.tags += tag",
    "params" : {
        "tag" : "blue"
    }
}'
```

We can also add a new field to the document:

```
curl -XPOST 'localhost:9200/test/type1/1/_update' -d '{
    "script" : "ctx._source.text = \"some text\""
}'
```

We can also remove a field from the document:

```
curl -XPOST 'localhost:9200/test/type1/1/_update' -d '{
    "script" : "ctx._source.remove(\"text\")"
}'
```

And, we can delete the doc if the tags contain blue, or ignore (noop):

```
curl -XPOST 'localhost:9200/test/type1/1/_update' -d '{
    "script" : "ctx._source.tags.contains(tag) ? ctx.op = \"delete\" : ctx.op = \
↪"none\"",
    "params" : {
        "tag" : "blue"
    }
}'
```

The update operation supports similar parameters as the index API, including:

- **routing**: Sets the routing that will be used to route the document to the relevant shard.
- **parent**: Simply sets the routing.
- **timeout**: Timeout waiting for a shard to become available.
- **replication**: The replication type for the delete/index operation (sync or async).
- **consistency**: The write consistency of the index/delete operation.
- **percolate**: Enables percolation and filters out which percolator queries will be executed.
- **refresh**: Refresh the index immediately after the operation occurs, so that the updated document appears in search results immediately.

And also support **retry_on_conflict** which controls how many times to retry if there is a version conflict between getting the document and indexing / deleting it. Defaults to **0**.

It also allows to update the **ttl** of a document using **ctx._ttl** and timestamp using **ctx._timestamp**. Note that if the timestamp is not updated and not extracted from the **_source** it will be set to the update date.

Validate

The validate API allows a user to validate a potentially expensive query without executing it. The following example shows how it can be used:

```
curl -XPUT 'http://localhost:9200/twitter/tweet/1' -d '{
    "user" : "kimchy",
    "post_date" : "2009-11-15T14:12:12",
    "message" : "trying out Elastic Search"
}'
```

When the query is valid, the response contains **valid:true**:

```
curl -XGET 'http://localhost:9200/twitter/_validate/query?q=user:foo'
{"valid":true,"_shards":{"total":1,"successful":1,"failed":0}}
```

Or, with a request body:

```
curl -XGET 'http://localhost:9200/twitter/tweet/_validate/query' -d '{
  "filtered" : {
    "query" : {
      "query_string" : {
        "query" : "*:*"
      }
    },
    "filter" : {
      "term" : { "user" : "kimchy" }
    }
  }
}'
{"valid":true,"_shards":{"total":1,"successful":1,"failed":0}}
```

If the query is invalid, **valid** will be **false**. Here the query is invalid because ElasticSearch knows the post_date field should be a date due to dynamic mapping, and 'foo' does not correctly parse into a date:

```
curl -XGET 'http://localhost:9200/twitter/tweet/_validate/query?q=post_date:foo'
{"valid":false,"_shards":{"total":1,"successful":1,"failed":0}}
```

An **explain** parameter can be specified to get more detailed information about why a query failed:

```
curl -XGET 'http://localhost:9200/twitter/tweet/_validate/query?q=post_date:foo&
↪pretty=true&explain=true'
{
  "valid" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "failed" : 0
  },
  "explanations" : [ {
    "index" : "twitter",
    "valid" : false,
    "error" : "org.elasticsearch.index.query.QueryParsingException: [twitter] Failed␣
↪to parse; org.elasticsearch.ElasticSearchParseException: failed to parse date field␣
↪[foo], tried both date format [dateOptionalTime], and timestamp number; java.lang.
↪IllegalArgumentException: Invalid format: \"foo\""
  } ]
}
```

Query Dsl

elasticsearch provides a full Query DSL based on JSON to define queries. In general, there are basic queries such as *term* or *prefix*. There are also compound queries like the *bool* query. Queries can also have filters associated with them such as the *filtered* or *constant_score* queries, with specific filter queries.

Think of the Query DSL as an AST of queries. Certain queries can contain other queries (like the *bool* query), other can contain filters (like the *constant_score*, and some can contain both a query and a filter (like the *filtered*. Each of those can contain *any* query of the list of queries or *any* filter from the list of filters, resulting in the ability to build quite complex (and interesting) queries.

Both queries and filters can be used in different APIs. For example, within a *search query*, or as a *facet filter*. This section explains the components (queries and filters) that can form the AST one can use.

Note Filters are very handy since they perform an order of magnitude better then plain queries since no scoring is performed and they are automatically cached.

Filters and Caching

Filters can be a great candidate for caching. Caching the result of a filter does not require a lot of memory, and will cause other queries executing against the same filter (same parameters) to be blazingly fast.

Some filters already produce a result that is easily cacheable, and the difference between caching and not caching them is the act of placing the result in the cache or not. These filters, which include the *term*, *terms*, *prefix*, and *range* filters, are by default cached and are recommended to use (compared to the equivalent query version) when the same filter (same parameters) will be used across multiple different queries (for example, a range filter with age higher than 10).

Other filters, usually already working with the field data loaded into memory, are not cached by default. Those filters are already very fast, and the process of caching them requires extra processing in order to allow the filter result to be used with different queries than the one executed. These filters, including the geo, *numeric_range*, and *script* filters are not cached by default.

The last type of filters are those working with other filters. The *and*, *not* and *or* filters are not cached as they basically just manipulate the internal filters.

All filters allow to set **_cache** element on them to explicitly control caching. They also allow to set **_cache_key** which will be used as the caching key for that filter. This can be handy when using very large filters (like a terms filter with many elements in it).

Query:

Bool Query

A query that matches documents matching boolean combinations of other queries. The bool query maps to Lucene **BooleanQuery**. It is built using one or more boolean clauses, each clause with a typed occurrence. The occurrence types are:

Occur	Description
must	The clause (query) must appear in matching documents.
should	The clause (query) should appear in the matching document. A boolean query with no **must** clauses, one or more **should** clauses must match a document. The minimum number of should clauses to match can be set using **minimum_number_should_match** parameter.
must_not	The clause (query) must not appear in the matching documents. Note that it is not possible to search on documents that only consists of a **must_not** clauses.

The bool query also supports **disable_coord** parameter (defaults to **false**).

```
{
    "bool" : {
        "must" : {
            "term" : { "user" : "kimchy" }
        },
        "must_not" : {
            "range" : {
                "age" : { "from" : 10, "to" : 20 }
            }
        },
        "should" : [
            {
                "term" : { "tag" : "wow" }
            },
            {
                "term" : { "tag" : "elasticsearch" }
            }
        ],
        "minimum_number_should_match" : 1,
        "boost" : 1.0
    }
}
```

Boosting Query

The **boosting** query can be used to effectively demote results that match a given query. Unlike the "NOT" clause in bool query, this still selects documents that contain undesirable terms, but reduces their overall score.

```
{
    "boosting" : {
        "positive" : {
            "term" : {
                "field1" : "value1"
            }
        },
        "negative" : {
            "term" : {
                "field2" : "value2"
            }
        },
        "negative_boost" : 0.2
    }
}
```

Constant Score Query

A query that wraps a filter or another query and simply returns a constant score equal to the query boost for every document in the filter. Maps to Lucene **ConstantScoreQuery**.

```
{
    "constant_score" : {
        "filter" : {
            "term" : { "user" : "kimchy"}
        },
        "boost" : 1.2
```

```
    }
}
```

The filter object can hold only filter elements, not queries. Filters can be much faster compared to queries since they don't perform any scoring, especially when they are cached.

A query can also be wrapped in a **constant_score** query:

```
{
    "constant_score" : {
        "query" : {
            "term" : { "user" : "kimchy"}
        },
        "boost" : 1.2
    }
}
```

Custom Boost Factor Query

custom_boost_factor query allows to wrap another query and multiply its score by the provided **boost_factor**. This can sometimes be desired since **boost** value set on specific queries gets normalized, while this query boost factor does not.

```
"custom_boost_factor" : {
    "query" : {
        ....
    },
    "boost_factor" : 5.2
}
```

Custom Score Query

custom_score query allows to wrap another query and customize the scoring of it optionally with a computation derived from other field values in the doc (numeric ones) using *script expression*. Here is a simple sample:

```
"custom_score" : {
    "query" : {
        ....
    },
    "script" : "_score * doc['my_numeric_field'].value"
}
```

On top of the different scripting field values and expression, the **_score** script parameter can be used to retrieve the score based on the wrapped query.

Script Parameters

Scripts are cached for faster execution. If the script has parameters that it needs to take into account, it is preferable to use the same script, and provide parameters to it:

```
"custom_score" : {
    "query" : {
        ....
```

```
    },
    "params" : {
        "param1" : 2,
        "param2" : 3.1
    },
    "script" : "_score * doc['my_numeric_field'].value / pow(param1, param2)"
}
```

Dis Max Query

A query that generates the union of documents produced by its subqueries, and that scores each document with the maximum score for that document as produced by any subquery, plus a tie breaking increment for any additional matching subqueries.

This is useful when searching for a word in multiple fields with different boost factors (so that the fields cannot be combined equivalently into a single search field). We want the primary score to be the one associated with the highest boost, not the sum of the field scores (as Boolean Query would give). If the query is "albino elephant" this ensures that "albino" matching one field and "elephant" matching another gets a higher score than "albino" matching both fields. To get this result, use both Boolean Query and DisjunctionMax Query: for each term a DisjunctionMaxQuery searches for it in each field, while the set of these DisjunctionMaxQuery's is combined into a BooleanQuery.

The tie breaker capability allows results that include the same term in multiple fields to be judged better than results that include this term in only the best of those multiple fields, without confusing this with the better case of two different terms in the multiple fields.The default **tie_breaker** is **0.0**.

This query maps to Lucene **DisjunctionMaxQuery**.

```
{
    "dis_max" : {
        "tie_breaker" : 0.7,
        "boost" : 1.2,
        "queries" : [
            {
                "term" : { "age" : 34 }
            },
            {
                "term" : { "age" : 35 }
            }
        ]
    }
}
```

Field Query

A query that executes a query string against a specific field. It is a simplified version of *query_string* query (by setting the **default_field** to the field this query executed against). In its simplest form:

```
{
    "field" : {
        "name.first" : "+something -else"
    }
}
```

Most of the **query_string** parameters are allowed with the **field** query as well, in such a case, the query should be formatted as follows:

```
{
    "field" : {
        "name.first" : {
            "query" : "+something -else",
            "boost" : 2.0,
            "enable_position_increments": false
        }
    }
}
```

Filtered Query

A query that applies a filter to the results of another query. This query maps to Lucene **FilteredQuery**.

```
{
    "filtered" : {
        "query" : {
            "term" : { "tag" : "wow" }
        },
        "filter" : {
            "range" : {
                "age" : { "from" : 10, "to" : 20 }
            }
        }
    }
}
```

The filter object can hold only filter elements, not queries. Filters can be much faster compared to queries since they don't perform any scoring, especially when they are cached.

Flt Field Query

The **fuzzy_like_this_field** query is the same as the **fuzzy_like_this** query, except that it runs against a single field. It provides nicer query DSL over the generic **fuzzy_like_this** query, and support typed fields query (automatically wraps typed fields with type filter to match only on the specific type).

```
{
    "fuzzy_like_this_field" : {
        "name.first" : {
            "like_text" : "text like this one",
            "max_query_terms" : 12
        }
    }
}
```

Note **fuzzy_like_this_field** can be shortened to **flt_field**.

The **fuzzy_like_this_field** top level parameters include:

Parameter	Description
like_text	The text to find documents like it, *required*.
ignore_tf	Should term frequency be ignored. Defaults to **false**.
max_query_terms	The maximum number of query terms that will be included in any generated query. Defaults to **25**.
min_similarity	The minimum similarity of the term variants. Defaults to **0.5**.
prefix_length	Length of required common prefix on variant terms. Defaults to **0**.
boost	Sets the boost value of the query. Defaults to **1.0**.
analyzer	The analyzer that will be used to analyze the text. Defaults to the analyzer associated with the field.

Flt Query

Fuzzy like this query find documents that are "like" provided text by running it against one or more fields.

```
{
    "fuzzy_like_this" : {
        "fields" : ["name.first", "name.last"],
        "like_text" : "text like this one",
        "max_query_terms" : 12
    }
}
```

Note **fuzzy_like_this** can be shortened to **flt**.

The **fuzzy_like_this** top level parameters include:

Parameter	Description
fields	A list of the fields to run the more like this query against. Defaults to the **_all** field.
like_text	The text to find documents like it, *required*.
ignore_tf	Should term frequency be ignored. Defaults to **false**.
max_query_terms	The maximum number of query terms that will be included in any generated query. Defaults to **25**.
min_similarity	The minimum similarity of the term variants. Defaults to **0.5**.
prefix_length	Length of required common prefix on variant terms. Defaults to **0**.
boost	Sets the boost value of the query. Defaults to **1.0**.
analyzer	The analyzer that will be used to analyze the text. Defaults to the analyzer associated with the field.

How it Works

Fuzzifies ALL terms provided as strings and then picks the best n differentiating terms. In effect this mixes the behaviour of FuzzyQuery and MoreLikeThis but with special consideration of fuzzy scoring factors. This generally produces good results for queries where users may provide details in a number offields and have no knowledge of boolean query syntax and also want a degree of fuzzy matching and a fast query.

For each source term the fuzzy variants are held in a BooleanQuery with no coord factor (because we are not looking for matches on multiple variants in any one doc). Additionally, a specialized TermQuery is used for variants and does not use that variant term's IDF because this would favour rarer terms eg misspellings. Instead, all variants use the same IDF ranking (the one for the source query term) and this is factored into the variant's boost. If the source query term does not exist in the index the average IDF of the variants is used.

Fuzzy Query

A fuzzy based query that uses similarity based on Levenshtein (edit distance) algorithm.

Note Warning: this query is not very scalable with its default prefix length of 0 - in this case, *every* term will be enumerated and cause an edit score calculation or **max_expansions** is not set.

Here is a simple example:

```
{
    "fuzzy" : { "user" : "ki" }
}
```

More complex settings can be set (the values here are the default values):

```
{
    "fuzzy" : {
        "user" : {
            "value" : "ki",
            "boost" : 1.0,
            "min_similarity" : 0.5,
            "prefix_length" : 0
        }
    }
}
```

The **max_expansions** parameter (unbounded by default) controls the number of terms the fuzzy query will expand to.

Numeric / Date Fuzzy

fuzzy query on a numeric field will result in a range query "around" the value using the **min_similarity** value. For example:

```
{
    "fuzzy" : {
        "price" : {
            "value" : 12,
            "min_similarity" : 2
        }
    }
}
```

Will result in a range query between 10 and 14. Same applies to dates, with support for time format for the **min_similarity** field:

```
{
    "fuzzy" : {
        "created" : {
            "value" : "2010-02-05T12:05:07",
            "min_similarity" : "1d"
        }
    }
}
```

In the mapping, numeric and date types now allow to configure a **fuzzy_factor** mapping value (defaults to 1), which will be used to multiply the fuzzy value by it when used in a **query_string** type query. For example, for dates, a

fuzzy factor of "1d" will result in multiplying whatever fuzzy value provided in the min_similarity by it. Note, this is explicitly supported since query_string query only allowed for similarity valued between 0.0 and 1.0.

Has Child Query

The **has_child** query works the same as the *has_child* filter, by automatically wrapping the filter with a *constant_score*. It has the same syntax as the *has_child* filter:

```
{
    "has_child" : {
        "type" : "blog_tag"
        "query" : {
            "term" : {
                "tag" : "something"
            }
        }
    }
}
```

Scope

A **_scope** can be defined on the filter allowing to run facets on the same scope name that will work against the child documents. For example:

```
{
    "has_child" : {
        "_scope" : "my_scope",
        "type" : "blog_tag"
        "query" : {
            "term" : {
                "tag" : "something"
            }
        }
    }
}
```

Memory Considerations

With the current implementation, all **_id** values are loaded to memory (heap) in order to support fast lookups, so make sure there is enough mem for it.

Ids Query

Filters documents that only have the provided ids. Note, this filter does not require the **_id** field to be indexed since it works using the **_uid** field.

```
{
    "ids" : {
        "type" : "my_type",
        "values" : ["1", "4", "100"]
    }
}
```

The **type** is optional and can be omitted, and can also accept an array of values.

Indices Query

The **indices** query can be used when executed across multiple indices, allowing to have a query that executes only when executed on an index that matches a specific list of indices, and another query that executes when it is executed on an index that does not match the listed indices.

```
{
    "indices" : {
        "indices" : ["index1", "index2"],
        "query" : {
            "term" : { "tag" : "wow" }
        },
        "no_match_query" : {
            "term" : { "tag" : "kow" }
        }
    }
}
```

no_match_query can also have "string" value of **none** (to match no documents), and **all** (to match all).

Match All Query

A query that matches all documents. Maps to Lucene **MatchAllDocsQuery**.

```
{
    "match_all" : { }
}
```

Which can also have boost associated with it:

```
{
    "match_all" : { "boost" : 1.2 }
}
```

Index Time Boost

When indexing, a boost value can either be associated on the document level, or per field. The match all query does not take boosting into account by default. In order to take boosting into account, the **norms_field** needs to be provided in order to explicitly specify which field the boosting will be done on (Note, this will result in slower execution time). For example:

```
{
    "match_all" : { "norms_field" : "my_field" }
}
```

Mlt Field Query

The **more_like_this_field** query is the same as the **more_like_this** query, except it runs against a single field. It provides nicer query DSL over the generic **more_like_this** query, and support typed fields query (automatically wraps typed fields with type filter to match only on the specific type).

```
{
    "more_like_this_field" : {
        "name.first" : {
            "like_text" : "text like this one",
            "min_term_freq" : 1,
            "max_query_terms" : 12
        }
    }
}
```

Note **more_like_this_field** can be shortened to **mlt_field**.

The **more_like_this_field** top level parameters include:

Parameter	Description
like_text	The text to find documents like it, *required*.
percent_terms_to_match	The percentage of terms to match on (float value). Defaults to **0.3** (30 percent).
min_term_freq	The frequency below which terms will be ignored in the source doc. The default frequency is **2**.
max_query_terms	The maximum number of query terms that will be included in any generated query. Defaults to **25**.
stop_words	An array of stop words. Any word in this set is considered "uninteresting" and ignored. Even if your Analyzer allows stopwords, you might want to tell the MoreLikeThis code to ignore them, as for the purposes of document similarity it seems reasonable to assume that "a stop word is never interesting".
min_doc_freq	The frequency at which words will be ignored which do not occur in at least this many docs. Defaults to **5**.
max_doc_freq	The maximum frequency in which words may still appear. Words that appear in more than this many docs will be ignored. Defaults to unbounded.
min_word_len	The minimum word length below which words will be ignored. Defaults to **0**.
max_word_len	The maximum word length above which words will be ignored. Defaults to unbounded (**0**).
boost_terms	Sets the boost factor to use when boosting terms. Defaults to **1**.
boost	Sets the boost value of the query. Defaults to **1.0**.
analyzer	The analyzer that will be used to analyze the text. Defaults to the analyzer associated with the field.

Mlt Query

More like this query find documents that are "like" provided text by running it against one or more fields.

```
{
    "more_like_this" : {
        "fields" : ["name.first", "name.last"],
        "like_text" : "text like this one",
        "min_term_freq" : 1,
        "max_query_terms" : 12
    }
}
```

Note **more_like_this** can be shortened to **mlt**.

The **more_like_this** top level parameters include:

Parameter	Description
fields	A list of the fields to run the more like this query against. Defaults to the **_all** field.
like_text	The text to find documents like it, *required*.
percent_terms_to_match	The percentage of terms to match on (float value). Defaults to **0.3** (30 percent).
min_term_freq	The frequency below which terms will be ignored in the source doc. The default frequency is **2**.
max_query_terms	The maximum number of query terms that will be included in any generated query. Defaults to **25**.
stop_words	An array of stop words. Any word in this set is considered "uninteresting" and ignored. Even if your Analyzer allows stopwords, you might want to tell the MoreLikeThis code to ignore them, as for the purposes of document similarity it seems reasonable to assume that "a stop word is never interesting".
min_doc_freq	The frequency at which words will be ignored which do not occur in at least this many docs. Defaults to **5**.
max_doc_freq	The maximum frequency in which words may still appear. Words that appear in more than this many docs will be ignored. Defaults to unbounded.
min_word_len	The minimum word length below which words will be ignored. Defaults to **0**.
max_word_len	The maximum word length above which words will be ignored. Defaults to unbounded (**0**).
boost_terms	Sets the boost factor to use when boosting terms. Defaults to **1**.
boost	Sets the boost value of the query. Defaults to **1.0**.
analyzer	The analyzer that will be used to analyze the text. Defaults to the analyzer associated with the field.

Multi Term Rewrite

Multi term queries, like *wildcard* and *prefix* are called multi term queries and end up going through a process of rewrite. This also happens on the *query_string*. All of those queries allow to control how they will get rewritten using the **rewrite** parameter:

- When not set, or set to **constant_score_default**, defaults to automatically choosing either **constant_score_boolean** or **constant_score_filter** based on query characteristics.
- **scoring_boolean**: A rewrite method that first translates each term into a should clause in a boolean query, and keeps the scores as computed by the query. Note that typically such scores are meaningless to the user, and require non-trivial CPU to compute, so it's almost always better to use **constant_score_default**. This rewrite method will hit too many clauses failure if it exceeds the boolean query limit (defaults to **1024**).
- **constant_score_boolean**: Similar to **scoring_boolean** except scores are not computed. Instead, each matching document receives a constant score equal to the query's boost. This rewrite method will hit too many clauses failure if it exceeds the boolean query limit (defaults to **1024**).
- **constant_score_filter**: A rewrite method that first creates a private Filter by visiting each term in sequence and marking all docs for that term. Matching documents are assigned a constant score equal to the query's boost.
- **top_terms_N**: A rewrite method that first translates each term into should clause in boolean query, and keeps the scores as computed by the query. This rewrite method only uses the top scoring terms so it will not overflow boolean max clause count. The **N** controls the size of the top scoring terms to use.
- **top_terms_boost_N**: A rewrite method that first translates each term into should clause in boolean query, but the scores are only computed as the boost. This rewrite method only uses the top scoring terms so it will not overflow the boolean max clause count. The **N** controls the size of the top scoring terms to use.

Nested Query

Nested query allows to query nested objects / docs (see *nested mapping*. The query is executed against the nested objects / docs as if they were indexed as separate docs (they are, internally) and resulting in the root parent doc (or parent nested mapping). Here is a sample mapping we will work with:

```
{
    "type1" : {
        "properties" : {
            "obj1" : {
                "type" : "nested"
            }
        }
    }
}
```

And here is a sample nested query usage:

```
{
    "nested" : {
        "path" : "obj1",
        "score_mode" : "avg",
        "query" : {
            "bool" : {
                "must" : [
                    {
                        "text" : {"obj1.name" : "blue"}
                    },
                    {
                        "range" : {"obj1.count" : {"gt" : 5}}
                    }
                ]
            }
        }
    }
}
```

The query **path** points to the nested object path, and the **query** (or **filter**) includes the query that will run on the nested docs matching the direct path, and joining with the root parent docs.

The **score_mode** allows to set how inner children matching affects scoring of parent. It defaults to **avg**, but can be **total**, **max** and **none**.

Multi level nesting is automatically supported, and detected, resulting in an inner nested query to automatically match the relevant nesting level (and not root) if it exists within another nested query.

Prefix Query

Matches documents that have fields containing terms with a specified prefix (*not analyzed*). The prefix query maps to Lucene **PrefixQuery**. The following matches documents where the user field contains a term that starts with **ki**:

```
{
    "prefix" : { "user" : "ki" }
}
```

A boost can also be associated with the query:

```
{
    "prefix" : { "user" :  { "value" : "ki", "boost" : 2.0 } }
}
```

Or :

```
{
    "prefix" : { "user" :  { "prefix" : "ki", "boost" : 2.0 } }
}
```

This multi term query allows to control how it gets rewritten using the *rewrite* parameter.

Query String Query

A query that uses a query parser in order to parse its content. Here is an example:

```
{
    "query_string" : {
        "default_field" : "content",
        "query" : "this AND that OR thus"
    }
}
```

The **query_string** top level parameters include:

Parameter	Description
query	The actual query to be parsed.
default_field	The default field for query terms if no prefix field is specified. Defaults to the **index.query.default_field** index settings, which in turn defaults to **_all**.
default_operator	The default operator used if no explicit operator is specified. For example, with a default operator of **OR**, the query **capital of Hungary** is translated to **capital OR of OR Hungary**, and with default operator of **AND**, the same query is translated to **capital AND of AND Hungary**. The default value is **OR**.
analyzer	The analyzer name used to analyze the query string.
allow_leading_wildcard	When set, ***** or **?** are allowed as the first character. Defaults to **true**.
lowercase_expanded_terms	Whether terms of wildcard, prefix, fuzzy, and range queries are to be automatically lower-cased or not (since they are not analyzed). Default it **true**.
enable_position_increments	Set to **true** to enable position increments in result queries. Defaults to **true**.
fuzzy_prefix_length	Set the prefix length for fuzzy queries. Default is **0**.
fuzzy_min_sim	Set the minimum similarity for fuzzy queries. Defaults to **0.5**
phrase_slop	Sets the default slop for phrases. If zero, then exact phrase matches are required. Default value is **0**.
boost	Sets the boost value of the query. Defaults to **1.0**.
analyze_wildcard	By default, wildcards terms in a query string are not analyzed. By setting this value to **true**, a best effort will be made to analyze those as well.
auto_generate_phrase_queries	Default to **false**.
minimum_should_match	A percent value (for example **20%**) controlling how many "should" clauses in the resulting boolean query should match.
lenient	If set to **true** will cause format based failures (like providing text to a numeric field) to be ignored. (since 0.19.4).

When a multi term query is being generated, one can control how it gets rewritten using the *rewrite* parameter.

Default Field

When not explicitly specifying the field to search on in the query string syntax, the **index.query.default_field** will be used to derive which field to search on. It defaults to **_all** field.

So, if **_all** field is disabled, it might make sense to change it to set a different default field.a

Multi Field

The **query_string** query can also run against multiple fields. The idea of running the **query_string** query against multiple fields is by internally creating several queries for the same query string, each with **default_field** that match the fields provided. Since several queries are generated, combining them can be automatically done either using a **dis_max** query or a simple **bool** query. For example (the **name** is boosted by 5 using **^5** notation):

```
{
    "query_string" : {
        "fields" : ["content", "name^5"],
        "query" : "this AND that OR thus",
        "use_dis_max" : true
    }
}
```

Simple wildcard can also be used to search "within" specific inner elements of the document. For example, if we have a **city** object with several fields (or inner object with fields) in it, we can automatically search on all "city" fields:

```
{
    "query_string" : {
        "fields" : ["city.*"],
        "query" : "this AND that OR thus",
        "use_dis_max" : true
    }
}
```

Another option is to provide the wildcard fields search in the query string itself (properly escapign the * sign), for example: **city.*:something**. (since 0.19.4).

When running the **query_string** query against multiple fields, the following additional parameters are allowed:

Parameter	Description
use_dis_max	Should the queries be combined using **dis_max** (set it to **true**), or a **bool** query (set it to **false**). Defaults to **true**.
tie_breaker	When using **dis_max**, the disjunction max tie breaker. Defaults to **0**.

The fields parameter can also include pattern based field names, allowing to automatically expand to the relevant fields (dynamically introduced fields included). For example:

```
{
    "query_string" : {
        "fields" : ["content", "name.*^5"],
        "query" : "this AND that OR thus",
        "use_dis_max" : true
    }
}
```

<h1 id="Syntax_Extension">Syntax Extension</h1>

There are several syntax extensions to the Lucene query language.

missing / exists

The **_exists_** and **_missing_** syntax allows to control docs that have fields that exists within them (have a value) and missing. The syntax is: **_exists_:field1**, **_missing_:field** and can be used anywhere a query string is used.

Range Query

Matches documents with fields that have terms within a certain range. The type of the Lucene query depends on the field type, for **string** fields, the **TermRangeQuery**, while for number/date fields, the query is a **NumericRangeQuery**. The following example returns all documents where **age** is between **10** and **20**:

```
{
    "range" : {
        "age" : {
            "from" : 10,
            "to" : 20,
            "include_lower" : true,
            "include_upper": false,
            "boost" : 2.0
        }
    }
}
```

The **range** query top level parameters include:

Name	Description
from	The lower bound. Defaults to start from the first.
to	The upper bound. Defaults to unbounded.
include_lower	Should the first from (if set) be inclusive or not. Defaults to **true**
include_upper	Should the last to (if set) be inclusive or not. Defaults to **true**.
gt	Same as setting **from** to the value, and **include_lower** to **false**.
gte	Same as setting **from** to the value,and **include_lower** to **true**.
lt	Same as setting **to** to the value, and **include_upper** to **false**.
lte	Same as setting **to** to the value, and **include_upper** to **true**.
boost	Sets the boost value of the query. Defaults to **1.0**.

Span First Query

Matches spans near the beginning of a field. The span first query maps to Lucene **SpanFirstQuery**. Here is an example:

```
{
    "span_first" : {
        "match" : {
            "span_term" : { "user" : "kimchy" }
        },
        "end" : 3
    }
}
```

The **match** clause can be any other span type query. The **end** controls the maximum end position permitted in a match.

Span Near Query

Matches spans which are near one another. One can specify _slop_, the maximum number of intervening unmatched positions, as well as whether matches are required to be in-order. The span near query maps to Lucene **SpanNearQuery**. Here is an example:

```
{
    "span_near" : {
        "clauses" : [
            { "span_term" : { "field" : "value1" } },
            { "span_term" : { "field" : "value2" } },
            { "span_term" : { "field" : "value3" } }
        ],
        "slop" : 12,
        "in_order" : false,
        "collect_payloads" : false
    }
}
```

The **clauses** element is a list of one or more other span type queries and the **slop** controls the maximum number of intervening unmatched positions permitted.

Span Not Query

Removes matches which overlap with another span query. The span not query maps to Lucene **SpanNotQuery**. Here is an example:

```
{
    "span_not" : {
        ©"include" : {
            "span_term" : { "field1" : "value1" }
        },
        "exclude" : {
            "span_term" : { "field2" : "value2" }
        }
    }
}
```

The **include** and **exclude** clauses can be any span type query. The **include** clause is the span query whose matches are filtered, and the **exclude** clause is the span query whose matches must not overlap those returned.

Span Or Query

Matches the union of its span clauses. The span or query maps to Lucene **SpanOrQuery**. Here is an example:

```
{
    "span_or" : {
        "clauses" : [
            { "span_term" : { "field" : "value1" } },
            { "span_term" : { "field" : "value2" } },
            { "span_term" : { "field" : "value3" } }
        ]
    }
}
```

The **clauses** element is a list of one or more other span type queries.

Span Term Query

Matches spans containing a term. The span term query maps to Lucene **SpanTermQuery**. Here is an example:

```
{
    "span_term" : { "user" : "kimchy" }
}
```

A boost can also be associated with the query:

```
{
    "span_term" : { "user" : { "value" : "kimchy", "boost" : 2.0 } }
}
```

Or :

```
{
    "span_term" : { "user" : { "term" : "kimchy", "boost" : 2.0 } }
}
```

Term Query

Matches documents that have fields that contain a term (*not analyzed*). The term query maps to Lucene **TermQuery**. The following matches documents where the user field contains the term **kimchy**:

```
{
    "term" : { "user" : "kimchy" }
}
```

A boost can also be associated with the query:

```
{
    "term" : { "user" : { "value" : "kimchy", "boost" : 2.0 } }
}
```

Or :

```
{
    "term" : { "user" : { "term" : "kimchy", "boost" : 2.0 } }
}
```

Terms Query

A query that match on any (configurable) of the provided terms. This is a simpler syntax query for using a **bool** query with several **term** queries in the **should** clauses. For example:

```
{
    "terms" : {
        "tags" : [ "blue", "pill" ],
        "minimum_match" : 1
    }
}
```

The **terms** query is also aliased with **in** as the query name for simpler usage.

Text Query

A family of **text** queries that accept text, analyzes it, and constructs a query out of it. For example:

```
{
    "text" : {
        "message" : "this is a test"
    }
}
```

Note, even though the name is text, it also supports exact matching (**term** like) on numeric values and dates.

Note, **message** is the name of a field, you can subsitute the name of any field (including **_all**) instead.

Types of Text Queries

boolean

The default **text** query is of type **boolean**. It means that the text provided is analyzed and the analysis process constructs a boolean query from the provided text. The **operator** flag can be set to **or** or **and** to control the boolean clauses (defaults to **or**).

The **analyzer** can be set to control which analyzer will perform the analysis process on the text. It default to the field explicit mapping definition, or the default search analyzer.

fuzziness can be set to a value (depending on the relevant type, for string types it should be a value between **0.0** and **1.0**) to constructs fuzzy queries for each term analyzed. The **prefix_length** and **max_expansions** can be set in this case to control the fuzzy process.

Here is an example when providing additional parameters (note the slight change in structure, **message** is the field name):

```
{
    "text" : {
        "message" : {
            "query" : "this is a test",
            "operator" : "and"
        }
    }
}
```

phrase

The **text_phrase** query analyzes the text and creates a **phrase** query out of the analyzed text. For example:

```
{
    "text_phrase" : {
        "message" : "this is a test"
    }
}
```

Since **text_phrase** is only a **type** of a **text** query, it can also be used in the following manner:

```
{
    "text" : {
```

```
        "message" : {
            "query" : "this is a test",
            "type" : "phrase"
        }
    }
}
```

A phrase query maintains order of the terms up to a configurable **slop** (which defaults to 0).

The **analyzer** can be set to control which analyzer will perform the analysis process on the text. It default to the field explicit mapping definition, or the default search analyzer, for example:

```
{
    "text_phrase" : {
        "message" : {
            "query" : "this is a test",
            "analyzer" : "my_analyzer"
        }
    }
}
```

text_phrase_prefix

The **text_phrase_prefix** is the same as **text_phrase**, expect it allows for prefix matches on the last term in the text. For example:

```
{
    "text_phrase_prefix" : {
        "message" : "this is a test"
    }
}
```

Or:

```
{
    "text" : {
        "message" : {
            "query" : "this is a test",
            "type" : "phrase_prefix"
        }
    }
}
```

It accepts the same parameters as the phrase type. In addition, it also accepts a **max_expansions** parameter that can control to how many prefixes the last term will be expanded. It is highly recommended to set it to an acceptable value to control the execution time of the query. For example:

```
{
    "text_phrase_prefix" : {
        "message" : {
            "query" : "this is a test",
            "max_expansions" : 10
        }
    }
}
```

Comparison to query_string / field

The text family of queries does not go through a "query parsing" process. It does not support field name prefixes, wildcard characters, or other "advance" features. For this reason, chances of it failing are very small / non existent, and it provides an excellent behavior when it comes to just analyze and run that text as a query behavior (which is usually what a text search box does). Also, the **phrase_prefix** can provide a great "as you type" behavior to automatically load search results.

Top Children Query

The **top_children** query runs the child query with an estimated hits size, and out of the hit docs, aggregates it into parent docs. If there aren't enough parent docs matching the requested from/size search request, then it is run again with a wider (more hits) search.

The **top_children** also provide scoring capabilities, with the ability to specify **max**, **sum** or **avg** as the score type.

One downside of using the **top_children** is that if there are more child docs matching the required hits when executing the child query, then the **total_hits** result of the search response will be incorrect.

How many hits are asked for in the first child query run is controlled using the **factor** parameter (defaults to **5**). For example, when asking for 10 parent docs (with **from** set to 0), then the child query will execute with 50 hits expected. If not enough parents are found (in our example 10), and there are still more child docs to query, then the child search hits are expanded by multiplying by the **incremental_factor** (defaults to **2**).

The required parameters are the **query** and **type** (the child type to execute the query on). Here is an example with all different parameters, including the default values:

```
{
    "top_children" : {
        :ref:`type <es-guide-reference-query-dsl>`  pe <es-guide-reference-query-dsl>
↪`  "blog_tag",
        "query" : {
            "term" : {
                "tag" : "something"
            }
        }
        "score" : "max",
        "factor" : 5,
        "incremental_factor" : 2
    }
}
```

Scope

A **_scope** can be defined on the query allowing to run facets on the same scope name that will work against the child documents. For example:

```
{
    "top_children" : {
        "_scope" : "my_scope",
        :ref:`type <es-guide-reference-query-dsl>`  pe <es-guide-reference-query-dsl>
↪`  "blog_tag",
        "query" : {
            "term" : {
                "tag" : "something"
```

```
            }
        }
    }
}
```

Memory Considerations

With the current implementation, all **_id** values are loaded to memory (heap) in order to support fast lookups, so make sure there is enough mem for it.

Wildcard Query

Matches documents that have fields matching a wildcard expression (*not analyzed*). Supported wildcards are ***, which matches any character sequence (including the empty one), and **?**, which matches any single character. Note this query can be slow, as it needs to iterate over many terms. In order to prevent extremely slow wildcard queries, a wildcard term should not start with one of the wildcards * or **?**. The wildcard query maps to Lucene **WildcardQuery**.

```
{
    "wildcard" : { "user" : "ki*y" }
}
```

A boost can also be associated with the query:

```
{
    "wildcard" : { "user" : { "value" : "ki*y", "boost" : 2.0 } }
}
```

Or :

```
{
    "wildcard" : { "user" : { "wildcard" : "ki*y", "boost" : 2.0 } }
}
```

This multi term query allows to control how it gets rewritten using the *rewrite* parameter.

Filter:

And Filter

A filter that matches documents using **AND** boolean operator on other queries. This filter is more performant then *bool* filter. Can be placed within queries that accept a filter.

```
{
    "filtered" : {
        "query" : {
            "term" : { "name.first" : "shay" }
        },
        "filter" : {
            "and" : [
                {
                    "range" : {
                        "postDate" : {
                            "from" : "2010-03-01",
                            "to" : "2010-04-01"
```

```
                        }
                    }
                },
                {
                    "prefix" : { "name.second" : "ba" }
                }
            ]
        }
    }
}
```

Caching

The result of the filter is not cached by default. The **_cache** can be set to **true** in order to cache it (tough usually not needed). Since the **_cache** element requires to be set on the **and** filter itself, the structure then changes a bit to have the filters provided within a **filters** element:

```
{
    "filtered" : {
        "query" : {
            "term" : { "name.first" : "shay" }
        },
        "filter" : {
            "and" :
                "filters": [
                    {
                        "range" : {
                            "postDate" : {
                                "from" : "2010-03-01",
                                "to" : "2010-04-01"
                            }
                        }
                    },
                    {
                        "prefix" : { "name.second" : "ba" }
                    }
                ],
                "_cache" : true
            }
        }
    }
}
```

Bool Filter

A filter that matches documents matching boolean combinations of other queries. Similar in concept to Boolean query, except that the clauses are other filters. Can be placed within queries that accept a filter.

```
{
    "filtered" : {
        "query" : {
            "queryString" : {
                "default_field" : "message",
                "query" : "elasticsearch"
```

```
            }
        },
        "filter" : {
            "bool" : {
                "must" : {
                    "term" : { "tag" : "wow" }
                },
                "must_not" : {
                    "range" : {
                        "age" : { "from" : 10, "to" : 20 }
                    }
                },
                "should" : [
                    {
                        "term" : { "tag" : "sometag" }
                    },
                    {
                        "term" : { "tag" : "sometagtag" }
                    }
                ]
            }
        }
    }
}
```

Caching

The result of the **bool** filter is not cached by default (though internal filters might be). The **_cache** can be set to **true** in order to enable caching.

Custom Filters Score Query

A **custom_filters_score** query allows to execute a query, and if the hit matches a provided filter (ordered), use either a boost or a script associated with it to compute the score. Here is an example:

```
{
    "custom_filters_score" : {
        "query" : {
            "match_all" : {}
        },
        "filters" : [
            {
                "filter" : { "range" : { "age" : {"from" : 0, "to" : 10} } },
                "boost" : "3"
            },
            {
                "filter" : { "range" : { "age" : {"from" : 10, "to" : 20} } },
                "boost" : "2"
            }
        ],
        "score_mode" : "first"
    }
}
```

This can considerably simplify and increase performance for parameterized based scoring since filters are easily cached for faster performance, and boosting / script is considerably simpler.

Score Mode

A **score_mode** can be defined to control how multiple matching filters control the score. By default, it is set to **first** which means the first matching filter will control the score of the result. It can also be set to **min/max/total/avg/multiply** which will aggregate the result from all matching filters based on the aggregation type.

max_boost

An option to cap the boost value computed.

Script

A **script** can be used instead of **boost** for more complex score calculations. With optional **params** and **lang** (on the same level as **query** and **filters**).

Exists Filter

Filters documents where a specific field has a value in them.

```
{
    "constant_score" : {
        "filter" : {
            "exists" : { "field" : "user" }
        }
    }
}
```

Caching

The result of the filter is always cached.

Geo Bounding Box Filter

A filter allowing to filter hits based on a point location using a bounding box. Assuming the following indexed document:

```
{
    "pin" : {
        "location" : {
            "lat" : 40.12,
            "lon" : -71.34
        }
    }
}
```

Then the following simple query can be executed with a **geo_bounding_box** filter:

```
{
    "filtered" : {
        "query" : {
            "match_all" : {}
        },
        "filter" : {
            "geo_bounding_box" : {
                "pin.location" : {
                    "top_left" : {
                        "lat" : 40.73,
                        "lon" : -74.1
                    },
                    "bottom_right" : {
                        "lat" : 40.717,
                        "lon" : -73.99
                    }
                }
            }
        }
    }
}
```

Accepted Formats

In much the same way the geo_point type can accept different representation of the geo point, the filter can accept it as well:

Lat Lon As Properties

```
{
    "filtered" : {
        "query" : {
            "match_all" : {}
        },
        "filter" : {
            "geo_bounding_box" : {
                "pin.location" : {
                    "top_left" : {
                        "lat" : 40.73,
                        "lon" : -74.1
                    },
                    "bottom_right" : {
                        "lat" : 40.717,
                        "lon" : -73.99
                    }
                }
            }
        }
    }
}
```

Lat Lon As Array

Format in **[lon, lat]**, note, the order of lon/lat here in order to conform with GeoJSON.

```
{
    "filtered" : {
        "query" : {
            "match_all" : {}
        },
        "filter" : {
            "geo_bounding_box" : {
                "pin.location" : {
                    "top_left" : [40.73, -74.1],
                    "bottom_right" : [40.717, -73.99]
                }
            }
        }
    }
}
```

Lat Lon As String

Format in **lat,lon**.

```
{
    "filtered" : {
        "query" : {
            "match_all" : {}
        },
        "filter" : {
            "geo_bounding_box" : {
                "pin.location" : {
                    "top_left" : "40.73, -74.1",
                    "bottom_right" : "40.717, -73.99"
                }
            }
        }
    }
}
```

Geohash

```
{
    "filtered" : {
        "query" : {
            "match_all" : {}
        },
        "filter" : {
            "geo_bounding_box" : {
                "pin.location" : {
                    "top_left" : "drm3btev3e86",
                    "bottom_right" : "drm3btev3e86"
                }
            }
```

```
        }
    }
}
```

geo_point Type

The filter *requires* the **geo_point** type to be set on the relevant field.

Multi Location Per Document

The filter can work with multiple locations / points per document. Once a single location / point matches the filter, the document will be included in the filter

Type

The type of the bounding box execution by default is set to **memory**, which means in memory checks if the doc falls within the bounding box range. In some cases, an **indexed** option will perform faster (but note that the **geo_point** type must have lat and lon indexed in this case). Note, when using the indexed option, multi locations per document field are not supported. Here is an example:

```
{
    "filtered" : {
        "query" : {
            "match_all" : {}
        },
        "filter" : {
            "geo_bounding_box" : {
                "pin.location" : {
                    "top_left" : {
                        "lat" : 40.73,
                        "lon" : -74.1
                    },
                    "bottom_right" : {
                        "lat" : 40.717,
                        "lon" : -73.99
                    }
                },
                "type" : "indexed"
            }
        }
    }
}
```

Caching

The result of the filter is not cached by default. The **_cache** can be set to **true** to cache the *result* of the filter. This is handy when the same bounding box parameters are used on several (many) other queries. Note, the process of caching the first execution is higher when caching (since it needs to satisfy different queries).

Geo Distance Filter

Filters documents that include only hits that exists within a specific distance from a geo point. Assuming the following indexed json:

```
{
    "pin" : {
        "location" : {
            "lat" : 40.12,
            "lon" : -71.34
        }
    }
}
```

Then the following simple query can be executed with a **geo_distance** filter:

```
{
    "filtered" : {
        "query" : {
            "match_all" : {}
        },
        "filter" : {
            "geo_distance" : {
                "distance" : "200km",
                "pin.location" : {
                    "lat" : 40,
                    "lon" : -70
                }
            }
        }
    }
}
```

Accepted Formats

In much the same way the **geo_point** type can accept different representation of the geo point, the filter can accept it as well:

Lat Lon As Properties

```
{
    "filtered" : {
        "query" : {
            "match_all" : {}
        },
        "filter" : {
            "geo_distance" : {
                "distance" : "12km",
                "pin.location" : {
                    "lat" : 40,
                    "lon" : -70
                }
            }
        }
```

```
    }
}
```

Lat Lon As Array

Format in **[lon, lat]**, note, the order of lon/lat here in order to conform with GeoJSON.

```
{
    "filtered" : {
        "query" : {
            "match_all" : {}
        },
        "filter" : {
            "geo_distance" : {
                "distance" : "12km",
                "pin.location" : [40, -70]
            }
        }
    }
}
```

Lat Lon As String

Format in **lat,lon**.

```
{
    "filtered" : {
        "query" : {
            "match_all" : {}
        },
        "filter" : {
            "geo_distance" : {
                "distance" : "12km",
                "pin.location" : "40,-70"
            }
        }
    }
}
```

Geohash

```
{
    "filtered" : {
        "query" : {
            "match_all" : {}
        },
        "filter" : {
            "geo_distance" : {
                "distance" : "12km",
                "pin.location" : "drm3btev3e86"
            }
        }
```

```
    }
}
```

Options

The following are options allowed on the filter:

Option	Description
distance	The distance to include hits in the filter. The distance can be a numeric value, and then the **distance_unit** (either **mi/miles** or **km** can be set) controlling the unit. Or a single string with the unit as well.
distance_type	How to compute the distance. Can either be **arc** (better precision) or **plane** (faster). Defaults to **arc**.
optimize_bbox	Will an optimization of using first a bounding box check will be used. Defaults to **memory** which will do in memory checks. Can also have values of **indexed** to use indexed value check (make sure the **geo_point** type index lat lon in this case), or **none** which disables bounding box optimization.

geo_point Type

The filter *requires* the **geo_point** type to be set on the relevant field.

Multi Location Per Document

The **geo_distance** filter can work with multiple locations / points per document. Once a single location / point matches the filter, the document will be included in the filter.

Caching

The result of the filter is not cached by default. The **_cache** can be set to **true** to cache the *result* of the filter. This is handy when the same point and distance parameters are used on several (many) other queries. Note, the process of caching the first execution is higher when caching (since it needs to satisfy different queries).

Geo Distance Range Filter

Filters documents that exists within a range from a specific point:

```
{
    "filtered" : {
        "query" : {
            "match_all" : {}
        },
        "filter" : {
            "geo_distance_range" : {
                "from" : "200km",
                "to" : "400km"
                "pin.location" : {
                    "lat" : 40,
                    "lon" : -70
                }
            }
```

```
        }
    }
}
```

Supports the same point location parameter as the **geo_distance** filter. And also support the common parameters for range (lt, lte, gt, gte, from, to, include_upper and include_lower).

Geo Polygon Filter

A filter allowing to include hits that only fall within a polygon of points. Here is an example:

```
{
    "filtered" : {
        "query" : {
            "match_all" : {}
        },
        "filter" : {
            "geo_polygon" : {
                "person.location" : {
                    "points" : [
                        {"lat" : 40, "lon" : -70},
                        {"lat" : 30, "lon" : -80},
                        {"lat" : 20, "lon" : -90}
                    ]
                }
            }
        }
    }
}
```

Allowed Formats

Lat Long as Array

Format in **[lon, lat]**, note, the order of lon/lat here in order to conform with GeoJSON.

```
{
    "filtered" : {
        "query" : {
            "match_all" : {}
        },
        "filter" : {
            "geo_polygon" : {
                "person.location" : {
                    "points" : [
                        [-70, 40],
                        [-80, 30],
                        [-90, 20]
                    ]
                }
            }
        }
    }
}
```

Lat Lon as String

Format in **lat,lon**.

```
{
    "filtered" : {
        "query" : {
            "match_all" : {}
        },
        "filter" : {
            "geo_polygon" : {
                "person.location" : {
                    "points" : [
                        "40, -70",
                        "30, -80",
                        "20, -90"
                    ]
                }
            }
        }
    }
}
```

Geohash

```
{
    "filtered" : {
        "query" : {
            "match_all" : {}
        },
        "filter" : {
            "geo_polygon" : {
                "person.location" : {
                    "points" : [
                        "drn5x1g8cu2y",
                        "30, -80",
                        "20, -90"
                    ]
                }
            }
        }
    }
}
```

geo_point Type

The filter *requires* the **geo_point** type to be set on the relevant field.

Caching

The result of the filter is not cached by default. The **_cache** can be set to **true** to cache the *result* of the filter. This is handy when the same points parameters are used on several (many) other queries. Note, the process of caching the first execution is higher when caching (since it needs to satisfy different queries).

Has Child Filter

The **has_child** filter accepts a query and the child type to run against, and results in parent documents that have child docs matching the query. Here is an example:

```
{
    "has_child" : {
        "type" : "blog_tag"
        "query" : {
            "term" : {
                "tag" : "something"
            }
        }
    }
}
```

The **type** is the child type to query against. The parent type to return is automatically detected based on the mappings.

The way that the filter is implemented is by first running the child query, doing the matching up to the parent doc for each document matched.

Scope

A **_scope** can be defined on the filter allowing to run facets on the same scope name that will work against the child documents. For example:

```
{
    "has_child" : {
        "_scope" : "my_scope",
        "type" : "blog_tag"
        "query" : {
            "term" : {
                "tag" : "something"
            }
        }
    }
}
```

Memory Considerations

With the current implementation, all **_id** values are loaded to memory (heap) in order to support fast lookups, so make sure there is enough mem for it.

Ids Filter

Filters documents that only have the provided ids. Note, this filter does not require the **_id** field to be indexed since it works using the **_uid** field.

```
{
    "ids" : {
        "type" : "my_type",
        "values" : ["1", "4", "100"]
    }
}
```

The **type** is optional and can be omitted, and can also accept an array of values.

Limit Filter

A limit filter limits the number of documents (per shard) to execute on. For example:

```
{
    "filtered" : {
        "filter" : {
             "limit" : {"value" : 100}
         },
         "query" : {
            "term" : { "name.first" : "shay" }
        }
    }
}
```

Match All Filter

A filter that matches on all documents:

```
{
    "constant_score" : {
        "filter" : {
            "match_all" : { }
        }
    }
}
```

Missing Filter

Filters documents where a specific field has no value in them.

```
{
    "constant_score" : {
        "filter" : {
            "missing" : { "field" : "user" }
        }
    }
}
```

By default, the filter will only find "missing" fields, i.e., fields that have no values. It can be configured also to find fields with an explicit **null_value** mapped for them. Here is an example that will both find missing field that don't exists (**existence** set to **true**), or have null values (**null_value** set to **true**).

```
{
    "constant_score" : {
        "filter" : {
            "missing" : {
                "field" : "user",
                "existence" : true,
                "null_value" : true
            }
        }
```

```
    }
}
```

Caching

The result of the filter is always cached.

Nested Filter

A **nested** filter, works in a similar fashion to the *nested* query, except used as a filter. It follows exactly the same structure, but also allows to cache the results (set **_cache** to **true**), and have it named (set the **_name** value). For example:

```
{
    "filtered" : {
        "query" : { "match_all" : {} },
        "filter" : {
            "nested" : {
                "path" : "obj1",
                "query" : {
                    "bool" : {
                        "must" : [
                            {
                                "text" : {"obj1.name" : "blue"}
                            },
                            {
                                "range" : {"obj1.count" : {"gt" : 5}}
                            }
                        ]
                    }
                },
                "_cache" : true
            }
        }
    }
}
```

Not Filter

A filter that filters out matched documents using a query. This filter is more performant then *bool* filter. Can be placed within queries that accept a filter.

```
{
    "filtered" : {
        "query" : {
            "term" : { "name.first" : "shay" }
        },
        "filter" : {
            "not" : {
                "range" : {
                    "postDate" : {
                        "from" : "2010-03-01",
                        "to" : "2010-04-01"
```

```
                    }
                }
            }
        }
    }
}
```

Or, in a longer form with a **filter** element:

```
{
    "filtered" : {
        "query" : {
            "term" : { "name.first" : "shay" }
        },
        "filter" : {
            "not" : {
                "filter" :  {
                    "range" : {
                        "postDate" : {
                            "from" : "2010-03-01",
                            "to" : "2010-04-01"
                        }
                    }
                }
            }
        }
    }
}
```

Caching

The result of the filter is not cached by default. The **_cache** can be set to **true** in order to cache it (tough usually not needed). Here is an example:

```
{
    "filtered" : {
        "query" : {
            "term" : { "name.first" : "shay" }
        },
        "filter" : {
            "not" : {
                "filter" :  {
                    "range" : {
                        "postDate" : {
                            "from" : "2010-03-01",
                            "to" : "2010-04-01"
                        }
                    }
                },
                "_cache" : true
            }
        }
    }
}
```

Numeric Range Filter

Filters documents with fields that have values within a certain numeric range. Similar to range filter, except that it works only with numeric values, and the filter execution works differently.

```
{
    "constant_score" : {
        "filter" : {
            "numeric_range" : {
                "age" : {
                    "from" : "10",
                    "to" : "20",
                    "include_lower" : true,
                    "include_upper" : false
                }
            }
        }
    }
}
```

The numeric range filter works by loading all the relevant field values into memory, and checking for the relevant docs if they satisfy the range requirements. This require more memory since the numeric range data are loaded to memory, but can provide a significant increase in performance. Note, if the relevant field values have already been loaded to memory, for example because it was used in facets or was sorted on, then this filter should be used.

The **numeric_range** filter top level parameters include:

Name	Description
from	The lower bound. Defaults to start from the first.
to	The upper bound. Defaults to unbounded.
include_lower	Should the first from (if set) be inclusive or not. Defaults to **true**
include_upper	Should the last to (if set) be inclusive or not. Defaults to **true**.
gt	Same as setting **from** and **include_lower** to **false**.
gte	Same as setting **from** and **include_lower** to **true**.
lt	Same as setting **to** and **include_upper** to **false**.
lte	Same as setting **to** and **include_upper** to **true**.

Caching

The result of the filter is not cached by default. The **_cache** can be set to **true** to cache the *result* of the filter. This is handy when the same points parameters are used on several (many) other queries. Note, the process of caching the first execution is higher when caching (since it needs to satisfy different queries).

If caching the *result* of the filter is desired (for example, using the same *teen" filter with ages between 10 and 20), then it is advisable to simply use the "range* filter.

Or Filter

A filter that matches documents using **OR** boolean operator on other queries. This filter is more performant then *bool* filter. Can be placed within queries that accept a filter.

```
{
    "filtered" : {
        "query" : {
            "term" : { "name.first" : "shay" }
```

```
        },
        "filter" : {
            "or" : [
                {
                    "term" : { "name.second" : "banon" }
                },
                {
                    "term" : { "name.nick" : "kimchy" }
                }
            ]
        }
    }
}
```

Caching

The result of the filter is not cached by default. The **_cache** can be set to **true** in order to cache it (tough usually not needed). Since the **_cache** element requires to be set on the **or** filter itself, the structure then changes a bit to have the filters provided within a **filters** element:

```
{
    "filtered" : {
        "query" : {
            "term" : { "name.first" : "shay" }
        },
        "filter" : {
            "or" : {
                "filters" : [
                    {
                        "term" : { "name.second" : "banon" }
                    },
                    {
                        "term" : { "name.nick" : "kimchy" }
                    }
                ],
                "_cache" : true
            }
        }
    }
}
```

Prefix Filter

Filters documents that have fields containing terms with a specified prefix (*not analyzed*). Similar to phrase query, except that it acts as a filter. Can be placed within queries that accept a filter.

```
{
    "constant_score" : {
        "filter" : {
            "prefix" : { "user" : "ki" }
        }
    }
}
```

Caching

The result of the filter is cached by default. The *_cache* can be set to *false* in order not to cache it. Here is an example:

```
{
    "constant_score" : {
        "filter" : {
            "prefix" : {
                "user" : "ki",
                "_cache" : false
            }
        }
    }
}
```

Query Filter

Wraps any query to be used as a filter. Can be placed within queries that accept a filter.

```
{
    "constantScore" : {
        "filter" : {
            "query" : {
                "query_string" : {
                    "query" : "this AND that OR thus"
                }
            }
        }
    }
}
```

Caching

The result of the filter is not cached by default. The **_cache** can be set to **true** to cache the *result* of the filter. This is handy when the same query is used on several (many) other queries. Note, the process of caching the first execution is higher when not caching (since it needs to satisfy different queries).

Setting the **_cache** element requires a different format for the **query**:

```
{
    "constantScore" : {
        "filter" : {
            "fquery" : {
                "query" : {
                    "query_string" : {
                        "query" : "this AND that OR thus"
                    }
                },
                "_cache" : true
            }
        }
    }
}
```

Range Filter

Filters documents with fields that have terms within a certain range. Similar to range query, except that it acts as a filter. Can be placed within queries that accept a filter.

```
{
    "constant_score" : {
        "filter" : {
            "range" : {
                "age" : {
                    "from" : "10",
                    "to" : "20",
                    "include_lower" : true,
                    "include_upper" : false
                }
            }
        }
    }
}
```

The **range** filter top level parameters include:

Name	Description
from	The lower bound. Defaults to start from the first.
to	The upper bound. Defaults to unbounded.
include_lower	Should the first from (if set) be inclusive or not. Defaults to **true**
include_upper	Should the last to (if set) be inclusive or not. Defaults to **true**.
gt	Same as setting **from** to the value, and **include_lower** to **false**.
gte	Same as setting **from** to the value, and **include_lower** to **true**.
lt	Same as setting **to** to the value, and **include_upper** to **false**.
lte	Same as setting **to** to the value, and **include_upper** to **true**.

Caching

The result of the filter is automatically cached by default. The **_cache** can be set to **false** to turn it off.

Script Filter

A filter allowing to define *scripts* as filters. For example:

```
"filtered" : {
    "query" : {
        ...
    },
    "filter" : {
        "script" : {
            "script" : "doc['num1'].value > 1"
        }
    }
}
```

Custom Parameters

Scripts are compiled and cached for faster execution. If the same script can be used, just with different parameters provider, it is preferable to use the ability to pass parameters to the script itself, for example:

```
"filtered" : {
    "query" : {
        ...
    },
    "filter" : {
        "script" : {
            "script" : "doc['num1'].value > param1"
            "params" : {
                "param1" : 5
            }
        }
    }
}
```

Caching

The result of the filter is not cached by default. The **_cache** can be set to **true** to cache the *result* of the filter. This is handy when the same script and parameters are used on several (many) other queries. Note, the process of caching the first execution is higher when caching (since it needs to satisfy different queries).

Term Filter

Filters documents that have fields that contain a term (*not analyzed*). Similar to term query, except that it acts as a filter. Can be placed within queries that accept a filter, for example:

```
{
    "constant_score" : {
        "filter" : {
            "term" : { "user" : "kimchy"}
        }
    }
}
```

Caching

The result of the filter is automatically cached by default. The *_cache* can be set to *false* to turn it off. Here is an example:

```
{
    "constant_score" : {
        "filter" : {
            "term" : {
                "user" : "kimchy",
                "_cache" : false
            }
        }
    }
}
```

Terms Filter

Filters documents that have fields that match any of the provided terms (*not analyzed*). For example:

```
{
    "constant_score" : {
        "filter" : {
            "terms" : { "user" : ["kimchy", "elasticsearch"]}
        }
    }
}
```

The **terms** filter is also aliased with **in** as the filter name for simpler usage.

Execution Mode

The way terms filter executes is by iterating over the terms provided and finding matches docs (loading into a bitset) and caching it. Sometimes, we want a different execution model that can still be achieved by building more complex queries in the DSL, but we can support them in the more compact model that terms filter provides.

The **execution** option now has the following options :

- **plain**: The default. Works as today. Iterates over all the terms, building a bit set matching it, and filtering. The total filter is cached (keyed by the terms).
- **bool**: Builds a bool filter wrapping each term in a term filter that is cached (the single term filter). The bool filter is optimized in this case since it bitwise or's the different term filter bitsets. The total filter is not cached by default in this case as it makes little sense to do so (each term on its own is cached).
- **and**: Builds an and filter wrapping each term in term filter that is cached. The total filter is not cached by default in this case. Most times, bool should be used as its faster thanks to its bitwise execution.

The "total" terms filter caching can still be explicitly controlled using the **_cache** option. Note the default value for it depends on the execution value.

For example:

```
{
    "constant_score" : {
        "filter" : {
            "terms" : {
                "user" : ["kimchy", "elasticsearch"],
                "execution" : "bool"
            }
        }
    }
}
```

Caching

The result of the filter is automatically cached by default. The *_cache* can be set to *false* to turn it off.

Type Filter

Filters documents matching the provided document / mapping type. Note, this filter can work even when the **_type** field is not indexed (using the **_uid** field).

```
{
    "type" : {
        "value" : "my_type"
    }
}
```

Mapping

Mapping is the process of defining how a document should be mapped to the Search Engine, including its searchable characteristics such as which fields are searchable and if/how they are tokenized. In ElasticSearch, an index may store documents of different "mapping types". ElasticSearch allows one to associate multiple mapping definitions for each mapping type.

Explicit mapping is defined on an index/type level. By default, there isn't a need to define an explicit mapping, since one is automatically created and registered when a new type or new field is introduced (with no performance overhead) and have sensible defaults. Only when the defaults need to be overridden must a mapping definition be provided.

Mapping Types

Mapping types are a way to try and divide the documents indexed into the same index into logical groups. Think of it as tables in a database. Though there is separation between types, it's not a full separation (all end up as a document within the same Lucene index).

Field names with the same name across types are highly recommended to have the same type and same mapping characteristics (analysis settings for example). There is an effort to allow to explicitly "choose" which field to use by using type prefix (**my_type.my_field**), but its not complete, and there are places where it will never work (like faceting on the field).

In practice though, this restriction is almost never an issue. The field name usually ends up being a good indication to its "typeness" (e.g. "first_name" will always be a string). Note also, that this does not apply to the cross index case.

Mapping API

To create a mapping, you will need the *Put Mapping API*, or you can add multiple mappings when you *create an index*.

All Field

The idea of the **_all** field is that it includes the text of one or more other fields within the document indexed. It can come very handy especially for search requests, where we want to execute a search query against the content of a document, without knowing which fields to search on. This comes at the expense of CPU cycles and index size.

The **_all** fields can be completely disabled. Explicit field mapping and object mapping can be excluded / included in the **_all** field. By default, it is enabled and all fields are included in it for ease of use.

When disabling the **_all** field, it is a good practice to set **index.query.default_field** to a different value (for example, if you have a main "message" field in your data, set it to **message**).

One of the nice features of the **_all** field is that it takes into account specific fields boost levels. Meaning that if a title field is boosted more than content, the title (part) in the **_all** field will mean more than the content (part) in the **_all** field.

Here is a sample mapping:

```
{
    "person" : {
        "_all" : {"enabled" : true},
        "properties" : {
            "name" : {
                "type" : "object",
                "dynamic" : false,
                "properties" : {
                    "first" : {"type" : "string", "store" : "yes", "include_in_all" : false},
                    "last" : {"type" : "string", "index" : "not_analyzed"}
                }
            },
            "address" : {
                "type" : "object",
                "include_in_all" : false,
                "properties" : {
                    "first" : {
                        "properties" : {
                            "location" : {"type" : "string", "store" : "yes", "index_name" : "firstLocation"}
                        }
                    },
                    "last" : {
                        "properties" : {
                            "location" : {"type" : "string"}
                        }
                    }
                }
            },
            "simple1" : {"type" : "long", "include_in_all" : true},
            "simple2" : {"type" : "long", "include_in_all" : false}
        }
    }
}
```

The **_all** fields allows for **store**, **term_vector** and **analyzer** (with specific **index_analyzer** and **search_analyzer**) to be set.

Highlighting

For any field to allow *highlighting* it has to be either stored or part of the **_source** field. By default **_all** field does not qualify for either, so highlighting for it does not yield any data.

Although it is possible to **store** the **_all** field, it is basically an aggregation of all fields, which means more data will be stored, and highlighting it might produce strange results.

Analyzer Field

The **_analyzer** mapping allows to use a document field property as the name of the analyzer that will be used to index the document. The analyzer will be used for any field that does not explicitly defines an **analyzer** or **index_analyzer** when indexing.

Here is a simple mapping:

```
{
    "type1" : {
        "_analyzer" : {
            "path" : "my_field"
        }
    }
}
```

The above will use the value of the **my_field** to lookup an analyzer registered under it. For example, indexing a the following doc:

```
{
    "my_field" : "whitespace"
}
```

Will cause the **whitespace** analyzer to be used as the index analyzer for all fields without explicit analyzer setting.

The default path value is **_analyzer**, so the analyzer can be driven for a specific document by setting **_analyzer** field in it. If custom json field name is needed, an explicit mapping with a different path should be set.

By default, the **_analyzer** field is indexed, it can be disabled by settings **index** to **no** in the mapping.

Array Type

JSON documents allow to define an array (list) of fields or objects. Mapping array types could not be simpler since arrays gets automatically detected and mapping them can be done either with *Core Types* or *Object Type* mappings. For example, the following JSON defines several arrays:

```
{
    "tweet" : {
        "message" : "some arrays in this tweet...",
        "tags" : ["elasticsearch", "wow"],
        "lists" : [
            {
                "name" : "prog_list",
                "description" : "programming list"
            },
            {
                "name" : "cool_list",
                "description" : "cool stuff list"
            }
        ]
    }
}
```

The above JSON has the **tags** property defining a list of a simple **string** type, and the **lists** property is an **object** type array. Here is a sample explicit mapping:

```
{
    "tweet" : {
        "properties" : {
            "message" : {"type" : "string"},
            "tags" : {"type" : "string", "index_name" : "tag"},
            "lists" : {
                "properties" : {
                    "name" : {"type" : "string"},
                    "description" : {"type" : "string"}
```

```
                }
            }
        }
    }
}
```

The fact that array types are automatically supported can be shown by the fact that the following JSON document is perfectly fine:

```
{
    "tweet" : {
        "message" : "some arrays in this tweet...",
        "tags" : "elasticsearch",
        "lists" : {
            "name" : "prog_list",
            "description" : "programming list"
        }
    }
}
```

Note also, that thanks to the fact that we used the **index_name** to use the non plural form (**tag** instead of **tags**), we can actually refer to the field using the **index_name** as well. For example, we can execute a query using **tweet.tags:wow** or **tweet.tag:wow**. We could, of course, name the field as **tag** and skip the **index_name** all together).

Attachment Type

The **attachment** type allows to index different attachment" type field (encoded as **base64**), for example, microsoft office formats, open document formats, ePub, HTML, and so on (full list can be found "here.

The **attachment** type is provided as a plugin extension. The plugin is a simple zip file that can be downloaded and placed under **$ES_HOME/plugins** location. It will be automatically detected and the **attachment** type will be added.

Note, the **attachment** type is experimental.

Using the attachment type is simple, in your mapping JSON, simply set a certain JSON element as attachment, for example:

```
{
    "person" : {
        "properties" : {
            "my_attachment" : { "type" : "attachment" }
        }
    }
}
```

In this case, the JSON to index can be:

```
{
    "my_attachment" : "... base64 encoded attachment ..."
}
```

Or it is possible to use more elaborated JSON if content type or resource name need to be set explicitly:

```
{
    "my_attachment" : {
        "_content_type" : "application/pdf",
        "_name" : "resource/name/of/my.pdf",
```

```
        "content" : "... base64 encoded attachment ..."
    }
}
```

The **attachment** type not only indexes the content of the doc, but also automatically adds meta data on the attachment as well (when available). The metadata supported are: **date**, **title**, **author**, and **keywords**. They can be queried using the "dot notation", for example: **my_attachment.author**.

Both the meta data and the actual content are simple core type mappers (string, date, ...), thus, they can be controlled in the mappings. For example:

```
{
    "person" : {
        "properties" : {
            "file" : {
                "type" : "attachment",
                "fields" : {
                    "file" : {"index" : "no"},
                    "date" : {"store" : "yes"},
                    "author" : {"analyzer" : "myAnalyzer"}
                }
            }
        }
    }
}
```

In the above example, the actual content indexed is mapped under **fields** name **file**, and we decide not to index it, so it will only be available in the **_all** field. The other fields map to their respective metadata names, but there is no need to specify the **type** (like **string** or **date**) since it is already known.

The plugin uses Apache Tika to parse attachments, so many formats are supported, listed here.

Boost Field

Boosting is the process of enhancing the relevancy of a document or field. Field level mapping allows to define explicit boost level on a specific field. The boost field mapping (applied on the *root object* allows to define a boost field mapping where *its content will control the boost level of the document*. For example, consider the following mapping:

```
{
    "tweet" : {
        "_boost" : {"name" : "_boost", "null_value" : 1.0}
    }
}
```

The above mapping defines mapping for a field named **_boost**. If the **_boost** field exists within the JSON document indexed, its value will control the boost level of the document indexed. For example, the following JSON document will be indexed with a boost value of **2.2**:

```
{
    "tweet" {
        "_boost" : 2.2,
        "message" : "This is a tweet!"
    }
}
```

Conf Mappings

Creating new mappings can be done using the **put_mapping** API. When a document is indexed with no mapping associated with it in the specific index, the *dynamic / default mapping* feature will kick in and automatically create mapping definition for it.

Mappings can also be provided on the node level, meaning that each index created will automatically be started with all the mappings defined within a certain location.

Mappings can be defined within files called **[mapping_name].json** and be placed either under **config/mappings/_default** location, or under **config/mappings/[index_name]** (for mappings that should be associated only with a specific index).

Core Types

Each JSON field can be mapped to a specific core type. JSON itself already provides us with some typing, with its support for **string**, **integer/long**, **float/double**, **boolean**, and **null**.

The following sample tweet JSON document will be used to explain the core types:

```
{
    "tweet" {
        "user" : "kimchy"
        "message" : "This is a tweet!",
        "postDate" : "2009-11-15T14:12:12",
        "priority" : 4,
        "rank" : 12.3
    }
}
```

Explicit mapping for the above JSON tweet can be:

```
{
    "tweet" : {
        "properties" : {
            "user" : {"type" : "string", "index" : "not_analyzed"},
            "message" : {"type" : "string", "null_value" : "na"},
            "postDate" : {"type" : "date"},
            "priority" : {"type" : "integer"},
            "rank" : {"type" : "float"}
        }
    }
}
```

String

The text based string type is the most basic type, and contains one or more characters. An example mapping can be:

```
{
    "tweet" : {
        "properties" : {
            "message" : {
                "type" : "string",
                "store" : "yes",
                "index" : "analyzed",
                "null_value" : "na"
```

```
            }
        }
    }
}
```

The above mapping defines a **string message** property/field within the **tweet** type. The field is stored in the index (so it can later be retrieved using selective loading when searching), and it gets analyzed (broken down into searchable terms). If the message has a **null** value, then then value that will be stored is **na**.

The following table lists all the attributes that can be used with the **string** type:

Attribute	Description
index_name	The name of the field that will be stored in the index. Defaults to the property/field name.
store	Set to **yes** the store actual field in the index, **no** to not store it. Defaults to **no** (note, the JSON document itself is stored, and it can be retrieved from it).
index	Set to **analyzed** for the field to be indexed and searchable after being broken down into token using an analyzer. **not_analyzed** means that its still searchable, but does not go through any analysis process or broken down into tokens. **no** means that it won't be searchable at all (as an individual field; it may still be included in **_all**). Defaults to **analyzed**.
term_vector	Possible values are **no**, **yes**, **with_offsets**, **with_positions**, **with_positions_offsets**. Defaults to **no**.
boost	The boost value. Defaults to **1.0**.
null_value	When there is a (JSON) null value for the field, use the **null_value** as the field value. Defaults to not adding the field at all.
omit_norms	Boolean value if norms should be omitted or not. Defaults to **false**.
omit_term_freq_and_positions	Boolean value if term freq and positions should be omitted. Defaults to **false**.
analyzer	The analyzer used to analyze the text contents when **analyzed** during indexing and when searching using a query string. Defaults to the globally configured analyzer.
index_analyzer	The analyzer used to analyze the text contents when **analyzed** during indexing.
search_analyzer	The analyzer used to analyze the field when part of a query string.
include_in_all	Should the field be included in the **_all** field (if enabled). Defaults to **true** or to the parent **object** type setting.

The **string** type also support custom indexing parameters associated with the indexed value. For example:

```
{
    "message" : {
        "_value":  "boosted value",
        "_boost":  2.0
    }
}
```

The mapping is required to disambiguate the meaning of the document. Otherwise, the structure would interpret "message" as a value of type "object". The key **_value** (or **value**) in the inner document specifies the real string content that should eventually be indexed. The **_boost** (or **boost**) key specifies the per field document boost (here 2.0).

Number

A number based type supporting **float**, **double**, **byte**, **short**, **integer**, and **long**. It uses specific constructs within Lucene is order to support numeric values. An example mapping can be:

```
{
    "tweet" : {
        "properties" : {
            "rank" : {
```

```
                "type" : "float",
                "null_value" : 1.0
            }
        }
    }
}
```

The following table lists all the attributes that can be used with a numbered type:

Attribute	Description
type	The type of the number. Can be **float**, **double**, **integer**, **long**, **short**, **byte**. Required.
index_name	The name of the field that will be stored in the index. Defaults to the property/field name.
store	Set to **yes** the store actual field in the index, **no** to not store it. Defaults to **no** (note, the JSON document itself is stored, and it can be retrieved from it).
index	Set to **no** if the value should not be indexed. In this case, **store** should be set to **yes**, since if its not indexed and not stored, there is nothing to do with it.
precision_step	The precision step (number of terms generated for each number value). Defaults to **4**.
boost	The boost value. Defaults to **1.0**.
null_value	When there is a (JSON) null value for the field, use the **null_value** as the field value. Defaults to not adding the field at all.
include_in_all	Should the field be included in the **_all** field (if enabled). Defaults to **true** or to the parent **object** type setting.

Date

The date type is a special type which maps to JSON string type. It follows a specific format that can be explicitly set. All dates are **UTC**. as. Internally, a date maps to a number type **long**, with the added parsing stage from string to long and from long to string. An example mapping:

```
{
    "tweet" : {
        "properties" : {
            "postDate" : {
                "type" : "date",
                "format" : "YYYY-MM-dd"
            }
        }
    }
}
```

The date type will also accept a long number representing UTC milliseconds since the epoch, regardless of the format it can handle.

The following table lists all the attributes that can be used with a date type:

Attribute	Description
index_name	The name of the field that will be stored in the index. Defaults to the property/field name.
format	The *date format*. rence-mapping-date-format>‘. Defaults to **dateOptionalTime**.
store	Set to **yes** the store actual field in the index, **no** to not store it. Defaults to **no** (note, the JSON document itself is stored, and it can be retrieved from it).
index	Set to **no** if the value should not be indexed. In this case, **store** should be set to **yes**, since if its not indexed and not stored, there is nothing to do with it.
precision_step	The precision step (number of terms generated for each number value). Defaults to **4**.
boost	The boost value. Defaults to **1.0**.
null_value	When there is a (JSON) null value for the field, use the **null_value** as the field value. Defaults to not adding the field at all.
include_in_all	Should the field be included in the **_all** field (if enabled). Defaults to **true** or to the parent **object** type setting.

Boolean

The boolean type Maps to the JSON boolean type. It ends up storing within the index either **T** or **F**, with automatic translation to **true** and **false** respectively.

```
{
    "tweet" : {
        "properties" : {
            "hes_my_special_tweet" : {
                "type" : "boolean",
            }
        }
    }
}
```

The boolean type also supports passing the value as a number (in this case **0** is **false**, all other values are **true**).

The following table lists all the attributes that can be used with the boolean type:

Attribute	Description
index_name	The name of the field that will be stored in the index. Defaults to the property/field name.
store	Set to **yes** the store actual field in the index, **no** to not store it. Defaults to **no** (note, the JSON document itself is stored, and it can be retrieved from it).
index	Set to **no** if the value should not be indexed. In this case, **store** should be set to **yes**, since if its not indexed and not stored, there is nothing to do with it.
boost	The boost value. Defaults to **1.0**.
null_value	When there is a (JSON) null value for the field, use the **null_value** as the field value. Defaults to not adding the field at all.
include_in_all	Should the field be included in the **_all** field (if enabled). Defaults to **true** or to the parent **object** type setting.

Binary

The binary type is a base64 representation of binary data that can be stored in the index. The field is always stored and not indexed at all.

```
{
    "tweet" : {
        "properties" : {
            "image" : {
                "type" : "binary",
            }
        }
    }
}
```

The following table lists all the attributes that can be used with the binary type:

Attribute	Description
index_name	The name of the field that will be stored in the index. Defaults to the property/field name.

Date Format

When defining a **date** type, or when defining **date_formats** in the **object** mapping, the value of it is the actual date format that will be used to parse the string representation of the date. There are built in formats supported, as well as complete custom one.

The parsing of dates uses Joda. The default date parsing used if no format is specified is ISODateTimeFormat.dateOptionalTimeParser().

An extension to the format allow to define several formats using || separator. This allows to define less strict formats that can be used, for example, the **yyyy/MM/dd HH:mm:ss||yyyy/MM/dd** format will parse both **yyyy/MM/dd HH:mm:ss** and **yyyy/MM/dd**. The first format will also act as the one that converts back from milliseconds to a string representation.

Date Math

The **date** type supports using date math expression when using it in a query/filter (mainly make sense in **range** query/filter).

The expression starts with an "anchor" date, which can be either **now** or a date string (in the applicable format) ending with ||. It can then follow by a math expression, supporting **+**, **-** and **/** (rounding). The units supported are **M** (month), **w** (week), **h** (hour), **m** (minute), and **s** (second).

Here are some samples: **now+1h**, **now+1h+1m**, **now+1h/d**, **2012-01-01||+1M/d**.

Note, when doing **range** type searches, and the upper value is inclusive, the rounding will properly be rounded to the ceiling instead of flooring it.

Built In Formats

The following tables lists all the defaults ISO formats supported:

Name	Description
basic_date	A basic formatter for a full date as four digit year, two digit month of year, and two digit day of
basic_date_time	A basic formatter that combines a basic date and time, separated by a 'T' (yyyyMMdd'T'HHm
basic_date_time_no_millis	A basic formatter that combines a basic date and time without millis, separated by a 'T' (yyyyM
basic_ordinal_date	A formatter for a full ordinal date, using a four digit year and three digit dayOfYear (yyyyDDD
basic_ordinal_date_time	A formatter for a full ordinal date and time, using a four digit year and three digit dayOfYear (y

Table 6.1 – continued from previous page

Name	Description
basic_ordinal_date_time_no_millis	A formatter for a full ordinal date and time without millis, using a four digit year and three digi
basic_time	A basic formatter for a two digit hour of day, two digit minute of hour, two digit second of minu
basic_time_no_millis	A basic formatter for a two digit hour of day, two digit minute of hour, two digit second of minu
basic_t_time	A basic formatter for a two digit hour of day, two digit minute of hour, two digit second of minu
basic_t_time_no_millis	A basic formatter for a two digit hour of day, two digit minute of hour, two digit second of minu
basic_week_date	A basic formatter for a full date as four digit weekyear, two digit week of weekyear, and one dig
basic_week_date_time	A basic formatter that combines a basic weekyear date and time, separated by a 'T' (xxxx'W'w
basic_week_date_time_no_millis	A basic formatter that combines a basic weekyear date and time without millis, separated by a '
date	A formatter for a full date as four digit year, two digit month of year, and two digit day of mont
date_hour	A formatter that combines a full date and two digit hour of day.
date_hour_minute	A formatter that combines a full date, two digit hour of day, and two digit minute of hour.
date_hour_minute_second	A formatter that combines a full date, two digit hour of day, two digit minute of hour, and two d
date_hour_minute_second_fraction	A formatter that combines a full date, two digit hour of day, two digit minute of hour, two digit
date_hour_minute_second_millis	A formatter that combines a full date, two digit hour of day, two digit minute of hour, two digit
date_optional_time	a generic ISO datetime parser where the date is mandatory and the time is optional.
date_time	A formatter that combines a full date and time, separated by a 'T' (yyyy-MM-dd'T'HH:mm:ss.
date_time_no_millis	A formatter that combines a full date and time without millis, separated by a 'T' (yyyy-MM-dd
hour	A formatter for a two digit hour of day.
hour_minute	A formatter for a two digit hour of day and two digit minute of hour.
hour_minute_second	A formatter for a two digit hour of day, two digit minute of hour, and two digit second of minut
hour_minute_second_fraction	A formatter for a two digit hour of day, two digit minute of hour, two digit second of minute, an
hour_minute_second_millis	A formatter for a two digit hour of day, two digit minute of hour, two digit second of minute, an
ordinal_date	A formatter for a full ordinal date, using a four digit year and three digit dayOfYear (yyyy-DDD
ordinal_date_time	A formatter for a full ordinal date and time, using a four digit year and three digit dayOfYear (y
ordinal_date_time_no_millis	A formatter for a full ordinal date and time without millis, using a four digit year and three digi
time	A formatter for a two digit hour of day, two digit minute of hour, two digit second of minute, th
time_no_millis	A formatter for a two digit hour of day, two digit minute of hour, two digit second of minute, an
t_time	A formatter for a two digit hour of day, two digit minute of hour, two digit second of minute, th
t_time_no_millis	A formatter for a two digit hour of day, two digit minute of hour, two digit second of minute, an
week_date	A formatter for a full date as four digit weekyear, two digit week of weekyear, and one digit day
week_date_time	A formatter that combines a full weekyear date and time, separated by a 'T' (xxxx-'W'ww-e'T'
weekDateTimeNoMillis	A formatter that combines a full weekyear date and time without millis, separated by a 'T' (xxx
week_year	A formatter for a four digit weekyear.
weekyearWeek	A formatter for a four digit weekyear and two digit week of weekyear.
weekyearWeekDay	A formatter for a four digit weekyear, two digit week of weekyear, and one digit day of week.
year	A formatter for a four digit year.
year_month	A formatter for a four digit year and two digit month of year.
year_month_day	A formatter for a four digit year, two digit month of year, and two digit day of month.

Custom Format

Allows for a completely customizable date format explained here.

Dynamic Mapping

Default mappings allow to automatically apply generic mapping definition to types that do not have mapping pre defined. This is mainly done thanks to the fact that the *object mapping* and namely the *root object mapping* allow for schema-less dynamic addition of unmapped fields.

The default mapping definition is plain mapping definition that is embedded within the distribution:

```
{
    "_default_" : {
    }
}
```

Pretty short, no? Basically, everything is defaulted, especially the dynamic nature of the root object mapping. The default mapping definition can be overridden in several manners. The simplest manner is to simply define a file called **default-mapping.json** and placed it under the **config** directory (which can be configured to exist in a different location). It can also be explicitly set using the **index.mapper.default_mapping_location** setting.

The dynamic creation of mappings for unmapped types can be completely disabled by setting **index.mapper.dynamic** to **false**.

As an example, here is how we can change the default *date_formats* used in the root and inner object types:

```
{
    "_default_" : {
        "date_formats" : ["yyyy-MM-dd", "dd-MM-yyyy", "date_optional_time"],
    }
}
```

Geo Point Type

Mapper type called **geo_point** to support geo based points. The declaration looks as follows:

```
{
    "pin" : {
        "properties" : {
            "location" : {
                "type" : "geo_point"
            }
        }
    }
}
```

Indexed Fields

The **geo_point** mapping will index a single field with the format of **lat,lon**. The **lat_lon** option can be set to also index the **.lat** and **.lon** as numeric fields, and **geohash** can be set to **true** to also index **.geohash** value.

A good practice is to enable indexing **lat_lon** as well, since both the geo distance and bounding box filters can either be executed using in memory checks, or using the indexed lat lon values, and it really depends on the data set which one performs better. Note though, that indexed lat lon only make sense when there is a single geo point value for the field, and not multi values.

Input Structure

The above mapping defines a **geo_point**, which accepts different formats. The following formats are supported:

Lat Lon as Properties

```
{
    "pin" : {
        "location" : {
            "lat" : 41.12,
            "lon" : -71.34
        }
    }
}
```

Lat Lon as String

Format in **lat,lon**.

```
{
    "pin" : {
        "location" : "41.12,-71.34"
    }
}
```

Geohash

```
{
    "pin" : {
        "location" : "drm3btev3e86"
    }
}
```

Lat Lon as Array

Format in **[lon, lat]**, note, the order of lon/lat here in order to conform with GeoJSON.

```
{
    "pin" : {
        "location" : [-71.34, 41.12]
    }
}
```

Mapping Options

Option	Description
lat_lon	Set to **true** to also index the **.lat** and **.lon** as fields. Defaults to **false**.
geohash	Set to **true** to also index the **.geohash** as a field. Defaults to **false**.
geohash_precision	Sets the geohash precision, defaults to 12.

Usage in Scripts

When using **doc[geo_field_name]** (in the above mapping, **doc['location']**), the **doc[...].value** returns a **GeoPoint**, which then allows access to **lat** and **lon** (for example, **doc[...].value.lat**). For performance, it is better to access the **lat** and **lon** directly using **doc[...].lat** and **doc[...].lon**.

Id Field

Each document indexed is associated with an id and a type. The **_id** field can be used to index just the id, and possible also store it. By default it is not indexed and not stored (thus, not created).

Note, even though the **_id** is not indexed, all the APIs still work (since they work with the **_uid** field), as well as fetching by ids using **term**, **term** or **prefix** queries / filters (including the specific **ids** query/filter).

The **_id** field can be enabled to be indexed, and possibly stored, using:

```
{
    "tweet" : {
        "_id" : {"index": "not_analyzed", "store" : "yes"}
    }
}
```

In order to maintain backward compatibility, a node level setting **index.mapping._id.indexed** can be set to **true** to make sure that the id is indexed when upgrading to **0.16**, though its recommended to not index the id.

The **_id** mapping can also be associated with a **path** that will be used to extract the id from a different location in the source document. For example, having the following mapping:

```
{
    "tweet" : {
        "_id" : {
            "path" : "post_id"
        }
    }
}
```

Will cause **1** to be used as the id for:

```
{
    "message" : "You know, for Search",
    "post_id" : "1"
}
```

This does require an additional lightweight parsing step while indexing, in order to extract the id in order to decide which shard the index operation will be executed on.

Index Field

The ability to store in a document the index it belongs to. By default it is disabled, in order to enable it, the following mapping should be defined:

```
{
    "tweet" : {
        "_index" : { "enabled" : true }
    }
}
```

Ip Type

An **ip** mapping type allows to store _ipv4_ addresses in a numeric form allowing to easily sort, and range query it (using ip values).

The following table lists all the attributes that can be used with an ip type:

Attribute	Description
index_name	The name of the field that will be stored in the index. Defaults to the property/field name.
store	Set to **yes** the store actual field in the index, **no** to not store it. Defaults to **no** (note, the JSON document itself is stored, and it can be retrieved from it).
index	Set to **no** if the value should not be indexed. In this case, **store** should be set to **yes**, since if its not indexed and not stored, there is nothing to do with it.
precision_step	The precision step (number of terms generated for each number value). Defaults to **4**.
boost	The boost value. Defaults to **1.0**.
null_value	When there is a (JSON) null value for the field, use the **null_value** as the field value. Defaults to not adding the field at all.
include_in_all	Should the field be included in the **_all** field (if enabled). Defaults to **true** or to the parent **object** type setting.

Meta

Each mapping can have custom meta data associated with it. These are simple storage elements that are simply persisted along with the mapping and can be retrieved when fetching the mapping definition. The meta is defined under the **_meta** element, for example:

```
{
    "tweet" : {
        "_meta" : {
            "attr1" : "value1",
            "attr2" : {
                "attr3" : "value3"
            }
        }
    }
}
```

Meta can be handy for example for client libraries that perform serialization and deserialization to store its meta model (for example, the class the document maps to).

Multi Field Type

The **multi_field** type allows to map several *core_types* of the same value. This can come very handy, for example, when wanting to map a **string** type, once when its **analyzed** and once when its **not_analyzed**. For example:

```
{
    "tweet" : {
        "properties" : {
            "name" : {
                "type" : "multi_field",
                "fields" : {
                    "name" : {"type" : "string", "index" : "analyzed"},
                    "untouched" : {"type" : "string", "index" : "not_analyzed"}
```

```
                    }
                }
            }
        }
}
```

The above example shows how the **name** field, which is of simple **string** type, gets mapped twice, once with it being **analyzed** under **name**, and once with it being **not_analyzed** under **untouched**.

Accessing Fields

With **multi_field** mapping, the field that has the same name as the property is treated as if it was mapped without a multi field. Thats the "default" field. It can be accessed regularly, for example using **name** or using typed navigation **tweet.name**.

Other fields with different names can be easily accessed using the navigation notation, for example, using: **name.untouched**, or using the typed navigation notation **tweet.name.untouched**.

Merging

When updating mapping definition using the **put_mapping** API, a core type mapping can be "upgraded" to a **multi_field** mapping. This means that if the old mapping has a plain core type mapping, the updated mapping for the same property can be a **multi_field** type, with the default field being the one being replaced.

Nested Type

Nested objects/documents allow to map certain sections in the document indexed as nested allowing to query them as if they are separate docs joining with the parent owning doc.

Note This feature is experimental and might require reindexing the data if using it.

One of the problems when indexing inner objects that occur several times in a doc is that "cross object" search match will occur, for example:

```
{
    "obj1" : [
        {
            "name" : "blue",
            "count" : 4
        },
        {
            "name" : "green",
            "count" : 6
        }
    ]
}
```

Searching for name set to blue and count higher than 5 will match the doc, because in the first element the name matches blue, and in the second element, count matches "higher than 5".

Nested mapping allow to map certain inner objects (usually multi instance ones), for example:

```
{
    "type1" : {
```

```
        "properties" : {
            "obj1" : {
                "type" : "nested"
            }
        }
    }
}
```

The above will cause all **obj1** to be indexed as a nested doc. The mapping is similar in nature to setting **type** to **object**, except that its **nested**.

The **nested** object fields can also be automatically added to the immediate parent by setting **include_in_parent** to true, and also included in the root object by setting **include_in_root** to true.

Nested docs will also automatically use the root doc **_all** field.

Searching on nested docs can be done using either the *nested query* or *nested filter*.

Internal Implementation

Internally, nested objects are indexed as additional documents, but, since they can be guaranteed to be indexed within the same "block", it allows for extremely fast joining with parent docs.

Those internal nested documents are automatically masked away when doing operations against the index (like searching with a match_all query), and they bubble out when using the nested query.

Object Type

JSON documents are hierarchal in nature, allowing to define inner "object"s within the actual JSON. ElasticSearch completely understand the nature of objects and allows to map them easily, automatically support their dynamic nature (one object can have different fields each time), and provides query support for their inner field. Lets take the following JSON as an example:

```
{
    "tweet" : {
        "person" : {
            "name" : {
                "first_name" : "Shay",
                "last_name" : "Banon"
            },
            "sid" : "12345"
        },
        "message" : "This is a tweet!"
    }
}
```

The above shows an example where a tweet includes the actual **person** details. A **person** is an object, with an **sid**, and a **name** object which has **first_name** and **last_name**. Its important to note that **tweet** is also an object, though a special *root object type* which allows for additional mapping definitions.

The following is an example of explicit mapping for the above JSON:

```
{
    "tweet" : {
        "properties" : {
            "person" : {
```

```
                "type" : "object",
                "properties" : {
                    "name" : {
                        "properties" : {
                            "first_name" : {"type" : "string"},
                            "last_name" : {"type" : "string"}
                        }
                    },
                    "sid" : {"type" : "string", "index" : "not_analyzed"}
                }
            }
            "message" : {"type" : "string"}
        }
    }
}
```

In order to mark a mapping of type **object**, set the **type** to object. This is an optional step, since if there are **properties** defined for it, it will automatically be identified as an **object** mapping.

properties

An object mapping can optionally define one or more properties using the **properties** tag. Properties list the properties this field will have. Each property can be either another **object**, or one of the *core_types*.

dynamic

One of the more important features of ElasticSearch is its ability to be schema-less. This means that, in our example above, the **person** object can later on be indexed with a new property, for example, **age**, and it will automatically be added to the mapping definitions. Same goes for the **tweet** root object.

This feature is by default turned on, and its the **dynamic** nature of each object mapped. Each object mapped is automatically dynamic, though it can be explicitly turned off:

```
{
    "tweet" : {
        "properties" : {
            "person" : {
                "type" : "object",
                "properties" : {
                    "name" : {
                        "dynamic" : false,
                        "properties" : {
                            "first_name" : {"type" : "string"},
                            "last_name" : {"type" : "string"}
                        }
                    },
                    "sid" : {"type" : "string", "index" : "not_analyzed"}
                }
            }
            "message" : {"type" : "string"}
        }
    }
}
```

In the above example, the **name** object mapped is not dynamic, meaning that if, in the future, we will try and index a JSON with a **middle_name** within the **name** object, it will get discarded and not added.

There is no performance overhead of an **object** being dynamic, the ability to turn it off is provided as a safe mechanism so "malformed" objects won't, by mistake, index data that we do not wish to be indexed.

The dynamic nature also works with inner objects, meaning that if a new **object** is provided within a mapped dynamic object, it will be automatically added to the index and mapped as well.

When processing dynamic new fields, their type is automatically derived. For example, if it is a **number**, it will automatically be treated as number *core_type*. Dynamic fields default to their default attributes, for example, they are not stored and they are always indexed.

Date fields are special since they are represented as a **string**. Date fields are detected if they can be parsed as a date when they are first introduced into the system. The set of date formats that are tested against can be configured using the **date_formats** and explained later.

Note, once a field has been added, *its type can not change*. For example, if we added age and its value is a number, then it can't be treated as a string.

The **dynamic** parameter can also be set to **strict**, meaning that not only new fields will not be introduced into the mapping, parsing (indexing) docs with such new fields will fail.

enabled

The **enabled** flag allows to disable parsing and adding a named object completely. This is handy when a portion of the JSON document passed should not be indexed. For example:

```
{
    "tweet" : {
        "properties" : {
            "person" : {
                "type" : "object",
                "properties" : {
                    "name" : {
                        "type" : "object",
                        "enabled" : false
                    },
                    "sid" : {"type" : "string", "index" : "not_analyzed"}
                }
            }
            "message" : {"type" : "string"}
        }
    }
}
```

In the above, **name** and its content will not be indexed at all.

path

In the *core_types* section, a field can have a **index_name** associated with it in order to control the name of the field that will be stored within the index. When that field exists within an object(s) that are not the root object, the name of the field of the index can either include the full "path" to the field with its **index_name**, or just the **index_name**. For example (under mapping of _type_ **person**, removed the tweet type for clarity):

```
{
    "person" : {
        "properties" : {
            "name1" : {
                "type" : "object",
                "path" : "just_name",
                "properties" : {
                    "first1" : {"type" : "string"},
                    "last1" : {"type" : "string", "index_name" : "i_last_1"}
                }
            },
            "name2" : {
                "type" : "object",
                "path" : "full",
                "properties" : {
                    "first2" : {"type" : "string"},
                    "last2" : {"type" : "string", "index_name" : "i_last_2"}
                }
            }
        }
    }
}
```

In the above example, the **name1** and **name2** objects within the **person** object have different combination of **path** and **index_name**. The document fields that will be stored in the index as a result of that are:

JSON Name	Document Field Name
name1/first1	**first1**
name1/last1	**i_last_1**
name2/first2	**name2.first2**
name2/last2	**name2.i_last_2**

Note, when querying or using a field name in any of the APIs provided (search, query, selective loading, ...), there is an automatic detection from logical full path and into the **index_name** and vice versa. For example, even though **name1/last1** defines that it is stored with **just_name** and a different **index_name**, it can either be referred to using **name1.last1** (logical name), or its actual indexed name of **i_last_1**.

More over, where applicable, for example, in queries, the full path including the type can be used such as **person.name.last1**, in this case, both the actual indexed name will be resolved to match against the index, and an automatic query filter will be added to only match **person** types.

include_in_all

include_in_all can be set on the **object** type level. When set, it propagates down to all the inner mapping defined within the **object** that do no explicitly set it.

Parent Field

The parent field mapping is defined on a child mapping, and points to the parent type this child relates to. For example, in case of a **blog** type and a **blog_tag** type child document, the mapping for **blog_tag** should be:

```
{
    "blog_tag" : {
        "_parent" : {
            "type" : "blog"
```

```
        }
    }
}
```

The mapping is automatically stored and indexed (meaning it can be searched on using the **_parent** field notation).

Root Object Type

The root object mapping is an *object type mapping* that maps the root object (the type itself). On top of all the different mappings that can be set using the *object type mapping*, it allows for additional, type level mapping definitions.

The root object mapping allows to index a JSON document that either starts with the actual mapping type, or only contains its fields. For example, the following **tweet** JSON can be indexed:

```
{
    "message" : "This is a tweet!"
}
```

But, also the following JSON can be indexed:

```
{
    "tweet" : {
        "message" : "This is a tweet!"
    }
}
```

Out of the two, it is preferable to use the document *without* the type explicitly set.

Index / Search Analyzers

The root object allows to define type mapping level analyzers for index and search that will be used with all different fields that do not explicitly set analyzers on their own. Here is an example:

```
{
    "tweet" : {
        "index_analyzer" : "standard",
        "search_analyzer" : "standard"
    }
}
```

The above simply explicitly defines both the **index_analyzer** and **search_analyzer** that will be used. There is also an option to use the **analyzer** attribute to set both the **search_analyzer** and **index_analyzer**.

dynamic_date_formats

dynamic_date_formats (old setting called **date_formats** still works) is the ability to set one or more date formats that will be used to detect **date** fields. For example:

```
{
    "tweet" : {
        "dynamic_date_formats" : ["yyyy-MM-dd", "dd-MM-yyyy"],
        "properties" : {
            "message" : {"type" : "string"}
        }
```

```
    }
}
```

In the above mapping, if a new JSON field of type string is detected, the date formats specified will be used in order to check if its a date. If it passes parsing, then the field will be declared with **date** type, and will use the matching format as its format attribute. The date format itself is explained *here*.

The default formats are: **dateOptionalTime** (ISO) and **yyyy/MM/dd HH:mm:ss Z||yyyy/MM/dd Z**.

Note: **dynamic_date_formats** are used only for dynamically added date fields, not for **date** fields that you specify in your mapping.

date_detection

Allows to disable automatic date type detection (a new field introduced and matches the provided format), for example:

```
{
    "tweet" : {
        "date_detection" : false,
        "properties" : {
            "message" : {"type" : "string"}
        }
    }
}
```

numeric_detection

Sometimes, even though json has support for native numeric types, numeric values are still provided as strings. In order to try and automatically detect numeric values from string, the **numeric_detection** can be set to **true**. For example:

```
{
    "tweet" : {
        "numeric_detection" : true,
        "properties" : {
            "message" : {"type" : "string"}
        }
    }
}
```

dynamic_templates

Dynamic templates allow to define mapping templates that will be applied when dynamic introduction of fields / objects happens.

For example, we might want to have all fields to be stored by default, or all *string* fields to be stored, or have *string* fields to always be indexed as *multi_field*, once analyzed and once not_analyzed. Here is a simple example:

```
{
    "person" : {
        "dynamic_templates" : [
            {
                "template_1" : {
```

```
                    "match" : "multi*",
                    "mapping" : {
                        "type" : "multi_field",
                        "fields" : {
                            "{name}" : {"type": "{dynamic_type}", "index" : "analyzed
↪"},
                            "org" : {"type": "{dynamic_type}", "index" : "not_analyzed
↪"}
                        }
                    }
                }
            },
            {
                "template_2" : {
                    "match" : "*",
                    "match_mapping_type" : "string",
                    "mapping" : {
                        "type" : "string",
                        "index" : "not_analyzed"
                    }
                }
            }
        ]
    }
}
```

The above mapping will create a **multi_field** mapping for all field names starting with multi, and will map all **string** types to be **not_analyzed**.

Dynamic templates are named to allow for simple merge behavior. A new mapping, just with a new template can be "put" and that template will be added, or if it has the same name, the template will be replaced.

The **match** allow to define matching on the field name. An **unmatch** option is also available to exclude fields if they do match on **match**. The **match_mapping_type** controls if this template will be applied only for dynamic fields of the specified type (as guessed by the json format).

Another option is to use **path_match**, which allows to match the dynamic template against the "full" dot notation name of the field (for example **obj1.*.value** or **obj1.obj2.***), with the respective **path_unmatch**.

The format of all the matching is simple format, allowing to use * as a matching element supporting simple patterns such as xxx*, *xxx, xxx*yyy (with arbitrary number of pattern types), as well as direct equality. The **match_pattern** can be set to **regex** to allow for regular expression based matching.

The **mapping** element provides the actual mapping definition. The **{name}** keyword can be used and will be replaced with the actual dynamic field name being introduced. The **{dynamic_type}** (or **{dynamicType}**) can be used and will be replaced with the mapping derived based on the field type (or the derived type, like **date**).

Complete generic settings can also be applied, for example, to have all mappings be stored, just set:

```
{
    "person" : {
        "dynamic_templates" : [
            {
                "store_generic" : {
                    "match" : "*",
                    "mapping" : {
                        "store" : "yes"
                    }
                }
```

```
            }
        ]
    }
}
```

Such generic templates should be placed at the end of the **dynamic_templates** list because when two or more dynamic templates match a field, only the first matching one from the list is used.

Routing Field

The routing field allows to control the **_routing** aspect when indexing data and explicit routing control is required.

store / index

The first thing the **_routing** mapping does is to store the routing value provided (**store** set to **yes**) and index it (**index** set to **not_analyzed**). The reason why the routing is stored by default is so reindexing data will be possible if the routing value is completely external and not part of the docs.

required

Another aspect of the **_routing** mapping is the ability to define it as required by setting **required** to **true**. This is very important to set when using routing features, as it allows different APIs to make use of it. For example, an index operation will be rejected if no routing value has been provided (or derived from the doc). A delete operation will be broadcasted to all shards if no routing value is provided and **_routing** is required.

path

The routing value can be provided as an external value when indexing (and still stored as part of the document, in much the same way **_source** is stored). But, it can also be automatically extracted from the index doc based on a **path**. For example, having the following mapping:

```
{
    "comment" : {
        "_routing" : {
            "required" : true,
            "path" : "blog.post_id"
        }
    }
}
```

Will cause the following doc to be routed based on the **111222** value:

```
{
    "text" : "the comment text"
    "blog" : {
        "post_id" : "111222"
    }
}
```

Note, using **path** without explicit routing value provided required an additional (though quite fast) parsing phase.

Size Field

The **_size** field allows to automatically index the size of the original **_source** indexed (not the compressed size, if compressed). By default, its disabled. In order to enable it, set the mapping to:

```
{
    "tweet" : {
        "_size" : {"enabled" : true}
    }
}
```

In order to also store it, use:

```
{
    "tweet" : {
        "_size" : {"enabled" : true, "store" : "yes"}
    }
}
```

Source Field

The **_source** field is an automatically generated field that stores the actual JSON that was used as the indexed document. It is not indexed (searchable), just stored. When executed "fetch" requests, like get or search, the **_source** field is returned by default.

Though very handy to have around, the source field does incur storage overhead within the index. For this reason, it can be disabled. For example:

```
{
    "tweet" : {
        "_source" : {"enabled" : false}
    }
}
```

Compression

The source field can be compressed (LZF) when stored in the index. This can greatly reduce the index size, as well as possibly improving performance (when decompression overhead is better than loading a bigger source from disk). The code takes special care to decompress the source only when needed, for example decompressing it directly into the REST stream of a result.

In order to enable compression, the **compress** option should be set to **true**. By default it is set to **false**. Note, this can be changed on an existing index, as a mix of compressed and uncompressed sources is supported.

Moreover, a **compress_threshold** can be set to control when the source will be compressed. It accepts a byte size value (for example **100b**, **10kb**). Note, **compress** should be set to **true**.

Includes / Excludes

Allow to specify paths in the source that would be included / excluded when its stored, supporting * as wildcard annotation. For example:

```
{
    "my_type" : {
        "_source" : {
            "includes" : ["path1.*", "path2.*"],
            "excludes" : ["pat3.*"]
        }
    }
}
```

Timestamp Field

The **_timestamp** field allows to automatically index the timestamp of a document. It can be provided externally via the index request or in the **_source**. If it is not provided externally it will be automatically set to the date the document was processed by the indexing chain.

enabled

By default it is disabled, in order to enable it, the following mapping should be defined:

```
{
    "tweet" : {
        "_timestamp" : { "enabled" : true }
    }
}
```

store / index

By default the **_timestamp** field has **store** set to **no** and **index** set to **not_analyzed**. It can be queried as a standard date field.

path

The **_timestamp** value can be provided as an external value when indexing. But, it can also be automatically extracted from the document to index based on a **path**. For example, having the following mapping:

```
{
    "tweet" : {
        "_timestamp" : {
            "enabled" : true,
            "path" : "post_date"
        }
    }
}
```

Will cause **2009-11-15T14:12:12** to be used as the timestamp value for:

```
{
    "message" : "You know, for Search",
    "post_date" : "2009-11-15T14:12:12"
}
```

Note, using **path** without explicit timestamp value provided require an additional (though quite fast) parsing phase.

format

You can define the *date format* used to parse the provided timestamp value. For example:

```
{
    "tweet" : {
        "_timestamp" : {
            "enabled" : true,
            "path" : "post_date",
            "format" : "YYYY-MM-dd"
        }
    }
}
```

Note, the default format is **dateOptionalTime**. The timestamp value will first be parsed as a number and if it fails the format will be tried.

Ttl Field

A lot of documents naturally come with an expiration date. Documents can therefore have a **_ttl** (time to live), which will cause the expired documents to be deleted automatically.

enabled

By default it is disabled, in order to enable it, the following mapping should be defined:

```
{
    "tweet" : {
        "_ttl" : { "enabled" : true }
    }
}
```

store / index

By default the **_ttl** field has **store** set to **yes** and **index** set to **not_analyzed**. Note that **index** property has to be set to **not_analyzed** in order for the purge process to work.

default

You can provide a per index/type default **_ttl** value as follows:

```
{
    "tweet" : {
        "_ttl" : { "enabled" : true, "default" : "1d" }
    }
}
```

In this case, if you don't provide a **_ttl** value in your query or in the **_source** all tweets will have a **_ttl** of one day.

If no **default** is set and no **_ttl** value is given then the document has an infinite **_ttl** and will not expire.

You can dynamically update the **default** value using the put mapping API. It won't change the **_ttl** of already indexed documents but will be used for future documents.

Note on documents expiration

Expired documents will be automatically deleted regularly. You can dynamically set the **indices.ttl.interval** to fit your needs. The default value is **60s**.

The deletion orders are processed by bulk. You can set **indices.ttl.bulk_size** to fit your needs. The default value is **10000**.

Note that the expiration procedure handle versioning properly so if a document is updated between the collection of documents to expire and the delete order, the document won't be deleted.

Type Field

Each document indexed is associated with an id and a type. The type, when indexing, is automatically indexed into a **_type** field. By default, the **_type** field is indexed (but *not* analyzed) and not stored. This means that the **_type** field can be queried.

The **_type** field can be stored as well, for example:

```
{
    "tweet" : {
        "_type" : {"store" : "yes"}
    }
}
```

The **_type** field can also not be indexed, and all the APIs will still work except for specific queries (term queries / filters) or faceting done on the **_type** field.

```
{
    "tweet" : {
        "_type" : {"index" : "no"}
    }
}
```

Uid Field

Each document indexed is associated with an id and a type, the internal **_uid** field is the unique identifier of a document within an index and is composed of the type and the id (meaning that different types can have the same id and still maintain uniqueness).

The **_uid** field is automatically used when **_type** is not indexed to perform type based filtering, and does not require the **_id** to be indexed.

Modules

Discovery

The discovery module is responsible for discovering nodes within a cluster, as well as electing a master node.

Note, ElasticSearch is a peer to peer based system, nodes communicate with one another directly if operations are delegated / broadcast. All the main APIs (index, delete, search) do not communicate with the master node. The responsibility of the master node is to maintain the global cluster state, and act if nodes join or leave the cluster by reassigning shards. Each time a cluster state is changed, the state is made known to the other nodes in the cluster (the manner depends on the actual discovery implementation).

Settings

The **cluster.name** allows to create separated clusters from one another. The default value for the cluster name is **elasticsearch**, though it is recommended to change this to reflect the logical group name of the cluster running.

Ec2

ec2 discovery allows to use the ec2 APIs to perform automatic discovery (similar to multicast in non hostile multicast environments). Here is a simple sample configuration:

```
cloud:
    aws:
        access_key: AKVAIQBF2RECL7FJWGJQ
        secret_key: vExyMThREXeRMm/b/LRzEB8jWwvzQeXgjqMX+6br

discovery:
    type: ec2
```

You'll need to install the **cloud-aws** plugin, by running **bin/plugin install cloud-aws** before (re)starting elasticsearch.

The following are a list of settings (prefixed with **discovery.ec2**) that can further control the discovery:

Setting	Description
groups	Either a comma separated list or array based list of (security) groups. Only instances with the provided security groups will be used in the cluster discovery.
host_type	The type of host type to use to communicate with other instances. Can be one of **private_ip**, **public_ip**, **private_dns**, **public_dns**. Defaults to **private_ip**.
availability_zones	Either a comma separated list or array based list of availability zones. Only instances within the provided availability zones will be used in the cluster discovery.
any_group	If set to **false**, will require all security groups to be present for the instance to be used for the discovery. Defaults to **true**.
ping_timeout	How long to wait for existing EC2 nodes to reply during discovery. Defaults to 3s.

Filtering by Tags

The ec2 discovery can also filter machines to include in the cluster based on tags (and not just groups). The settings to use include the **discovery.ec2.tag.** prefix. For example, setting **discovery.ec2.tag.stage** to **dev** will only filter instances with a tag key set to **stage**, and a value of **dev**. Several tags set will require all of those tags to be set for the instance to be included.

One practical use for tag filtering is when an ec2 cluster contains many nodes that are not running elasticsearch. In this case (particularly with high **ping_timeout** values) there is a risk that a new node's discovery phase will end before it has found the cluster (which will result in it declaring itself master of a new cluster with the same name - highly undesirable). Tagging elasticsearch ec2 nodes and then filtering by that tag will resolve this issue.

Region

The **cloud.aws.region** can be set to a region and will automatically use the relevant settings for both **ec2** and **s3**. The available values are: **us-east-1**, **us-west-1**, **ap-southeast-1**, **eu-west-1**.

Automatic Node Attributes

Though not dependent on actually using **ec2** as discovery (but still requires the cloud aws plugin installed), the plugin can automatically add node attributes relating to ec2 (for example, availability zone, that can be used with the awareness allocation feature). In order to enable it, set **cloud.node.auto_attributes** to **true** in the settings.

Zen

The zen discovery is the built in discovery module for elasticsearch and the default. It provides both multicast and unicast discovery as well being easily extended to support cloud environments.

The zen discovery is integrated with other modules, for example, all communication between nodes is done using the *transport* module.

It is separated into several sub modules, which are explained below:

Ping

This is the process where a node uses the discovery mechanisms to find other nodes. There is support for both multicast and unicast based discovery (can be used in conjunction as well).

Multicast

Multicast ping discovery of other nodes is done by sending one or more multicast requests where existing nodes that exists will receive and respond to. It provides the following settings with the **discovery.zen.ping.multicast** prefix:

Setting	Description
group	The group address to use. Defaults to **224.2.2.4**.
port	The port to use. Defaults to **54328**.
ttl	The ttl of the multicast message. Defaults to **3**.
address	The address to bind to, defaults to **null** which means it will bind to all available network interfaces.

Multicast can be disabled by setting **multicast.enabled** to **false**.

Unicast

The unicast discovery allows to perform the discovery when multicast is not enabled. It basically requires a list of hosts to use that will act as gossip routers. It provides the following settings with the **discovery.zen.ping.unicast** prefix:

Setting	Description
hosts	Either an array setting or a comma delimited setting. Each value is either in the form of **host:port**, or in the form of **host[port1-port2]**.

The unicast discovery uses the *transport* module to perform the discovery.

Master Election

As part of the initial ping process a master of the cluster is either elected or joined to. This is done automatically. The **discovery.zen.ping_timeout** (which defaults to **3s**) allows to configure the election to handle cases of slow or

congested networks (higher values assure less chance of failure). Note, this setting was changed from 0.15.1 onwards, prior it was called **discovery.zen.initial_ping_timeout**.

Nodes can be excluded from becoming a master by setting **node.master** to **false**. Note, once a node is a client node (**node.client** set to **true**), it will not be allowed to become a master (**node.master** is automatically set to **false**).

The **discovery.zen.minimum_master_nodes** allows to control the minimum number of master eligible nodes a node should "see" in order to operate within the cluster. Its recommended to set it to a higher value than 1 when running more than 2 nodes in the cluster.

Fault Detection

There are two fault detection processes running. The first is by the master, to ping all the other nodes in the cluster and verify that they are alive. And on the other end, each node pings to master to verify if its still alive or an election process needs to be initiated.

The following settings control the fault detection process using the **discovery.zen.fd** prefix:

Setting	Description
ping_interval	How often a node gets pinged. Defaults to **1s**.
ping_timeout	How long to wait for a ping response, defaults to **30s**.
ping_retries	How many ping failures / timeouts cause a node to be considered failed. Defaults to **3**.

External Multicast

The multicast discovery also supports external multicast requests to discover nodes. The external client can send a request to the multicast IP/group and port, in the form of:

```
{
    "request" : {
        "cluster_name": "test_cluster"
    }
}
```

And the response will be similar to node info response (with node level information only, including transport/http addresses, and node attributes):

```
{
    "response" : {
        "cluster_name" : "test_cluster",
        "transport_address" : "...",
        "http_address" : "...",
        "attributes" : {
            "..."
        }
    }
}
```

Note, it can still be enabled, with disabled internal multicast discovery, but still have external discovery working by keeping **discovery.zen.ping.multicast.enabled** set to **true** (the default), but, setting **discovery.zen.ping.multicast.ping.enabled** to **false**.

Gateway

The gateway module allows one to store the state of the cluster meta data across full cluster restarts. The cluster meta data mainly holds all the indices created with their respective (index level) settings and explicit type mappings.

Each time the cluster meta data changes (for example, when an index is added or deleted), those changes will be persisted using the gateway. When the cluster first starts up, the state will be read from the gateway and applied.

The gateway set on the node level will automatically control the index gateway that will be used. For example, if the **fs** gateway is used, then automatically, each index created on the node will also use its own respective index level **fs** gateway. In this case, if an index should not persist its state, it should be explicitly set to **none** (which is the only other value it can be set to).

The default gateway used is the *local* gateway.

Recovery After Nodes / Time

In many cases, the actual cluster meta data should only be recovered after specific nodes have started in the cluster, or a timeout has passed. This is handy when restarting the cluster, and each node local index storage still exists to be reused and not recovered from the gateway (which reduces the time it takes to recover from the gateway).

The **gateway.recover_after_nodes** setting (which accepts a number) controls after how many nodes within the cluster recovery will start. The **gateway.recover_after_time** setting (which accepts a time value) sets the time to wait till recovery happens once the nodes are met.

The **gateway.expected_nodes** allows to set how many nodes are expected to be in the cluster, and once met, the *recover_after_time* is ignored and recovery starts. For example setting:

```
gateway:
    recover_after_nodes: 1
    recover_after_time: 5m
    expected_nodes: 2
```

In an expected 2 nodes cluster will cause recovery to start 5 minutes after the first node is up, but once there are 2 nodes in the cluster, recovery will begin immediately (without waiting).

Note, once the meta data has been recovered from the gateway (which indices to create, mappings and so on), then this setting is no longer effective until the next full restart of the cluster.

Operations are blocked while the cluster meta data has not been recovered in order not to mix with the actual cluster meta data that will be recovered once the settings has been reached.

Fs

The file system based gateway stores the cluster meta data and indices in a *shared* file system. Note, since its a distributed system, the file system should be shared between all different nodes. Here is an example config to enable it:

```
gateway:
    type: fs
```

location

The location where the gateway stores the cluster state can be set using the **gateway.fs.location** setting. By default, it will be stored under the **work** directory. Note, the **work** directory is considered a temporal directory with ElasticSearch

(meaning it is safe to **rm -rf** it), the default location of the persistent gateway in work intentional, *it should be changed*.

When explicitly specifying the **gateway.fs.location**, each node will append its **cluster.name** to the provided location. It means that the location provided can safely support several clusters.

concurrent_streams

The **gateway.fs.concurrent_streams** allow to throttle the number of streams (per node) opened against the shared gateway performing the snapshot operation. It defaults to **5**.

Hadoop

The hadoop (HDFS) based gateway stores the cluster meta and indices data in hadoop. Hadoop support is provided as a plugin and installing is explained here or downloading the hadoop plugin and placing it under the **plugins** directory. Here is an example config to enable it:

```
gateway:
    type: hdfs
    hdfs:
        uri: hdfs://myhost:8022
```

Settings

The hadoop gateway requires two simple settings. The **gateway.hdfs.uri** controls the URI to connect to the hadoop cluster, for example: **hdfs://myhost:8022**. The **gateway.hdfs.path** controls the path under which the gateway will store the data.

concurrent_streams

The **gateway.hdfs.concurrent_streams** allow to throttle the number of streams (per node) opened against the shared gateway performing the snapshot operation. It defaults to **5**.

Local

The local gateway allows for recovery of the full cluster state and indices from the local storage of each node, and does not require a common node level shared storage.

In order to use the local gateway, the indices must be file system based with no memory caching.

Note, different from shared gateway types, the persistency to the local gateway is *not* done in an async manner. Once an operation is performed, the data is there for the local gateway to recover it in case of full cluster failure.

It is important to configure the **gateway.recover_after_nodes** setting to include most of the expected nodes to be started after a full cluster restart. This will insure that the latest cluster state is recovered. For example:

```
gateway:
    recover_after_nodes: 1
    recover_after_time: 5m
    expected_nodes: 2
```

Note, to backup/snapshot the full cluster state it is recommended that the local storage for all nodes be copied (in theory not all are required, just enough to guarantee a copy of each shard has been copied, ie depending on the replication settings) while disabling flush. Shared storage such as S3 can be used to keep the different nodes' copies in one place, though it does comes at a price of more IO.

S3

s3 based gateway allows to do long term reliable async persistency of the cluster state and indices directly to Amazon s3. Here is how it can be configured:

```
cloud:
    aws:
        access_key: AKVAIQBF2RECL7FJWGJQ
        secret_key: vExyMThREXeRMm/b/LRzEB8jWwvzQeXgjqMX+6br


gateway:
    type: s3
    s3:
        bucket: bucket_name
```

You'll need to install the **cloud-aws** plugin, by running **bin/plugin install cloud-aws** before (re)starting elasticsearch.

The following are a list of settings (prefixed with **gateway.s3**) that can further control the s3 gateway:

Setting	Description
chunk_size	Big files are broken down into chunks (to overcome AWS 5g limit and use concurrent snapshotting). Default set to **100m**.

concurrent_streams

The **gateway.s3.concurrent_streams** allow to throttle the number of streams (per node) opened against the shared gateway performing the snapshot operation. It defaults to **5**.

Region

The **cloud.aws.region** can be set to a region and will automatically use the relevant settings for both **ec2** and **s3**. The available values are: **us-east-1**, **us-west-1**, **ap-southeast-1**, **eu-west-1**.

Cluster

Shards Allocation

Shards allocation is the process of allocating shards to nodes. This can happen during initial recovery, replica allocation, rebalancing, or handling nodes being added or removed.

The following settings allow to control it. Note **c.r.a** is an abbreviation for **cluster.routing.allocation**, so make sure to use the full form in the configuration.

Setting	Description
c.r.a.allow_rebalance	Allow to control when rebalancing will happen based on the total state of all the indices shards in the cluster. **always**, **indices_primaries_active**, and **indices_all_active** are allowed, defaulting to **indices_all_active** to reduce chatter during initial recovery.
c.r.a.cluster_concurrent_rebalance	Allow to control how many concurrent rebalancing of shards are allowed cluster wide, and default it to **2**.
c.r.a.node_initial_primaries_recoveries	Allow to control specifically the number of initial recoveries of primaries that are allowed per node. Since most times local gateway is used, those should be fast and we can handle more of those per node without creating load.
c.r.a.node_concurrent_recoveries	How many concurrent recoveries are allowed to happen on a node. Defaults to **2**.
c.r.a.disable_allocation	Allows to disable either primary or replica allocation. Note, a replica will still be promoted to primary if one does not exists. This setting really make sense when dynamically updating it using the cluster update settings API.
c.r.a.disable_replica_allocation	Allow to disable only replica allocation. Similar to the previous setting, mainly make sense when using it dynamically using the cluster update settings API.
indices.recovery.concurrent_streams	The number of streams to open (on a *node* level) to recover a shard from a peer shard. Default is **5**.

Shard Allocation Awareness

Cluster allocation awareness allows to configure shard and replicas allocation across generic attributes associated the nodes. Lets explain it through an example:

Assume we have several racks. When we start a node, we can configure an attribute called **rack_id** (any attribute name works), for example, here is a sample config:

<pre> node.rack_id: rack_one

The above sets an attribute called **rack_id** for the relevant node with a value of **rack_one**. Now, we need to configure the **rack_id** attribute as one of the awareness allocation attributes (set it on *all* (master eligible) nodes config):

<pre> cluster.routing.allocation.awareness.attributes: rack_id

The above will mean that the **rack_id** attribute will be used to do awareness based allocation of shard and its replicas. For example, lets say we start 2 nodes with **node.rack_id** set to **rack_one**, and deploy a single index with 5 shards and 1 replica. The index will be fully deployed on the current nodes (5 shards and 1 replica each, total of 10 shards).

Now, if we start two more nodes, with **node.rack_id** set to **rack_two**, shards will relocate to even the number of shards across the nodes, but, a shard and its replica will not be allocated in the same **rack_id** value.

The awareness attributes can hold several values, for example:

<pre> cluster.routing.allocation.awareness.attributes: rack_id,zone

NOTE: When using awareness attributes, shards will not be allocated to nodes that don't have values set for those attributes.

Forced Awareness

Sometimes, we know in advance the number of values an awareness attribute can have, and more over, we would like never to have more replicas then needed allocated on a specific group of nodes with the same awareness attribute value. For that, we can force awareness on specific attributes.

For example, lets say we have an awareness attribute called **zone**, and we know we are going to have two zones, **zone1** and **zone2**. Here is how we can force awareness one a node:

<pre> cluster.routing.allocation.awareness.force.zone.values: zone1,zone2 cluster.routing.allocation.awareness.attributes: zone

Now, lets say we start 2 nodes with **node.zone** set to **zone1** and create an index with 5 shards and 1 replica. The index will be created, but only 5 shards will be allocated (with no replicas). Only when we start more shards with **node.zone** set to **zone2** will the replicas be allocated.

Automatic Preference When Searching / GETing

When executing a search, or doing a get, the node receiving the request will prefer to execute the request on shards that exists on nodes that have the same attribute values as the executing node.

Realtime Settings Update

The settings can be updated using the cluster update settings API on a live cluster.

Shard Allocation Filtering

Allow to control allocation if indices on nodes based on include/exclude filters. The filters can be set both on the index level and on the cluster level. Lets start with an example of setting it on the cluster level:

Lets say we have 4 nodes, each has specific attribute called **tag** associated with it (the name of the attribute can be any name). Each node has a specific value associated with **tag**. Node 1 has a setting **node.tag: value1**, Node 2 a setting of **node.tag: value2**, and so on.

We can create an index that will only deploy on nodes that have **tag** set to **value1** and **value2** by setting **index.routing.allocation.include.tag** to **value1,value2**. For example:

```
curl -XPUT localhost:9200/test/_settings -d '{
    "index.routing.allocation.include.tag" : "value1,value2"
}'
```

On the other hand, we can create an index that will be deployed on all nodes except for nodes with a **tag** of value **value3** by setting **index.routing.allocation.exclude.tag** to **value3**. For example:

```
curl -XPUT localhost:9200/test/_settings -d '{
    "index.routing.allocation.exclude.tag" : "value3"
}'
```

The **include** and **exclude** values can have generic simple matching wildcards, for example, **value1***. A special attribute name called **_ip** can be used to match on node ip values.

Obviously a node can have several attributes associated with it, and both the attribute name and value are controlled in the setting. For example, here is a sample of several node configurations:

```
node.group1: group1_value1
node.group2: group2_value4
```

In the same manner, **include** and **exclude** can work against several attributes, for example:

```
curl -XPUT localhost:9200/test/_settings -d '{
    "index.routing.allocation.include.group1" : "xxx"
    "index.routing.allocation.include.group2" : "yyy",
    "index.routing.allocation.exclude.group3" : "zzz",
}'
```

The provided settings can also be updated in real time using the update settings API, allowing to "move" indices (shards) around in realtime.

Cluster wide filtering can also be defined, and be updated in real time using the cluster update settings API. This setting can come in handy for things like decommissioning nodes (even if the replica count is set to 0). Here is a sample of how to decommission a node based on **_ip** address:

```
curl -XPUT localhost:9200/_cluster/settings -d '{
    "transient" : {
        "cluster.routing.allocation.exclude._ip" : "10.0.0.1"
    }
}'
```

Http

The http module allows to expose *elasticsearch API* over HTTP.

The http mechanism is completely asynchronous in nature, meaning that there is no blocking thread waiting for a response. The benefit of using asynchronous communication for HTTP is solving the C10k problem.

When possible, consider using HTTP keep alive when connecting for better performance and try to get your favorite client not to do HTTP chunking.

Settings

The following are the settings the can be configured for HTTP:

Setting	Description
http.port	A bind port range. Defaults to **9200-9300**.
http.max_content_length	The max content of an HTTP request. Defaults to **100mb**
http.compression	Support for compression when possible (with Accept-Encoding). Defaults to **false**.
http.compression_level	Defines the compression level to use. Defaults to **6**.

It also shares the uses the common *network settings*.

Disable HTTP

The http module can be completely disabled and not started by setting **http.enabled** to **false**. This make sense when creating non *data nodes* which accept HTTP requests, and communicate with data nodes using the internal *transport*.

Indices

The indices module allow to control settings that are globally managed for all indices.

Indexing Buffer

The indexing buffer setting allows to control how much memory will be allocated for the indexing process. It is a global setting that bubbles down to all the different shards allocated on a specific node.

The **indices.memory.index_buffer_size** accepts either a percentage or a byte size value. It defaults to **10%**, meaning that **10%** of the total memory allocated to a node will be used as the indexing buffer size. This amount is then divided between all the different shards. Also, if percentage is used, allow to set **min_index_buffer_size** (defaults to **48mb**) and **max_index_buffer_size** which by default is unbounded.

The **indices.memory.min_shard_index_buffer_size** allows to set a hard lower limit for the memory allocated per shard for its own indexing buffer. It defaults to **4mb**.

Jmx

The JMX module exposes node information through JMX. JMX can be used by either jconsole or VisualVM.

Exposed JMX data include both node level information, as well as instantiated index and shard on specific node. This is a work in progress with each version exposing more information.

jmx.domain

The domain under which the JMX will register under can be set using **jmx.domain** setting. It defaults to **{elasticsearch}**.

jmx.create_connector

An RMI connector can be started to accept JMX requests. This can be enabled by setting **jmx.create_connector** to **true**. An RMI connector does come with its own overhead, make sure you really need it.

When an RMI connector is created, the **jmx.port** setting provides a port range setting for the ports the rmi connector can open on. By default, it is set to **9400-9500**.

Memcached

The memcached module allows to expose *elasticsearch API* over the memcached protocol (as closely as possible).

It is provided as a plugin called **transport-memcached** and installing is explained here . Another option is to download the memcached plugin and placing it under the **plugins** directory.

The memcached protocol supports both the binary and the text protocol, automatically detecting the correct one to use.

Mapping REST to Memcached Protocol

Memcached commands are mapped to REST and handled by the same generic REST layer in elasticsearch. Here is a list of the memcached commands supported:

GET

The memcached **GET** command maps to a REST **GET**. The key used is the URI (with parameters). The main downside is the fact that the memcached **GET** does not allow body in the request (and **SET** does not allow to return a result...). For this reason, most REST APIs (like search) allow to accept the "source" as a URI parameter as well.

SET

The memcached **SET** command maps to a REST **POST**. The key used is the URI (with parameters), and the body maps to the REST body.

DELETE

The memcached **DELETE** command maps to a REST **DELETE**. The key used is the URI (with parameters).

QUIT

The memcached **QUIT** command is supported and disconnects the client.

Settings

The following are the settings the can be configured for memcached:

Setting	Description
memcached.port	A bind port range. Defaults to **11211-11311**.

It also shares the uses the common *network settings*.

Disable memcached

The memcached module can be completely disabled and not started using by setting **memcached.enabled** to **false**. By default it is enabled once it is detected as a plugin.

Network

There are several modules within a Node that use network based configuration, for example, the *transport* and *http* modules. Node level network settings allows to set common settings that will be shared among all network based modules (unless explicitly overridden in each module).

The **network.bind_host** setting allows to control the host different network components will bind on. By default, the bind host will be **anyLocalAddress** (typically **0.0.0.0** or **::0**).

The **network.publish_host** setting allows to control the host the node will publish itself within the cluster so other nodes will be able to connect to it. Of course, this can't be the **anyLocalAddress**, and by default, it will be the first non loopback address (if possible), or the local address.

The **network.host** setting is a simple setting to automatically set both **network.bind_host** and **network.publish_host** to the same host value.

Both settings allows to be configured with either explicit host address or host name. The settings also accept logical setting values explained in the following table:

Logical Host Setting Value	Description
local	Will be resolved to the local ip address.
_non_loopback_	The first non loopback address.
_non_loopback:ipv4_	The first non loopback IPv4 address.
_non_loopback:ipv6_	The first non loopback IPv6 address.
[networkInterface]	Resolves to the ip address of the provided network interface. For example **_en0_**.
[networkInterface]:ipv4	Resolves to the ipv4 address of the provided network interface. For example **_en0:ipv4_**.
[networkInterface]:ipv6	Resolves to the ipv6 address of the provided network interface. For example **_en0:ipv6_**.

When the **cloud-aws** plugin is installed, the following are also allowed as valid network host settings:

EC2 Host Value	Description
ec2:privateIpv4	The private IP address (ipv4) of the machine.
ec2:privateDns	The private host of the machine.
ec2:publicIpv4	The public IP address (ipv4) of the machine.
ec2:publicDns	The public host of the machine.
_ec2_	Less verbose option for the private ip address.
ec2:privateIp	Less verbose option for the private ip address.
ec2:publicIp	Less verbose option for the public ip address.

TCP Settings

Any component that uses TCP (like the HTTP, Transport and Memcached) share the following allowed settings:

Setting	Description
network.tcp.no_delay	Enable or disable tcp no delay setting. Defaults to **true**.
network.tcp.keep_alive	Enable or disable tcp keep alive. By default not explicitly set.
network.tcp.reuse_address	Should an address be reused or not. Defaults to **true** on none windows machines.
network.tcp.send_buffer_size	The size of the tcp send buffer size (in size setting format). By default not explicitly set.
network.tcp.receive_buffer_size	The size of the tcp receive buffer size (in size setting format). By default not explicitly set.

Node

elasticsearch allows to configure a node to either be allowed to store data locally or not. Storing data locally basically means that shards of different indices are allowed to be allocated on that node. By default, each node is considered to be a data node, and it can be turned off by setting **node.data** to **false**.

This is a powerful setting allowing to simply create smart load balancers that take part in some of different API processing. Lets take an example:

We can start a whole cluster of data nodes which do not even start an HTTP transport by setting **http.enabled** to **false**. Such nodes will communicate with one another using the *transport* module. In front of the cluster we can start one or more "non data" nodes which will start with HTTP enabled. All HTTP communication will be performed through these "non data" nodes.

The benefit of using that is first the ability to create smart load balancers. These "non data" nodes are still part of the cluster, and they redirect operations exactly to the node that holds the relevant data. The other benefit is the fact that for scatter / gather based operations (such as search), these nodes will take part of the processing since they will start the scatter process, and perform the actual gather processing.

This relieves the data nodes to do the heavy duty of indexing and searching, without needing to process HTTP requests (parsing), overload the network, or perform the gather processing.

Plugins

Plugins

Plugins are a way to enhance the basic elasticsearch functionality in a custom manner. They range from adding custom mapping types, custom analyzers (in a more built in fashion), native scripts, custom discovery and more.

Installing plugins

Installing plugins can either be done manually by placing them under the **plugins** directory, or using the **plugin** script. Several plugins can be found under the elasticsearch organization in GitHub, starting with **elasticsearch-**.

Plugins can also be automatically downloaded and installed from gitub using: **user_name/repo_name** structure, or, for explicit versions, using **user_name/repo_name/version_number**. When no version number is specified, first a version based on the elasticsearch version is tried, and if it does not work, then master is used.

Site Plugins

Plugins can have "sites" in them, any plugin that exists under the **plugins** directory with a **_site** directory, its content will be statically served when hitting **/_plugin/[plugin_name]/** url. Those can be added even after the process has started.

Installed plugins that do not contain any java related content, will automatically be detected as site plugins, and their content will be moved under **_site**.

The ability to install plugins from github allows to easily install site plugins hosted there, for example, running:

```
bin/plugin -install Aconex/elasticsearch-head
bin/plugin -install lukas-vlcek/bigdesk
```

Will install both of those site plugins, with **elasticsearch-head** available under **http://localhost:9200/_plugin/head/** and **bigdesk** available under **http://localhost:9200/_plugin/bigdesk/**.

Mandatory Plugins

If you rely on some plugins, you can define mandatory plugins using the **plugin.mandatory** attribute, for example, here is a sample config:

```
plugin.mandatory: mapper-attachments,lang-groovy
```

For safety reasons, if a mandatory plugin is not installed, the node will not start.

Known Plugins

Analysis Plugins

- Smart Chinese Analysis Plugin (by elasticsearch team)
- ICU Analysis plugin (by elasticsearch team)
- Stempel (Polish) Analysis plugin (by elasticsearch team)
- IK Analysis Plugin (by Medcl)
- Mmseg Analysis Plugin (by Medcl)
- Hunspell Analysis Plugin (by Jörg Prante)
- Japanese (Kuromoji) Analysis plugin (by elasticsearch team).
- Japanese Analysis plugin (by suguru).
- Russian and English Morphological Analysis Plugin (by Igor Motov)

- Pinyin Analysis Plugin (by Medcl)

River Plugins

- CouchDB River Plugin (by elasticsearch team)
- Wikipedia River Plugin (by elasticsearch team)
- Twitter River Plugin (by elasticsearch team)
- RabbitMQ River Plugin (by elasticsearch team)
- RSS River Plugin (by David Pilato)
- MongoDB River Plugin (by Richard Louapre)
- Open Archives Initiative (OAI) River Plugin (by Jörg Prante)
- St9 River Plugin (by Sunny Gleason)
- Sofa River Plugin (by adamlofts)
- Amazon SQS River Plugin (by Alex B)
- JDBC River Plugin (by Jörg Prante)
- FileSystem River Plugin (by David Pilato)

Transport Plugins

- Servlet transport (by elasticsearch team)
- Memcached transport plugin (by elasticsearch team)
- Thrift Transport (by elasticsearch team)
- ZeroMQ transport layer plugin (by Tanguy Leroux)
- Jetty HTTP transport plugin (by Sonian Inc.)

Scripting Plugins

- Python language Plugin (by elasticsearch team)
- JavaScript language Plugin (by elasticsearch team)
- Groovy lang Plugin (by elasticsearch team)

Site Plugins

- BigDesk Plugin (by Lukáš Vlček)
- Elasticsearch Head Plugin (by Ben Birch)

Misc Plugins

- Mapper Attachments Type plugin (by elasticsearch team)
- Hadoop Plugin (by elasticsearch team)
- AWS Cloud Plugin (by elasticsearch team)
- ElasticSearch Mock Solr Plugin (by Matt Weber)
- Suggester Plugin (by Alexander Reelsen)
- ElasticSearch PartialUpdate Plugin (by Medcl)
- ZooKeeper Discovery Plugin (by Sonian Inc.)

Scripting

The scripting module allows to use scripts in order to evaluate custom expressions. For example, scripts can be used to return "script fields" as part of a search request, or can be used to evaluate a custom score for a query and so on.

The scripting module uses by default mvel as the scripting language with some extensions. mvel is used since its extremely fast and very simple to use, and in most cases, simple expressions are needed (for example, mathematical equations).

Additional **lang** plugins are provided to allow to execute scripts in different languages. Currently supported plugins are **lang-javascript** for JavaScript, **lang-groovy** for Groovy, and **lang-python** for Python. All places where a *script* parameter can be used, a **lang** parameter (on the same level) can be provided to define the language of the script. The **lang** options are **mvel**, **js**, **groovy**, **python**, and **native**.

Default Scripting Language

The default scripting language (assuming no **lang** parameter is provided) is **mvel**. In order to change it set the **script.default_lang** to the appropriate language.

Preloaded Scripts

Scripts can always be provided as part of the relevant API, but they can also be preloaded by placing them under *config/scripts* and then referencing them by the script name (instead of providing the full script). This helps reduce the amount of data passed between the client and the nodes.

The name of the script is derived from the hierarchy of directories it exists under, and the file name without the lang extension. For example, a script placed under *config/scripts/group1/group2/test.py* will be named *group1_group2_test*.

Native (Java) Scripts

Even though **mvel** is pretty fast, allow to register native Java based scripts for faster execution.

In order to allow for scripts, the **NativeScriptFactory** needs to be implemented that constructs the script that will be executed. There are two main types, one that extends **AbstractExecutableScript** and one that extends **AbstractSearchScript** (probably the one most users will extend, with additional helper classes in **AbstractLongSearchScript**, **AbstractDoubleSearchScript**, and **AbstractFloatSearchScript**).

Registering them can either be done by settings, for example: **script.native.my.type** set to **sample.MyNativeScriptFactory** will register a script named **my**. Another option is in a plugin, access **ScriptModule** and call **registerScript** on it.

Executing the script is done by specifying the **lang** as **native**, and the name of the script as the **script**.

Note, the scripts need to be in the classpath of elasticsearch. One simple way to do it is to create a directory under plugins (choose a descriptive name), and place the jar / classes files there, they will be automatically loaded.

Score

In all scripts that can be used in facets, allow to access the current doc score using **doc.score**.

Document Fields

Most scripting revolve around the use of specific document fields data. The **doc['field_name']** can be used to access specific field data within a document (the document in question is usually derived by the context the script is used). Document fields are very fast to access since they end up being loaded into memory (all the relevant field values/tokens are loaded to memory).

The following data can be extracted from a field:

Expression	Description
doc['field_name'].value	The native value of the field. For example, if its a short type, it will be short.
doc['field_name'].values	The native array values of the field. For example, if its a short type, it will be short[]. Remember, a field can have several values within a single doc. Returns an empty array if the field has no values.
doc['field_name'].stringValue	The string value of the field.
doc['field_name'].doubleValue	The converted double of the field. Replace *double* with *int*, *long*, *float*, *short*, *byte* for the respective values.
doc['field_name'].doubleValues	The converted double values array.
doc['field_name'].date	Applies only to date / long (timestamp) types, returns a MutableDateTime allowing to get date / time specific data. For example: **doc['field_name'].date.minuteOfHour**
doc['field_name'].dates	Return an array of date values for the field.
doc['field_name'].empty	A boolean indicating if the field has no values within the doc.
doc['field_name'].multiValued	A boolean indicating that the field has several values within the corpus.
doc['field_name'].lat	The latitude of a geo point type.
doc['field_name'].lon	The longitude of a geo point type.
doc['field_name'].lats	The latitudes of a geo point type.
doc['field_name'].lons	The longitudes of a geo point type.
doc['field_name'].distance(lat, lon)	The distance (in miles) of this geo point field from the provided lat/lon.
doc['field_name'].distanceInKm(lat, lon)	The distance (in km) of this geo point field from the provided lat/lon.
doc['field_name'].geohashDistance(geohash)	The distance (in miles) of this geo point field from the provided geohash.
doc['field_name'].geohashDistanceInKm(geohash)	The distance (in km) of this geo point field from the provided geohash.

Stored Fields

Stored fields can also be accessed when executed a script. Note, they are much slower to access compared with document fields, but are not loaded into memory. They can be simply accessed using **_fields['my_field_name'].value** or **_fields['my_field_name'].values**.

Source Field

The source field can also be accessed when executing a script. The source field is loaded per doc, parsed, and then provided to the script for evaluation. The **_source** forms the context under which the source field can be accessed, for example **_source.obj2.obj1.field3**.

mvel Built In Functions

There are several built in functions that can be used within scripts. They include:

Function	Description
time()	The current time in milliseconds.
sin(a)	Returns the trigonometric sine of an angle.
cos(a)	Returns the trigonometric cosine of an angle.
tan(a)	Returns the trigonometric tangent of an angle.
asin(a)	Returns the arc sine of a value.
acos(a)	Returns the arc cosine of a value.
atan(a)	Returns the arc tangent of a value.
toRadians(angdeg)	Converts an angle measured in degrees to an approximately equivalent angle measured in radians
toDegrees(angrad)	Converts an angle measured in radians to an approximately equivalent angle measured in degrees.
exp(a)	Returns Euler's number _e_ raised to the power of value.
log(a)	Returns the natural logarithm (base _e_) of a value.
log10(a)	Returns the base 10 logarithm of a value.
sqrt(a)	Returns the correctly rounded positive square root of a value.
cbrt(a)	Returns the cube root of a double value.
IEEEremainder(f1, f2)	Computes the remainder operation on two arguments as prescribed by the IEEE 754 standard.
ceil(a)	Returns the smallest (closest to negative infinity) value that is greater than or equal to the argument and is e
floor(a)	Returns the largest (closest to positive infinity) value that is less than or equal to the argument and is equal
rint(a)	Returns the value that is closest in value to the argument and is equal to a mathematical integer.
atan2(y, x)	Returns the angle theta from the conversion of rectangular coordinates (_x_, _y_) to polar coordina
pow(a, b)	Returns the value of the first argument raised to the power of the second argument.
round(a)	Returns the closest _int_ to the argument.
random()	Returns a random _double_ value.
abs(a)	Returns the absolute value of a value.
max(a, b)	Returns the greater of two values.
min(a, b)	Returns the smaller of two values.
ulp(d)	Returns the size of an ulp of the argument.
signum(d)	Returns the signum function of the argument.
sinh(x)	Returns the hyperbolic sine of a value.
cosh(x)	Returns the hyperbolic cosine of a value.
tanh(x)	eturns the hyperbolic tangent of a value.
hypot(x, y)	Returns sqrt(_x^2_ + _y^2_) without intermediate overflow or underflow.

Threadpool

A node holds several thread pools in order to improve how threads are managed and memory consumption within a node. There are several thread pools, but the important ones include:

- **index**: For index/delete operations (defaults to **cached** type).
- **search**: For get/count/search operations (defaults to **cached** type).

- **bulk**: For bulk operations (defaults to **cached** type).
- **refresh**: For refresh operations (defaults to **cached** type).

Changing a specific thread pool can be done by setting its type and specific type parameters, for example, changing the **index** thread pool to **blocking** type:

```
threadpool:
    index:
        type: blocking
        min: 1
        size: 30
        wait_time: 30s
```

The following are the types of thread pools that can be used and their respective parameters:

cache

The **cache** thread pool is an unbounded thread pool that will spawn a thread if there are pending requests. Here is an example of how to set it:

```
threadpool:
    index:
        type: cached
```

fixed

The **fixed** thread pool holds a fixed size of threads to handle the requests with a queue (optionally bounded) for pending requests that have no threads to service them.

The **size** parameter controls the number of threads, and defaults to the number of cores times 5.

The **queue_size** allows to control the size of the queue of pending requests that have no threads to execute them. By default, it is set to **-1** which means its unbounded. When a request comes in and the queue is full, the **reject_policy** parameter can control how it will behave. The default, **abort**, will simply fail the request. Setting it to **caller** will cause the request to execute on an IO thread allowing to throttle the execution on the networking layer.

```
threadpool:
    index:
        type: fixed
        size: 30
        queue: 1000
        reject_policy: caller
```

blocking

The **blocking** pool allows to configure a **min** (defaults to **1**) and **size** (defaults to the number of cores times 5) parameters for the number of threads.

It also has a backlog queue with a default **queue_size** of **1000**. Once the queue is full, it will wait for the provided **wait_time** (defaults to **60s**) on the calling IO thread, and fail if it has not been executed.

```
threadpool:
    index:
        type: blocking
```

```
        min: 1
        size: 30
        wait_time: 30s
```

Thrift

The thrift transport module allows to expose the REST interface of elasticsearch using thrift. Thrift should provide better performance over http. Since thrift provides both the wire protocol and the transport, it should make using it simpler (thought its lacking on docs...).

Using thrift requires installing the **transport-thrift** plugin, located here.

The thrift schema can be used to generate thrift clients.

Setting	Description
thrift.port	The port to bind to. Defaults to 9500-9600
thrift.frame	Defaults to **-1**, which means no framing. Set to a higher value to specify the frame size (like **15mb**).

Transport

The transport module is used for internal communication between nodes within the cluster. Each call that goes from one node to the other uses the transport module (for example, when an HTTP GET request is processed by one node, and should actually be processed by another node that holds the data).

The transport mechanism is completely asynchronous in nature, meaning that there is no blocking thread waiting for a response. The benefit of using asynchronous communication is first solving the C10k problem, as well as being the idle solution for scatter (broadcast) / gather operations such as search in ElasticSearch.

TCP Transport

The TCP transport is an implementation of the transport module using TCP. It allows for the following settings:

Setting	Description
transport.tcp.port	A bind port range. Defaults to **9300-9400**.
transport.tcp.connect_timeout	The socket connect timeout setting (in time setting format). Defaults to **2s**.
transport.tcp.compress	Set to **true** to enable compression (LZF) between all nodes. Defaults to **false**.

It also shares the uses the common *network settings*.

Local Transport

This is a handy transport to use when running integration tests within the JVM. It is automatically enabled when using **NodeBuilder#local(true)**.

Index Modules

Index Modules are modules created per index and control all aspects related to an index. Since those modules lifecycle are tied to an index, all the relevant modules settings can be provided when creating an index (and it is actually the recommended way to configure an index).

Index Settings

There are specific index level settings that are not associated with any specific module. These include:

Setting	Description
index.compound_format	Should the compound file format be used (boolean setting). If not set, controlled by the actually store used, this is because the compound format was created to reduce the number of open file handles when using file based storage. By default, it is set to **false** for better performance (really applicable for file system based index storage), but, requires adapting the max open file handles.
index.term_index_interval	Set the interval between indexed terms. Large values cause less memory to be used by a reader / searcher, but slow random-access to terms. Small values cause more memory to be used by a reader / searcher, and speed random-access to terms. Defaults to **128**.
index.term_index_divisor	Subsamples which indexed terms are loaded into RAM. This has the same effect as **index.term_index_interval** except that setting must be done at indexing time while this setting can be set per reader / searcher. When set to N, then one in every N*termIndexInterval terms in the index is loaded into memory. By setting this to a value > 1 you can reduce memory usage, at the expense of higher latency when loading a TermInfo. The default value is 1. Set this to -1 to skip loading the terms index entirely.
index.refresh_interval	A time setting controlling how often the refresh operation will be executed. Defaults to **1s**. Can be set to **-1** in order to disable it.

Allocation

Shard Allocation Filtering

Allow to control allocation if indices on nodes based on include/exclude filters. The filters can be set both on the index level and on the cluster level. Lets start with an example of setting it on the cluster level:

Lets say we have 4 nodes, each has specific attribute called **tag** associated with it (the name of the attribute can be any name). Each node has a specific value associated with **tag**. Node 1 has a setting **node.tag: value1**, Node 2 a setting of **node.tag: value2**, and so on.

We can create an index that will only deploy on nodes that have **tag** set to **value1** and **value2** by setting **index.routing.allocation.include.tag** to **value1,value2**. For example:

```
curl -XPUT localhost:9200/test/_settings -d '{
    "index.routing.allocation.include.tag" : "value1,value2"
}'
```

On the other hand, we can create an index that will be deployed on all nodes except for nodes with a **tag** of value **value3** by setting **index.routing.allocation.exclude.tag** to **value3**. For example:

```
curl -XPUT localhost:9200/test/_settings -d '{
    "index.routing.allocation.exclude.tag" : "value3"
}'
```

The **include** and **exclude** values can have generic simple matching wildcards, for example, **value1***. A special attribute name called **_ip** can be used to match on node ip values.

Obviously a node can have several attributes associated with it, and both the attribute name and value are controlled in the setting. For example, here is a sample of several node configurations:

```
node.group1: group1_value1
node.group2: group2_value4
```

In the same manner, **include** and **exclude** can work against several attributes, for example:

```
curl -XPUT localhost:9200/test/_settings -d '{
    "index.routing.allocation.include.group1" : "xxx"
    "index.routing.allocation.include.group2" : "yyy",
    "index.routing.allocation.exclude.group3" : "zzz",
}'
```

The provided settings can also be updated in real time using the update settings API, allowing to "move" indices (shards) around in realtime.

Cluster wide filtering can also be defined, and be updated in real time using the cluster update settings API. This setting can come in handy for things like decommissioning nodes (even if the replica count is set to 0). Here is a sample of how to decommission a node based on **_ip** address:

```
curl -XPUT localhost:9200/_cluster/settings -d '{
    "transient" : {
        "cluster.routing.allocation.exclude._ip" : "10.0.0.1"
    }
}'
```

Total Shards Per Node

The **index.routing.allocation.total_shards_per_node** setting allows to control how many total shards for an index will be allocated per node. It can be dynamically set on a live index using the update index settings API.

Analysis

The index analysis module acts as a configurable registry of Analyzers that can be used in order to both break indexed (analyzed) fields when a document is indexed and process query strings. It maps to the Lucene **Analyzer**.

Analyzers are (generally) composed of a single **Tokenizer** and zero or more **TokenFilters**. A set of **CharFilters** can be associated with an analyzer to process the characters prior to other analysis steps. The analysis module allows one to register **TokenFilters**, **Tokenizers** and **Analyzers** under logical names that can then be referenced either in mapping definitions or in certain APIs. The Analysis module automatically registers (*if not explicitly defined*) built in analyzers, token filters, and tokenizers.

Here is a sample configuration:

```
index :
    analysis :
        analyzer :
            standard :
                type : standard
                stopwords : [stop1, stop2]
            myAnalyzer1 :
                type : standard
                stopwords : [stop1, stop2, stop3]
                max_token_length : 500
            # configure a custom analyzer which is
            # exactly like the default standard analyzer
            myAnalyzer2 :
                tokenizer : standard
                filter : [standard, lowercase, stop]
        tokenizer :
            myTokenizer1 :
                type : standard
```

```
            max_token_length : 900
        myTokenizer2 :
            type : keyword
            buffer_size : 512
    filter :
        myTokenFilter1 :
            type : stop
            stopwords : [stop1, stop2, stop3, stop4]
        myTokenFilter2 :
            type : length
            min : 0
            max : 2000
```

All analyzers, tokenizers, and token filters can be configured with a **version** parameter to control which Lucene version behavior they should use. Possible values are: **3.0**, **3.1**, **3.2**, **3.3**, **3.4** and **3.5** (the highest version number is the default option).

Types

Analyzer

Analyzers in general are broken down into a **Tokenizer** with zero or more **TokenFilter** applied to it. The analysis module allows one to register **TokenFilters**, **Tokenizers** and **Analyzers** under logical names which can then be referenced either in mapping definitions or in certain APIs. Here is a list of analyzer types:

Char Filter

Char filters allow one to filter out the stream of text before it gets tokenized (used within an **Analyzer**).

Tokenizer

Tokenizers act as the first stage of the analysis process (used within an **Analyzer**).

Token Filter

Token filters act as additional stages of the analysis process (used within an **Analyzer**).

Default Analyzers

An analyzer is registered under a logical name. It can then be referenced from mapping definitions or certain APIs. When none are defined, defaults are used. There is an option to define which analyzers will be used by default when none can be derived.

The **default** logical name allows one to configure an analyzer that will be used both for indexing and for searching APIs. The **default_index** logical name can be used to configure a default analyzer that will be used just when indexing, and the **default_search** can be used to configure a default analyzer that will be used just when searching.

Aliasing Analyzers

Analyzers can be aliased to have several registered lookup names associated with them. For example:

```
index :
  analysis :
    analyzer :
      standard :
        alias: [alias1, alias2]
        type : standard
        stopwords : [test1, test2, test3]
```

Will allow the **standard** analyzer to also be referenced with **alias1** and **alias2** values.

Custom Analyzer

An analyzer of type **custom** that allows to combine a **Tokenizer** with zero or more **Token Filters**, and zero or more **Char Filters**. The custom analyzer accepts a logical/registered name of the tokenizer to use, and a list of logical/registered names of token filters.

The following are settings that can be set for a **custom** analyzer type:

Setting	Description
tokenizer	The logical / registered name of the tokenizer to use.
filter	An optional list of logical / registered name of token filters.
char_filter	An optional list of logical / registered name of char filters.

Here is an example:

```
index :
    analysis :
        analyzer :
            myAnalyzer2 :
                type : custom
                tokenizer : myTokenizer1
                filter : [myTokenFilter1, myTokenFilter2]
                char_filter : [my_html]
        tokenizer :
            myTokenizer1 :
                type : standard
                max_token_length : 900
        filter :
            myTokenFilter1 :
                type : stop
                stopwords : [stop1, stop2, stop3, stop4]
            myTokenFilter2 :
                type : length
                min : 0
                max : 2000
        char_filter :
              my_html :
                type : html_strip
                escaped_tags : [xxx, yyy]
                read_ahead : 1024
```

Keyword Analyzer

An analyzer of type **keyword** that "tokenizes" an entire stream as a single token. This is useful for data like zip codes, ids and so on. Note, when using mapping definitions, it make more sense to simply mark the field as **not_analyzed**.

Lang Analyzer

A set of analyzers aimed at analyzing specific language text. The following types are supported: **arabic**, **armenian**, **basque**, **brazilian**, **bulgarian**, **catalan**, **chinese**, **cjk**, **czech**, **danish**, **dutch**, **english**, **finnish**, **french**, **galician**, **german**, **greek**, **hindi**, **hungarian**, **indonesian**, **italian**, **norwegian**, **persian**, **portuguese**, **romanian**, **russian**, **spanish**, **swedish**, **turkish**, **thai**.

All analyzers support setting custom **stopwords** either internally in the config, or by using an external stopwords file by setting **stopwords_path**.

The following analyzers support setting custom **stem_exclusion** list: **arabic**, **armenian**, **basque**, **brazilian**, **bulgarian**, **catalan**, **czech**, **danish**, **dutch**, **english**, **finnish**, **french**, **galician**, **german**, **hindi**, **hungarian**, **indonesian**, **italian**, **norwegian**, **portuguese**, **romanian**, **russian**, **spanish**, **swedish**, **turkish**.

Pattern Analyzer

An analyzer of type **pattern** that can flexibly separate text into terms via a regular expression. Accepts the following settings:

The following are settings that can be set for a **pattern** analyzer type:

Setting	Description
lowercase	Should terms be lowercased or not. Defaults to **true**.
pattern	The regular expression pattern, defaults to **W+**.
flags	The regular expression flags.

IMPORTANT: The regular expression should match the *token separators*, not the tokens themselves.

Flags should be pipe-separated, eg **'CASE_INSENSITIVE|COMMENTS"**. Check "Java Pattern API <http://download.oracle.com/javase/6/docs/api/java/util/regex/Pattern.html#field_summary>'_ for more details about **flags** options.

Pattern Analyzer Examples

In order to try out these examples, you should delete the **test** index before running each example:

```
curl -XDELETE localhost:9200/test
```

Whitespace tokenizer

```
curl -XPUT 'localhost:9200/test' -d '
{
    "settings":{
        "analysis": {
            "analyzer": {
                "whitespace":{
                    "type": "pattern",
```

```
                    "pattern":"\\s+"
                }
            }
        }
    }
}'

curl 'localhost:9200/test/_analyze?pretty=1&analyzer=whitespace' -d 'foo,bar baz'
# "foo,bar", "baz"
```

Non-word character tokenizer

```
curl -XPUT 'localhost:9200/test' -d '
{
    "settings":{
        "analysis": {
            "analyzer": {
                "nonword":{
                    "type": "pattern",
                    "pattern":"[^\\w]+"
                }
            }
        }
    }
}'

curl 'localhost:9200/test/_analyze?pretty=1&analyzer=nonword' -d 'foo,bar baz'
# "foo,bar baz" becomes "foo", "bar", "baz"

curl 'localhost:9200/test/_analyze?pretty=1&analyzer=nonword' -d 'type_1-type_4'
# "type_1","type_4"
```

CamelCase tokenizer

```
curl -XPUT 'localhost:9200/test?pretty=1' -d '
{
    "settings":{
        "analysis": {
            "analyzer": {
                "camel":{
                    "type": "pattern",
                    "pattern":"([^\\p{L}\\d]+)|(?<=\\D)(?=\\d)|(?<=\\d)(?=\\D)|(?
↪<=[\\p{L}&&[^\\p{Lu}]])(?=\\p{Lu})|(?<=\\p{Lu})(?=\\p{Lu}[\\p{L}&&[^\\p{Lu}]])"
                }
            }
        }
    }
}'

curl 'localhost:9200/test/_analyze?pretty=1&analyzer=camel' -d '
    MooseX::FTPClass2_beta
'
# "moose","x","ftp","class","2","beta"
```

The regex above is easier to understand as:

```
  ([^\\p{L}\\d]+)                 # swallow non letters and numbers,
| (?<=\\D)(?=\\d)                 # or non-number followed by number,
| (?<=\\d)(?=\\D)                 # or number followed by non-number,
| (?<=[ \\p{L} && [^\\p{Lu}]])    # or lower case
  (?=\\p{Lu})                     #   followed by upper case,
| (?<=\\p{Lu})                   # or upper case
  (?=\\p{Lu}                      #   followed by upper case
    [\\p{L}&&[^\\p{Lu}]]          #   then lower case
  )
```

Simple Analyzer

An analyzer of type **simple** that is built using a *Lower Case Tokenizer*.

Snowball Analyzer

An analyzer of type **snowball** that uses the *standard tokenizer*, with *standard filter*, *lowercase filter*, *stop filter*, and *snowball filter*.

The Snowball Analyzer is a stemming analyzer from Lucene that is originally based on the snowball project from snowball.tartarus.org.

Sample usage:

```
{
    "index" : {
        "analysis" : {
            "analyzer" : {
                "my_analyzer" : {
                    "type" : "snowball",
                    "language" : "English"
                }
            }
        }
    }
}
```

The **language** parameter can have the same values as the *snowball filter* and defaults to **English**. Note that not all the language analyzers have a default set of stopwords provided.

The **stopwords** parameter can be used to provide stopwords for the languages that has no defaults, or to simply replace the default set with your custom list. A default set of stopwords for many of these languages is available from for instance here and here.

A sample configuration (in YAML format) specifying Swedish with stopwords:

```
index :
    analysis :
        analyzer :
           my_analyzer:
                type: snowball
                language: Swedish
                stopwords: "och,det,att,i,en,jag,hon,som,han,på,den,med,var,sig,för,
↪så,till,är,men,ett,om,hade,de,av,icke,mig,du,henne,då,sin,nu,har,inte,hans,honom,
↪skulle,hennes,där,min,man,ej,vid,kunde,något,från,ut,när,efter,upp,vi,dem,vara,vad,
↪över,än,dig,kan,sina,här,ha,mot,alla,under,någon,allt,mycket,sedan,ju,denna,själv,
↪detta,åt,utan,varit,hur,ingen,mitt,ni,bli,blev,oss,din,dessa,några,deras,blir,mina,
↪samma,vilken,er,sådan,vår,blivit,dess,inom,mellan,sådant,varför,varje,vilka,ditt,
↪vem,vilket,sitta,sådana,vart,dina,vars,vårt,våra,ert,era,vilkas"
```

Standard Analyzer

An analyzer of type **standard** that is built of using *Standard Tokenizer*, with *Standard Token Filter*, *Lower Case Token Filter*, and *Stop Token Filter*.

The following are settings that can be set for a **standard** analyzer type:

Setting	Description
stopwords	A list of stopword to initialize the stop filter with. Defaults to the english stop words.
max_token_length	The maximum token length. If a token is seen that exceeds this length then it is discarded. Defaults to **255**.

Stop Analyzer

An analyzer of type **stop** that is built using a *Lower Case Tokenizer*, with *Stop Token Filter*.

The following are settings that can be set for a **stop** analyzer type:

Setting	Description
stopwords	A list of stopword to initialize the stop filter with. Defaults to the english stop words.
stopwords_path	A path (either relative to **config** location, or absolute) to a stopwords file configuration.

Whitespace Analyzer

An analyzer of type **whitespace** that is built using a *Whitespace Tokenizer*.

Asciifolding Tokenfilter

A token filter of type **asciifolding** that converts alphabetic, numeric, and symbolic Unicode characters which are not in the first 127 ASCII characters (the "Basic Latin" Unicode block) into their ASCII equivalents, if one exists.

Compound Word Tokenfilter

Token filters that allow to decompose compound words. There are two types available: **dictionary_decompounder** and **hyphenation_decompounder**.

The following are settings that can be set for a compound word token filter type:

Setting	Description
word_list	A list of words to use.
word_list_path	A path (either relative to **config** location, or absolute) to a list of words.

Here is an example:

```
index :
    analysis :
        analyzer :·
            myAnalyzer2 :
                type : custom
                tokenizer : standard
                filter : [myTokenFilter1, myTokenFilter2]
```

```
        filter :
            myTokenFilter1 :
                type : dictionary_decompounder
                word_list: [one, two, three]
            myTokenFilter2 :
                type : hyphenation_decompounder
                word_list_path: path/to/words.txt
```

Edgengram Tokenfilter

A token filter of type **edgeNGram**.

The following are settings that can be set for a **edgeNGram** token filter type:

Setting	Description
min_gram	Defaults to **1**.
max_gram	Defaults to **2**.
side	Either **front** or **back**.

Elision Tokenfilter

A token filter which removes elisions. For example, "l'avion" (the plane) will tokenized as "avion" (plane).

Accepts **articles** setting which is a set of stop words articles. For example:

```
"index" : {
    "analysis" : {
        "analyzer" : {
            "default" : {
                "tokenizer" : "standard",
                "filter" : ["standard", "elision"]
            }
        },
        "filter" : {
            "elision" : {
                "type" : "elision",
                "articles" : ["l", "m", "t", "qu", "n", "s", "j"]
            }
        }
    }
}
```

Kstem Tokenfilter

The **kstem** token filter is a high performance filter for english. All terms must already be lowercased (use **lowercase** filter) for this filter to work correctly.

Length Tokenfilter

A token filter of type **length** that removes words that are too long or too short for the stream.

The following are settings that can be set for a **length** token filter type:

Setting	Description
min	The minimum number. Defaults to **0**.
max	The maximum number. Defaults to **Integer.MAX_VALUE**.

Lowercase Tokenfilter

A token filter of type **lowercase** that normalizes token text to lower case.

Lowercase token filter supports Greek and Turkish lowercase token filters through the **language** parameter. Below is a usage example in a custom analyzer

```
index :
    analysis :
        analyzer :
            myAnalyzer2 :
                type : custom
                tokenizer : myTokenizer1
                filter : [myTokenFilter1, myGreekLowerCaseFilter]
                char_filter : [my_html]
        tokenizer :
            myTokenizer1 :
                type : standard
                max_token_length : 900
        filter :
            myTokenFilter1 :
                type : stop
                stopwords : [stop1, stop2, stop3, stop4]
            myGreekLowerCaseFilter :
                type : lowercase
                language : greek
        char_filter :
              my_html :
                type : html_strip
                escaped_tags : [xxx, yyy]
                read_ahead : 1024
```

Ngram Tokenfilter

A token filter of type **nGram**.

The following are settings that can be set for a **nGram** token filter type:

Setting	Description
min_gram	Defaults to **1**.
max_gram	Defaults to **2**.

Pattern_Replace Tokenfilter

The **pattern_replace** token filter allows to easily handle string replacements based on a regular expression. The regular expression is defined using the **pattern** parameter, and the replacement string can be provided using the **replacement** parameter (supporting referencing the original text, as explained here, java.lang.String)).

Phonetic Tokenfilter

The **phonetic** token filter is provided as a plugin and located here.

Porterstem Tokenfilter

A token filter of type **porterStem** that transforms the token stream as per the Porter stemming algorithm.

Note, the input to the stemming filter must already be in lower case, so you will need to use *Lower Case Token Filter* or *Lower Case Tokenizer* farther down the Tokenizer chain in order for this to work properly!. For example, when using custom analyzer, make sure the **lowercase** filter comes before the **porterStem** filter in the list of filters.

Reverse Tokenfilter

A token filter of type **reverse** that simply reverses the tokens.

Shingle Tokenfilter

A token filter of type **shingle** that constructs shingles (token n-grams) from a token stream. In other words, it creates combinations of tokens as a single token. For example, the sentence "please divide this sentence into shingles" might be tokenized into shingles "please divide", "divide this", "this sentence", "sentence into", and "into shingles".

This filter handles position increments > 1 by inserting filler tokens (tokens with termtext "_"). It does not handle a position increment of 0.

The following are settings that can be set for a **shingle** token filter type:

Setting	Description
max_shingle_size	Defaults to **2**.
output_unigrams	Defaults to **true**.

Snowball Tokenfilter

A filter that stems words using a Snowball-generated stemmer. The **language** parameter controls the stemmer with the following available values: **Armenian**, **Basque**, **Catalan**, **Danish**, **Dutch**, **English**, **Finnish**, **French**, **German**, **German2**, **Hungarian**, **Italian**, **Kp**, **Lovins**, **Norwegian**, **Porter**, **Portuguese**, **Romanian**, **Russian**, **Spanish**, **Swedish**, **Turkish**.

For example:

```
{
    "index" : {
        "analysis" : {
            "analyzer" : {
                "my_analyzer" : {
                    "tokenizer" : "standard",
                    "filter" : ["standard", "lowercase", "my_snow"]
                }
            },
            "filter" : {
                "my_snow" : {
                    "type" : "snowball",
                    "language" : "Lovins"
```

```
                }
            }
        }
    }
}
```

Standard Tokenfilter

A token filter of type **standard** that normalizes tokens extracted with the *Standard Tokenizer*.

Standard Tokenizer

A tokenizer of type **standard** providing grammar based tokenizer that is a good tokenizer for most European language documents. The tokenizer implements the Unicode Text Segmentation algorithm, as specified in Unicode Standard Annex #29.

The following are settings that can be set for a **standard** tokenizer type:

Setting	Description
max_token_length	The maximum token length. If a token is seen that exceeds this length then it is discarded. Defaults to **255**.

Stemmer Tokenfilter

A filter that stems words (similar to **snowball**, but with more options). The **language/name** parameter controls the stemmer with the following available values:

armenian, basque, bulgarian, catalan, danish, dutch, english, finnish, french, german, german2, greek, hungarian, italian, kp, lovins, norwegian, porter, porter2, portuguese, romanian, russian, spanish, swedish, turkish, minimal_english, possessive_english, light_finish, light_french, minimal_french, light_german, minimal_german, hindi, light_hungarian, indonesian, light_italian, light_portuguese, minimal_portuguese, portuguese, light_russian, light_spanish, light_swedish.

For example:

```
{
    "index" : {
        "analysis" : {
            "analyzer" : {
                "my_analyzer" : {
                    "tokenizer" : "standard",
                    "filter" : ["standard", "lowercase", "my_stemmer"]
                }
            },
            "filter" : {
                "my_stemmer" : {
                    "type" : "stemmer",
                    "name" : "light_german"
                }
            }
        }
    }
}
```

Stop Tokenfilter

A token filter of type **stop** that removes stop words from token streams.

The following are settings that can be set for a **stop** token filter type:

Setting	Description
stopwords	A list of stop words to use. Defaults to english stop words.
stopwords_path	A path (either relative to **config** location, or absolute) to a stopwords file configuration.
enable_position_increments	Set to **true** if token positions should record the removed stop words, **false** otherwise. Defaults to **true**.
ignore_case	Set to **true** to lower case all words first. Defaults to **false**.

stopwords allow for custom language specific expansion of default stopwords. It follows the **_lang_** notation and supports: arabic, armenian, basque, brazilian, bulgarian, catalan, czech, danish, dutch, english, finnish, french, galician, german, greek, hindi, hungarian, indonesian, italian, norwegian, persian, portuguese, romanian, russian, spanish, swedish, turkish.

Synonym Tokenfilter

The **synonym** token filter allows to easily handle synonyms during the analysis process. Synonyms are configured using a configuration file. Here is an example:

```
{
    "index" : {
        "analysis" : {
            "analyzer" : {
                "synonym" : {
                    "tokenizer" : "whitespace",
                    "filter" : ["synonym"]
                }
            },
            "filter" : {
                "synonym" : {
                    "type" : "synonym",
                    "synonyms_path" : "analysis/synonym.txt"
                }
            }
        }
    }
}
```

The above configures a **synonym** filter, with a path of **analysis/synonym.txt** (relative to the **config** location). The **synonym** analyzer is then configured with the filter. Additional settings are: **ignore_case** (defaults to **false**), and **expand** (defaults to **true**).

The **tokenizer** parameter controls the tokenizers that will be used to tokenize the synonym, and defaults to the **whitespace** tokenizer.

As of elasticsearch 0.17.9 two synonym formats are supported: Solr, WordNet.

Solr synonyms

The following is a sample format of the file:

```
# blank lines and lines starting with pound are comments.

#Explicit mappings match any token sequence on the LHS of "=>"
#and replace with all alternatives on the RHS.  These types of mappings
#ignore the expand parameter in the schema.
#Examples:
i-pod, i pod => ipod,
sea biscuit, sea biscit => seabiscuit

#Equivalent synonyms may be separated with commas and give
#no explicit mapping.  In this case the mapping behavior will
#be taken from the expand parameter in the schema.  This allows
#the same synonym file to be used in different synonym handling strategies.
#Examples:
ipod, i-pod, i pod
foozball , foosball
universe , cosmos

# If expand==true, "ipod, i-pod, i pod" is equivalent to the explicit mapping:
ipod, i-pod, i pod => ipod, i-pod, i pod
# If expand==false, "ipod, i-pod, i pod" is equivalent to the explicit mapping:
ipod, i-pod, i pod => ipod

#multiple synonym mapping entries are merged.
foo => foo bar
foo => baz
#is equivalent to
foo => foo bar, baz
```

You can also define synonyms for the filter directly in the configuration file (note use of **synonyms** instead of **synonyms_path**):

```
{
    "filter" : {
        "synonym" : {
            "type" : "synonym",
            "synonyms" : [
                "i-pod, i pod => ipod",
                "universe, cosmos"
            ]
        }
    }
}
```

However, it is recommended to define large synonyms set in a file using **synonyms_path**.

WordNet synonyms

Synonyms based on WordNet format can be declared using **format**:

```
{
    "filter" : {
        "synonym" : {
            "type" : "synonym",
            "format" : "wordnet",
            "synonyms" : [
```

```
                "s(100000001,1,'abstain',v,1,0).",
                "s(100000001,2,'refrain',v,1,0).",
                "s(100000001,3,'desist',v,1,0)."
            ]
        }
    }
}
```

Using **synonyms_path** to define WordNet synonyms in a file is supported as well.

Trim Tokenfilter

The **trim** token filter trims surrounding whitespaces around a token.

Word Delimiter Tokenfilter

Named **word_delimiter**, it Splits words into subwords and performs optional transformations on subword groups. Words are split into subwords with the following rules:

- split on intra-word delimiters (by default, all non alpha-numeric characters).
- "Wi-Fi" -> "Wi", "Fi"
- split on case transitions: "PowerShot" -> "Power", "Shot"
- split on letter-number transitions: "SD500" -> "SD", "500"
- leading and trailing intra-word delimiters on each subword are ignored: "//hello—there, 'dude"' -> "hello", "there", "dude"
- trailing "'s" are removed for each subword: "O'Neil's" -> "O", "Neil"

Parameters include:

- **generate_word_parts**: If **true** causes parts of words to be generated: "PowerShot" => "Power" "Shot". Defaults to **true**.
- **generate_number_parts**: If **true** causes number subwords to be generated: "500-42" => "500" "42". Defaults to **true**.
- **catenate_words**: If **true** causes maximum runs of word parts to be catenated: "wi-fi" => "wifi". Defaults to **false**.
- **catenate_numbers**: If **true** causes maximum runs of number parts to be catenated: "500-42" => "50042". Defaults to **false**.
- **catenate_all**: If **true** causes all subword parts to be catenated: "wi-fi-4000" => "wifi4000". Defaults to **false**.
- **split_on_case_change**: If **true** causes "PowerShot" to be two tokens; ("Power-Shot" remains two parts regards). Defaults to **true**.
- **preserve_original**: If **true** includes original words in subwords: "500-42" => "500" "42" "500-42". Defaults to **false**.
- **split_on_numerics**: If **true** causes "j2se" to be three tokens; "j" "2" "se". Defaults to **true**.
- **stem_english_possessive**: If **true** causes trailing "'s" to be removed for each subword: "O'Neil's" => "O", "Neil". Defaults to **true**.

Advance settings include:

protected_words: A list of protected words from being delimiter. Either an array, or also can set **protected_words_path** which resolved to a file configured with protected words (one on each line). Automatically resolves to **config/** based location if exists.

type_table: A custom type mapping table, for example (when configured using **type_table_path**):

```
# Map the $, %, '.', and ',' characters to DIGIT
# This might be useful for financial data.
$ => DIGIT
% => DIGIT
. => DIGIT
\u002C => DIGIT

# in some cases you might not want to split on ZWJ
# this also tests the case where we need a bigger byte[]
# see http://en.wikipedia.org/wiki/Zero-width_joiner
\u200D => ALPHANUM
```

Edgengram Tokenizer

A tokenizer of type **edgeNGram**.

The following are settings that can be set for a **edgeNGram** tokenizer type:

Setting	Description
min_gram	Defaults to **1**.
max_gram	Defaults to **2**.
side	Either **front** or **back**.

Keyword Tokenizer

A tokenizer of type **keyword** that emits the entire input as a single input.

The following are settings that can be set for a **keyword** tokenizer type:

Setting	Description
buffer_size	The term buffer size. Defaults to **256**.

Letter Tokenizer

A tokenizer of type **letter** that divides text at non-letters. That's to say, it defines tokens as maximal strings of adjacent letters. Note, this does a decent job for most European languages, but does a terrible job for some Asian languages, where words are not separated by spaces.

Lowercase Tokenizer

A tokenizer of type **lowercase** that performs the function of *Letter Tokenizer* and *Lower Case Token Filter* together. It divides text at non-letters and converts them to lower case. While it is functionally equivalent to the combination of *Letter Tokenizer* and *Lower Case Token Filter*, there is a performance advantage to doing the two tasks at once, hence this (redundant) implementation.

Ngram Tokenizer

A tokenizer of type **nGram**.

The following are settings that can be set for a **nGram** tokenizer type:

Setting	Description
min_gram	Defaults to **1**.
max_gram	Defaults to **2**.

Pathhierarchy Tokenizer

The **path_hierarchy** tokenizer takes something like this:

<pre> /something/something/else

And produces tokens:

<pre> /something /something/something /something/something/else

Setting	Description
delimiter	The character delimiter to use, defaults to **/**.
replacement	An optional replacement character to use. Defaults to the **delimiter**.
buffer_size	The buffer size to use, defaults to **1024**.
reverse	Generates tokens in reverse order, defaults to **false**.
skip	Controls initial tokens to skip, defaults to **0**.

Pattern Tokenizer

A tokenizer of type **pattern** that can flexibly separate text into terms via a regular expression. Accepts the following settings:

Setting	Description
pattern	The regular expression pattern, defaults to **W+**.
flags	The regular expression flags.
group	Which group to extract into tokens. Defaults to **-1** (split).

IMPORTANT: The regular expression should match the *token separators*, not the tokens themselves.

group set to **-1** (the default) is equivalent to "split". Using Using group >= 0 selects the matching group as the token. For example, if you have:

<pre> pattern = '([^']+)' group = 0 input = aaa 'bbb' 'ccc'

the output will be two tokens: 'bbb' and 'ccc' (including the ' marks). With the same input but using group=1, the output would be: bbb and ccc (no ' marks).

Uaxurlemail Tokenizer

A tokenizer of type **uax_url_email** which works exactly like the **standard** tokenizer, but tokenizes emails and urls as single tokens.

The following are settings that can be set for a **uax_url_email** tokenizer type:

Setting	Description
max_token_length	The maximum token length. If a token is seen that exceeds this length then it is discarded. Defaults to **255**.

Truncate Tokenfilter

The **truncate** token filter can be used to truncate tokens into a specific length. This can come in handy with keyword (single token) based mapped fields that are used for sorting in order to reduce memory usage.

It accepts a **length** parameter which control the number of characters to truncate to, defaults to **10**.

Whitespace Tokenizer

A tokenizer of type **whitespace** that divides text at whitespace.

Unique Tokenfilter

The **unique** token filter can be used to only index unique tokens during analysis. By default it is applied on all the token stream. If **only_on_same_position** is set to **true**, it will only remove duplicate tokens on the same position.

Htmlstrip Charfilter

A char filter of type **html_strip** stripping out HTML elements from an analyzed text.

Mapping Charfilter

A char filter of type **mapping** replacing characters of an analyzed text with given mapping.

Here is a sample configuration:

```
{
    "index" : {
        "analysis" : {
            "char_filter" : {
                "my_mapping" : {
                    "type" : "mapping",
                    "mappings" : ["ph=>f", "qu=>q"]
                }
            },
            "analyzer" : {
                "custom_with_char_filter" : {
                    "tokenizer" : "standard",
                    "char_filter" : ["my_mapping"]
                },
            }
        }
    }
}
```

Otherwise the setting **mappings_path** can specify a file where you can put the list of char mapping :

```
ph => f
qu => k
```

Icu Plugin

The ICU analysis plugin allows for unicode normalization, collation and folding. The plugin is called elasticsearch-analysis-icu.

The plugin includes the following analysis components:

ICU Normalization

Normalizes characters as explained here. It registers itself by default under **icu_normalizer** or **icuNormalizer** using the default settings. Allows for the name parameter to be provided which can include the following values: **nfc**, **nfkc**, and **nfkc_cf**. Here is a sample settings:

```
{
    "index" : {
        "analysis" : {
            "analyzer" : {
                "normalization" : {
                    "tokenizer" : "keyword",
                    "filter" : ["icu_normalizer"]
                }
            }
        }
    }
}
```

ICU Folding

Folding of unicode characters based on **UTR#30**. It registers itself under **icu_folding** and **icuFolding** names.

The filter also does lowercasing, which means the lowercase filter can normally be left out. Sample setting:

```
{
    "index" : {
        "analysis" : {
            "analyzer" : {
                "folding" : {
                    "tokenizer" : "keyword",
                    "filter" : ["icu_folding"]
                }
            }
        }
    }
}
```

Filtering

The folding can be filtered by a set of unicode characters with the parameter **unicodeSetFilter**. This is useful for a non-internationalized search engine where retaining a set of national characters which are primary letters in a specific language is wanted. See syntax for the UnicodeSet here2.

The Following example excempt Swedish characters from the folding. Note that the filtered characters are NOT lowercased which is why we add that filter below.

```
{
    "index" : {
        "analysis" : {
            "analyzer" : {
                "folding" : {
                    "tokenizer" : "standard",
                    "filter" : ["my_icu_folding", "lowercase"]
                }
            }
            "filter" : {
                "my_icu_folding" : {
                    "type" : "icu_folding"
                    "unicodeSetFilter" : "[^åäöÅÄÖ]"
                }
            }
        }
    }
}
```

ICU Collation

Uses collation token filter. Allows to either specify the rules for collation (defined here3 using the **rules** parameter (can point to a location or expressed in the settings, location can be relative to config location), or using the **language** parameter (further specialized by country and variant). By default registers under **icu_collation** or **icuCollation** and uses the default locale.

Here is a sample settings:

```
{
    "index" : {
        "analysis" : {
            "analyzer" : {
                "collation" : {
                    "tokenizer" : "keyword",
                    "filter" : ["icu_collation"]
                }
            }
        }
    }
}
```

And here is a sample of custom collation:

```
{
    "index" : {
        "analysis" : {
            "analyzer" : {
                "collation" : {
                    "tokenizer" : "keyword",
                    "filter" : ["myCollator"]
                }
            },
            "filter" : {
                "myCollator" : {
                    "type" : "icu_collation",
                    "language" : "en"
```

```
                    }
                }
            }
        }
}
```

Cache

There are different caching inner modules associated with an index. They include **filter**, **field** and others.

Filter Cache

The filter cache is responsible for caching the results of filters (used in the query). The default implementation of a filter cache (and the one recommended to use in almost all cases) is the **node** filter cache type.

Node Filter Cache

The **node** filter cache may be configured to use either a percentage of the total memory allocated to the process or an specific amount of memory. All shards present on a node share a single node cache (thats why its called **node**). The cache implements an LRU eviction policy: when a cache becomes full, the least recently used data is evicted to make way for new data.

The setting that allows one to control the memory size for the filter cache is **indices.cache.filter.size**, which defaults to **20%**. *Note*, this is *not* an index level setting but a node level setting (can be configured in the node configuration).

indices.cache.filter.size can accept either a percentage value, like **30%**, or an exact value, like **512mb**.

Index Filter Cache

A filter cache that exists on the index level (on each node). Generally, not recommended for use since its memory usage depends on which shards are allocated on each node and its hard to predict it. The types are: **resident**, **soft** and **weak**.

All types support the following settings:

Setting	Description
index.cache.filter.max_size	The max size (count, not byte size) of the cache (per search segment in a shard). Defaults to **-1**, which is usually fine with **soft** cache and proper cacheable filters.
index.cache.filter.expire	A time based setting that expires filters after a certain time of inactivity. Defaults to **-1**. For example, can be set to **5m** for a 5 minute expiry.

Field Data Cache

The field data cache is used mainly when sorting on or faceting on a field. It loads all the field values to memory in order to provide fast document based access to those values. The field data cache can be expensive to build for a field, so its recommended to have enough memory to allocate it, and keep it loaded.

The default type for the field data cache is **resident** (because of the cost of rebuilding it). Other types include **soft**.

Setting	Description
index.cache.field.max_size	The max size (count, not byte size) of the cache (per search segment in a shard). Defaults to unset (**-1**).
index.cache.field.expire	A time based setting that expires filters after a certain time of inactivity. Defaults to **-1**. For example, can be set to **5m** for a 5 minute expiry.

Mapper

The mapper module acts as a registry for the type mapping definitions added to an index either when creating it or by using the put mapping api. It also handles the dynamic mapping support for types that have no explicit mappings pre defined. For more information about mapping definitions, check out the *mapping section*.

Dynamic / Default Mappings

Dynamic mappings allow to automatically apply generic mapping definition to types that do not have mapping pre defined or applied to new mapping definitions (overridden). This is mainly done thanks to the fact that the **object** type and namely the root **object** type allow for schema less dynamic addition of unmapped fields.

The default mapping definition is plain mapping definition that is embedded within ElasticSearch:

```
{
    _default_ : {
    }
}
```

Pretty short, no? Basically, everything is defaulted, especially the dynamic nature of the root object mapping. The default mapping definition can be overridden in several manners. The simplest manner is to simply define a file called **default-mapping.json** and placed it under the **config** directory (which can be configured to exist in a different location). It can also be explicitly set using the **index.mapper.default_mapping_location** setting.

Dynamic creation of mappings for unmapped types can be completely disabled by setting **index.mapper.dynamic** to **false**.

Merge

A shard in elasticsearch is a Lucene index, and a lucene index is broken down into segments. Segments are internal storage elements in the index where the index data is stored, and are immutable up to delete markers. Segments are, periodically, merged into larger segments to keep the index size at bay and expunge deletes.

The more segments one has in the Lucene index mean slower searches and more memory used, but, low number of segments means more merging that has to go on.

Since merges can be expensive to perform, especially on low IO environments, they can be throtted using store level throttling. Read the store module documentation on how to set it.

Policy

The index merge policy module allows one to control which segments of a shard index are to be merged. There are several types of policies with the default set to **tiered**.

tiered

Merges segments of approximately equal size, subject to an allowed number of segments per tier. This is similar to **log_bytes_size** merge policy, except this merge policy is able to merge non-adjacent segment, and separates how many segments are merged at once from how many segments are allowed per tier. This merge policy also does not over-merge (ie, cascade merges).

This policy has the following settings:

Setting	Description
index.merge.policy.expunge_deletes_allowed	When expungeDeletes is called, we only merge away a segment if its delete percentage is over this threshold. Default is **10**.
index.merge.policy.floor_segment	Segments smaller than this are "rounded up" to this size, ie treated as equal (floor) size for merge selection. This is to prevent frequent flushing of tiny segments from allowing a long tail in the index. Default is **2mb**.
index.merge.policy.max_merge_at_once	Maximum number of segments to be merged at a time during "normal" merging. Default is **10**.
index.merge.policy.max_merge_at_once_explicit	Maximum number of segments to be merged at a time, during optimize or expunge_deletes. Default is **30**.
index.merge.policy.max_merged_segment	Maximum sized segment to produce during normal merging (not explicit optimize). This setting is approximate: the estimate of the merged segment size is made by summing sizes of to-be-merged segments (compensating for percent deleted docs). Default is **5gb**.
index.merge.policy.segments_per_tier	Sets the allowed number of segments per tier. Smaller values mean more merging but fewer segments. Default is **10**. Note, this value needs to be >= then the **max_merge_at_once_** otherwise you'll force too many merges to occur.
index.reclaim_deletes_weight	Controls how aggressively merges that reclaim more deletions are favored. Higher values favor selecting merges that reclaim deletions. A value of **0.0** means deletions don't impact merge selection. Defaults to **2.0**.
index.compound_format	Should the index be stored in compound format or not. Defaults to **false**.

For normal merging, this policy first computes a "budget" of how many segments are allowed by be in the index. If the index is over-budget, then the policy sorts segments by decreasing size (pro-rating by percent deletes), and then finds the least-cost merge. Merge cost is measured by a combination of the "skew" of the merge (size of largest seg divided by smallest seg), total merge size and pct deletes reclaimed, so that merges with lower skew, smaller size and those reclaiming more deletes, are favored.

If a merge will produce a segment that's larger than **max_merged_segment** then the policy will merge fewer segments (down to 1 at once, if that one has deletions) to keep the segment size under budget.

Note, this can mean that for large shards that holds many gigabytes of data, the default of **max_merged_segment** (**5gb**) can cause for many segments to be in an index, and causing searches to be slower. Use the indices segments API to see the segments that an index have, and possibly either increase the **max_merged_segment** or issue an optimize call for the index (try and aim to issue it on a low traffic time).

log_byte_size

A merge policy that merges segments into levels of exponentially increasing *byte size*, where each level has fewer segments than the value of the merge factor. Whenever extra segments (beyond the merge factor upper bound) are encountered, all segments within the level are merged.

This policy has the following settings:

Setting	Description
index.merge.policy.merge_factor	Determines how often segment indices are merged by index operation. With smaller values, less RAM is used while indexing, and searches on unoptimized indices are faster, but indexing speed is slower. With larger values, more RAM is used during indexing, and while searches on unoptimized indices are slower, indexing is faster. Thus larger values (greater than 10) are best for batch index creation, and smaller values (lower than 10) for indices that are interactively maintained. Defaults to **10**.
index.merge.policy.min_merge_size	A size setting type which sets the minimum size for the lowest level segments. Any segments below this size are considered to be on the same level (even if they vary drastically in size) and will be merged whenever there are mergeFactor of them. This effectively truncates the "long tail" of small segments that would otherwise be created into a single level. If you set this too large, it could greatly increase the merging cost during indexing (if you flush many small segments). Defaults to **1.6mb**
index.merge.policy.max_merge_size	A size setting type which sets the largest segment (measured by total byte size of the segment's files) that may be merged with other segments. Defaults to unbounded.
index.merge.policy.max_merge_docs	Determines the largest segment (measured by document count) that may be merged with other segments. Defaults to unbounded.

log_doc

A merge policy that tries to merge segments into levels of exponentially increasing *document count*, where each level has fewer segments than the value of the merge factor. Whenever extra segments (beyond the merge factor upper bound) are encountered, all segments within the level are merged.

Setting	Description
index.merge.policy.merge_factor	Determines how often segment indices are merged by index operation. With smaller values, less RAM is used while indexing, and searches on unoptimized indices are faster, but indexing speed is slower. With larger values, more RAM is used during indexing, and while searches on unoptimized indices are slower, indexing is faster. Thus larger values (greater than 10) are best for batch index creation, and smaller values (lower than 10) for indices that are interactively maintained. Defaults to **10**.
index.merge.policy.min_merge_docs	Sets the minimum size for the lowest level segments. Any segments below this size are considered to be on the same level (even if they vary drastically in size) and will be merged whenever there are mergeFactor of them. This effectively truncates the "long tail" of small segments that would otherwise be created into a single level. If you set this too large, it could greatly increase the merging cost during indexing (if you flush many small segments). Defaults to **1000**.
index.merge.policy.max_merge_docs	Determines the largest segment (measured by document count) that may be merged with other segments. Defaults to unbounded.

Scheduling

The merge schedule controls the execution of merge operations once they are needed (according to the merge policy). The following types are supported, with the default being the **ConcurrentMergeScheduler**.

ConcurrentMergeScheduler

A merge scheduler that runs merges using a separated thread, until the maximum number of threads at which when a merge is needed, the thread(s) that are updating the index will pause until one or more merges completes.

The scheduler supports the following settings:

Setting	Description
index.merge.scheduler.max_thread_count	The maximum number of threads to perform the merge operation. Defaults to **Math.max(1, Math.min(3, Runtime.getRuntime().availableProcessors() / 2))**.

SerialMergeScheduler

A merge scheduler that simply does each merge sequentially using the calling thread (blocking the operations that triggered the merge, the index operation).

Slowlog

Search Slow Log

Shard level slow search log allows to log slow search (query and fetch executions) into a dedicated log file.

Thresholds can be set for both the query phase of the execution, and fetch phase, here is a sample:

```
#index.search.slowlog.threshold.query.warn: 10s
#index.search.slowlog.threshold.query.info: 5s
#index.search.slowlog.threshold.query.debug: 2s
#index.search.slowlog.threshold.query.trace: 500ms

#index.search.slowlog.threshold.fetch.warn: 1s
#index.search.slowlog.threshold.fetch.info: 800ms
#index.search.slowlog.threshold.fetch.debug: 500ms
#index.search.slowlog.threshold.fetch.trace: 200ms
```

By default, none are enabled (set to **-1**). Levels (**warn**, **info**, **debug**, **trace**) allow to control under which logging level the log will be logged. Not all are required to be configured (for example, only **warn** threshold can be set). The benefit of several levels is the ability to quickly "grep" for specific thresholds breached.

The logging is done on the shard level scope, meaning the execution of a search request within a specific shard. It does not encompass the whole search request, which can be broadcast to several shards in order to execute. Some of the benefits of shard level logging is the association of the actual execution on the specific machine, compared with request level.

All settings are index level settings (and each index can have different values for it), and can be changed in runtime using the index update settings API.

The logging file is configured by default using the following configuration (found in **logging.yml**):

```
index_search_slow_log_file:
  type: dailyRollingFile
  file: ${path.logs}/${cluster.name}_index_search_slowlog.log
  datePattern: "'.'yyyy-MM-dd"
  layout:
    type: pattern
    conversionPattern: "[%d{ISO8601}][%-5p][%-25c] %m%n"
```

Store

The store module allow you to control how index data is stored. It's important to note that the storage is considered temporal only for the shared gateway (non **local**, which is the default).

The index can either be stored in-memory (no persistence) or on-disk (the default). In-memory indices provide better performance at the cost of limiting the index size to the amount of available physical memory.

When using a local gateway (the default), file system storage with *no* in memory storage is required to maintain index consistency. This is required since the local gateway constructs its state from the local index state of each node. When using a shared gateway (like NFS or S3), the index can be safely stored in memory (with replicas).

Another important aspect of memory based storage is the fact that ElasticSearch supports storing the index in memory *outside of the JVM heap space* using the "Memory" (see below) storage type. It translates to the fact that there is no need for extra large JVM heaps (with their own consequences) for storing the index in memory.

Store Level Compression

(0.19.5 and above).

In the mapping, one can configure the **_source** field to be compressed. The problem with it is the fact that small documents don't end up compressing well, as several documents compressed in a single compression "block" will provide a considerable better compression ratio. This version introduces the ability to compress stored fieds using the **index.store.compress.stored** setting, as well as term vector using the **index.store.compress.tv** setting.

The settings can be set on the index level, and are dynamic, allowing to change them using the index update settings API. elasticsearch can handle mixed stored / non stored cases. This allows, for example, to enable compression at a later stage in the index lifecycle, and optimize the index to make use of it (generating new segmetns that use compression).

Best compression, compared to _source level compression, will mainly happen when indexing smaller documents (less than 64k). The price on the other hand is the fact that for each doc returned, a block will need to be decompressed (its fast though) in order to extract the document data.

Store Level Throttling

(0.19.5 and above).

The way Lucene, the IR library elasticsearch uses under the covers, works is by creating immutable segments (up to deletes) and constantly merging them (the merge policy settings allow to control how those merges happen). The merge process happen in an asyncronous manner without affecting the indexing / search speed. The problem though, especially on systems with low IO, is that the merge process can be expensive and affect search / index operation simply by the fact that the box is now taxed with more IO happening.

The store module allows to have throttling configured for merges (or all) either on the node level, or on the index level. The node level throttling will make sure that out of all the shards allocated on that node, the merge process won't pass the specific setting bytes per second. It can be set by setting **indices.store.throttle.type** to **merge**, and setting **indices.store.throttle.max_bytes_per_sec** to something like **5mb**. The node level settings can be changed dynamically using the cluster update settings API.

If specific index level configuration is needed, regardless of the node level settings, it can be set as well using the **index.store.throttle.type**, and **index.store.throttle.max_bytes_per_sec**. The default value for the type is **node**, meaning it will throttle based on the node level settings and participate in the global throttling happening. Both settings can be set using the index udpate settings API dynamically.

The following sections lists all the different storage types supported.

File System

File system based storage is the default storage used. There are different implementations or storage types. The best one for the operating environment will be automatically chosen: **mmapfs** on Solaris/Windows 64bit, **simplefs** on

Windows 32bit, and **niofs** for the rest.

The following are the different file system based storage types:

Simple FS

The **simplefs** type is a straightforward implementation of file system storage (maps to Lucene **SimpleFsDirectory**) using a random access file. This implementation has poor concurrent performance (multiple threads will bottleneck). Its usually better to use the **niofs** when you need index persistence.

NIO FS

The **niofs** type stores the shard index on the file system (maps to Lucene **NIOFSDirectory**) using NIO. It allows multiple threads to read from the same file concurrently. It is not recommended on Windows because of a bug in the SUN Java implementation.

MMap FS

The **mmapfs** type stores the shard index on the file system (maps to Lucene **MMapDirectory**) by mapping a file into memory (mmap). Memory mapping uses up a portion of the virtual memory address space in your process equal to the size of the file being mapped. Before using this class, be sure your have plenty of virtual address space.

Memory

The **memory** type stores the index in main memory with the following configuration options:

There are also *node* level settings that control the caching of buffers (important when using direct buffers):

Setting	Description
cache.memory.direct	Should the memory be allocated outside of the JVM heap. Defaults to **true**.
cache.memory.small_buffer_size	The small buffer size, defaults to **1kb**.
cache.memory.large_buffer_size	The large buffer size, defaults to **1mb**.
cache.memory.small_cache_size	The small buffer cache size, defaults to **10mb**.
cache.memory.large_cache_size	The large buffer cache size, defaults to **500mb**.

Translog

Each shard has a transaction log or write ahead log associated with it. It allows to guarantee that when an index/delete operation occurs, it is applied atomically, while not "committing" the internal lucene index for each request. A flush ("commit") still happens based on several parameters:

Setting	Description
index.translog.flush_threshold_ops	After how many operations to flush. Defaults to **5000**.
index.translog.flush_threshold_size	Once the translog hits this size, a flush will happen. Defaults to **500mb**.
index.translog.flush_threshold_period	The period with no flush happening to force a flush. Defaults to **60m**.

River

A river is a pluggable service running within elasticsearch cluster pulling data (or being pushed with data) that is then indexed into the cluster.

A river is composed of a unique name and a type. The type is the type of the river (out of the box, there is the **dummy** river that simply logs that its running). The name uniquely identifies the river within the cluster. For example, one can run a river called **my_river** with type **dummy**, and another river called **my_other_river** with type **dummy**.

Couchdb

—

The CouchDB River allows to automatically index couchdb and make it searchable using the excellent _changes stream couchdb provides.

See README file for details.

Rabbitmq

—

RabbitMQ River allows to automatically index a RabbitMQ queue.

See README file for details.

Twitter

—

The twitter river indexes the public twitter stream, aka the hose, and makes it searchable.

See README file for details.

Wikipedia

—

A simple river to index Wikipedia.

See README file for details.

How it Works

A river instance (and its name) is a type within the **_river** index. All different rivers implementations accept a document called **_meta** that at the very least has the type of the river (twitter / couchdb / ...) associated with it. Creating a river is a simple curl request to index that **_meta** document (there is actually a **dummy** river used for testing):

```
curl -XPUT 'localhost:9200/_river/my_river/_meta' -d '{
    "type" : "dummy"
}'
```

A river can also have more data associated with it in the form of more documents indexed under the given index type (the river name). For example, storing the last indexed state can be stored in a document that holds it.

Deleting a river is a call to delete the type (and all documents associated with it):

```
curl -XDELETE 'localhost:9200/_river/my_river/'
```

Cluster Allocation

Rivers are singletons within the cluster. They get allocated automatically to one of the nodes and run. If that node fails, an river will be automatically allocated to another node.

River allocation on nodes can be controlled on each node. The **node.river** can be set to **_none_** disabling any river allocation to it. The **node.river** can also include a comma separated list of either river names or types controlling the rivers allowed to run on it. For example: **my_river1,my_river2**, or **dummy,twitter**.

Status

Each river (regardless of the implementation) exposes a high level **_status** doc which includes the node the river is running on. Getting the status is a simple curl GET request to **/_river/{river name}/_status**.

Java Api

This section describes the Java API that elasticsearch provides. All elasticsearch operations are executed using a *Client* object. All operations are completely asynchronous in nature (either accepts a listener, or return a future).

Additionally, operations on a client may be accumulated and executed in *Bulk*.

Note, all the *APIs* are exposed through the Java API (actually, the Java API is used internally to execute them).

Maven Repository

elasticsearch is hosted on Sonatype, with both a releases repo and a snapshots repo.

Bulk

The bulk API allows one to index and delete several documents in a single request. Here is a sample usage:

```
import static org.elasticsearch.common.xcontent.XContentFactory.*;

BulkRequestBuilder bulkRequest = client.prepareBulk();

// either use client#prepare, or use Requests# to directly build index/delete requests
bulkRequest.add(client.prepareIndex("twitter", "tweet", "1")
        .setSource(jsonBuilder()
                    .startObject()
                        .field("user", "kimchy")
                        .field("postDate", new Date())
                        .field("message", "trying out Elastic Search")
                    .endObject()
                  )
        );
```

```
bulkRequest.add(client.prepareIndex("twitter", "tweet", "2")
        .setSource(jsonBuilder()
                    .startObject()
                        .field("user", "kimchy")
                        .field("postDate", new Date())
                        .field("message", "another post")
                    .endObject()
                  )
        );

BulkResponse bulkResponse = bulkRequest.execute().actionGet();
if (bulkResponse.hasFailures()) {
    // process failures by iterating through each bulk response item
}
```

Client

Obtaining an elasticsearch **Client** is simple. The most common way to get a client is by 1) creating an embedded **Node** that acts as a node within a cluster and 2) requesting a **Client** from your embedded **Node**. Another manner is by creating a **TransportClient** that connects to a cluster.

Node Client

Instantiating a node based client is the simplest way to get a **Client** that can execute operations against elasticsearch.

```
import static org.elasticsearch.node.NodeBuilder.*;

// on startup

Node node = nodeBuilder().node();
Client client = node.client();

// on shutdown

node.close();
```

When you start a **Node**, it joins an elasticsearch cluster. You can have different clusters by simple setting the **cluster.name** setting, or explicitly using the **clusterName** method on the builder. The benefit of using the **Client** is the fact that operations are automatically routed to the node(s) the operations need to be executed on, without performing a "double hop". For example, the index operation will automatically be executed on the shard that it will end up existing at.

When you start a **Node**, the most important decision is whether it should hold data or not. In other words, should indices and shards be allocated to it. Many times we would like to have the clients just be clients, without shards being allocated to them. This is simple to configure by setting either **node.data** setting to **false** or **node.client** to **true** (the **NodeBuilder** respective helper methods on it):

```
import static org.elasticsearch.node.NodeBuilder.*;

// on startup

Node node = nodeBuilder().client(true).node();
Client client = node.client();
```

```
// on shutdown

node.close();
```

Another common usage is to start the **Node** and use the **Client** in unit/integration tests. In such a case, we would like to start a "local" **Node** (with a "local" discovery and transport). Again, this is just a matter of a simple setting when starting the **Node**. Note, "local" here means local on the JVM (well, actually class loader) level, meaning that two *local* servers started within the same JVM will discover themselves and form a cluster.

```
import static org.elasticsearch.node.NodeBuilder.*;

// on startup

Node node = nodeBuilder().local(true).node();
Client client = node.client();

// on shutdown

node.close();
```

Transport Client

The **TransportClient** connects remotely to an elasticsearch cluster using the transport module. It does not join the cluster, but simply gets one or more initial transport addresses and communicates with them in round robin fashion on each action (though most actions will probably be "two hop" operations).

```
// on startup

Client client = new TransportClient()
        .addTransportAddress(new InetSocketTransportAddress("host1", 9300))
        .addTransportAddress(new InetSocketTransportAddress("host2", 9300));

// on shutdown

client.close();
```

Note that you have to set the cluster name if you use one different to *elasticsearch*

```
Settings settings = ImmutableSettings.settingsBuilder()
            .put("cluster.name", "myClusterName").build();
Client client =     new TransportClient(settings);
//Add transport addresses and do something with the client...
```

The client allows to sniff the rest of the cluster, and add those into its list of machines to use. In this case, note that the ip addresses used will be the ones that the other nodes were started with (the "publish" address). In order to enable it, set the **client.transport.sniff** to **true**:

```
Settings settings = ImmutableSettings.settingsBuilder()
            .put("client.transport.sniff", true).build();
TransportClient client = new TransportClient(settings);
```

Other transport client level settings include:

Parameter	Description
client.transport.ignore_cluster_name	Set to **true** to ignore cluster name validation of connected nodes. (since 0.19.4)
client.transport.ping_timeout	The time to wait for a ping response from a node. Defaults to **5s**.
client.transport.nodes_sampler_interva	How often to sample / ping the nodes listed and connected. Defaults to **5s**.

Count

The count API allows to easily execute a query and get the number of matches for that query. It can be executed across one or more indices and across one or more types. The query can be provided using the *Query DSL*.

```
import static org.elasticsearch.index.query.xcontent.FilterBuilders.*;
import static org.elasticsearch.index.query.xcontent.QueryBuilders.*;

CountResponse response = client.prepareCount("test")
        .setQuery(termQuery("_type", "type1"))
        .execute()
        .actionGet();
```

For more information on the count operation, check out the REST *count* docs.

Operation Threading

The count API allows to set the threading model the operation will be performed when the actual execution of the API is performed on the same node (the API is executed on a shard that is allocated on the same server).

There are three threading modes.The **NO_THREADS** mode means that the count operation will be executed on the calling thread. The **SINGLE_THREAD** mode means that the count operation will be executed on a single different thread for all local shards. The **THREAD_PER_SHARD** mode means that the count operation will be executed on a different thread for each local shard.

The default mode is **SINGLE_THREAD**.

Delete By Query

The delete by query API allows to delete documents from one or more indices and one or more types based on a query. The query can either be provided the *Query DSL*. Here is an example:

```
import static org.elasticsearch.index.query.FilterBuilders.*;
import static org.elasticsearch.index.query.QueryBuilders.*;

DeleteByQueryResponse response = client.prepareDeleteByQuery("test")
        .setQuery(termQuery("_type", "type1"))
        .execute()
        .actionGet();
```

For more information on the delete by query operation, check out the *delete_by_query API* docs.

Delete

The delete API allows to delete a typed JSON document from a specific index based on its id. The following example deletes the JSON document from an index called twitter, under a type called tweet, with id valued 1:

```
DeleteResponse response = client.prepareDelete("twitter", "tweet", "1")
        .execute()
        .actionGet();
```

For more information on the delete operation, check out the *delete API* docs.

Operation Threading

The delete API allows to set the threading model the operation will be performed when the actual execution of the API is performed on the same node (the API is executed on a shard that is allocated on the same server).

The options are to execute the operation on a different thread, or to execute it on the calling thread (note that the API is still async). By default, **operationThreaded** is set to **true** which means the operation is executed on a different thread. Here is an example that sets it to **false**:

```
DeleteResponse response = client.prepareDelete("twitter", "tweet", "1")
        .setOperationThreaded(false)
        .execute()
        .actionGet();
```

Get

The get API allows to get a typed JSON document from the index based on its id. The following example gets a JSON document from an index called twitter, under a type called tweet, with id valued 1:

```
GetResponse response = client.prepareGet("twitter", "tweet", "1")
        .execute()
        .actionGet();
```

For more information on the index operation, check out the REST *get* docs.

Operation Threading

The get API allows to set the threading model the operation will be performed when the actual execution of the API is performed on the same node (the API is executed on a shard that is allocated on the same server).

The options are to execute the operation on a different thread, or to execute it on the calling thread (note that the API is still async). By default, **operationThreaded** is set to **true** which means the operation is executed on a different thread. Here is an example that sets it to **false**:

```
GetResponse response = client.prepareGet("twitter", "tweet", "1")
        .setOperationThreaded(false)
        .execute()
        .actionGet();
```

Index

The index API allows one to index a typed JSON document into a specific index and make it searchable. The following example indexes a JSON document into an index called twitter, under a type called tweet, with id valued 1:

```
import static org.elasticsearch.common.xcontent.XContentFactory.*;

IndexResponse response = client.prepareIndex("twitter", "tweet", "1")
        .setSource(jsonBuilder()
                    .startObject()
                        .field("user", "kimchy")
                        .field("postDate", new Date())
                        .field("message", "trying out Elastic Search")
                    .endObject()
                  )
        .execute()
        .actionGet();
```

The source to be indexed a json object that can be built easily using the elasticsearch **XContent** JSON Builder.

For more information on the index operation, check out the REST *index* docs.

Source Parameter

The source parameter represents a JSON object. It can be provided in different ways: as a native **byte[]**, as a **String**, as a byte array built using the **jsonBuilder**, or as a **Map** (that will be automatically converted to its JSON equivalent). Internally, each type is converted to **byte[]** (so a String is converted to a **byte[]**). Therefore, if the object is in this form already, then use it. The **jsonBuilder** is highly optimized JSON generator that directly constructs a **byte[]**.

Operation Threading

The index API allows to set the threading model the operation will be performed when the actual execution of the API is performed on the same node (the API is executed on a shard that is allocated on the same server).

The options are to execute the operation on a different thread, or to execute it on the calling thread (note that the API is still async). By default, **operationThreaded** is set to **true** which means the operation is executed on a different thread.

Percolate

The percolator allows to register queries against an index, and then send **percolate** requests which include a doc, and getting back the queries that match on that doc out of the set of registered queries.

Read the main *percolate* documentation before reading this guide.

```
//This is the query we're registering in the percolator
QueryBuilder qb = termQuery("content", "amazing");

//Index the query = register it in the percolator
client.prepareIndex("_percolator", "myIndexName", "myDesignatedQueryName")
    .setSource(qb)
    .setRefresh(true) //Needed when the query shall be available immediately
    .execute().actionGet();
```

This indexes the above term query under the name *myDesignatedQueryName*.

In order to check a document against the registered queries, use this code:

```
//Build a document to check against the percolator
XContentBuilder docBuilder = XContentFactory.jsonBuilder().startObject();
docBuilder.field("doc").startObject(); //This is needed to designate the document
docBuilder.field("content", "This is amazing!");
docBuilder.endObject(); //End of the doc field
docBuilder.endObject(); //End of the JSON root object
//Percolate
PercolateResponse response =
    client.preparePercolate("myIndexName", "myDocumentType").setSource(docBuilder).
↪execute().actionGet();
//Iterate over the results
for(String result : response) {
    //Handle the result which is the name of
    //the query in the percolator
}
```

Query Dsl

elasticsearch provides a full Java query dsl in a similar manner to the REST *Query DSL*. The factory for query builders is **QueryBuilders** and the factory for filter builders is **FilterBuilders**. Here is an example:

```
import static org.elasticsearch.index.query.FilterBuilders.*;
import static org.elasticsearch.index.query.QueryBuilders.*;

QueryBuilder qb1 = termQuery("name", "kimchy");

QueryBuilder qb2 = boolQuery()
                    .must(termQuery("content", "test1"))
                    .must(termQuery("content", "test4"))
                    .mustNot(termQuery("content", "test2"))
                    .should(termQuery("content", "test3"));

QueryBuilder qb3 = filteredQuery(
            termQuery("name.first", "shay"),
            rangeFilter("age")
                .from(23)
                .to(54)
                .includeLower(true)
                .includeUpper(false)
            );
```

The **QueryBuilder** can then be used with any API that accepts a query, such as **count** and **search**.

Search

The search API allows to execute a search query and get back search hits that match the query. It can be executed across one or more indices and across one or more types. The query can either be provided using the *Query DSL*. The body of the search request is built using the **SearchSourceBuilder**. Here is an example:

```
import static org.elasticsearch.index.query.FilterBuilders.*;
import static org.elasticsearch.index.query.QueryBuilders.*;

SearchResponse response = client.prepareSearch("test")
        .setSearchType(SearchType.DFS_QUERY_THEN_FETCH)
        .setQuery(termQuery("multi", "test"))
```

```
        .setFrom(0).setSize(60).setExplain(true)
        .execute()
        .actionGet();
```

For more information on the search operation, check out the REST *search* docs.

Using scrolls in Java

Read the scroll documentation first!

```
import static org.elasticsearch.index.query.FilterBuilders.*;
import static org.elasticsearch.index.query.QueryBuilders.*;

QueryBuilder qb = termQuery("multi", "test");

SearchResponse scrollResp = client.prepareSearch(test)
            .setSearchType(SearchType.SCAN)
            .setScroll(new TimeValue(60000))
            .setQuery(qb)
            .setSize(100).execute().actionGet(); //100 hits per shard will be␣
↪returned for each scroll
//Scroll until no hits are returned
while (true) {
    scrollResp = client.prepareSearchScroll(scrollResp.getScrollId()).setScroll(new␣
↪TimeValue(600000)).execute().actionGet();
    for (SearchHit hit : scrollResp.getHits()) {
        //Handle the hit...
    }
    //Break condition: No hits are returned
    if (scrollResp.hits().hits().length == 0) {
        break;
    }
}
```

Operation Threading

The search API allows to set the threading model the operation will be performed when the actual execution of the API is performed on the same node (the API is executed on a shard that is allocated on the same server).

There are three threading modes.The **NO_THREADS** mode means that the search operation will be executed on the calling thread. The **SINGLE_THREAD** mode means that the search operation will be executed on a single different thread for all local shards. The **THREAD_PER_SHARD** mode means that the search operation will be executed on a different thread for each local shard.

The default mode is **SINGLE_THREAD**.

Groovy Api

This section describes the Groovy API elasticsearch provides. All of elasticsearch APIs are executed using a *GClient*, and are completely asynchronous in nature (either accepts a listener, or return a future).

The Groovy API is a wrapper on top of the *Java API* exposing it in a groovier manner. The execution options for each API follow a similar manner and covered in *the anatomy of a Groovy API*.

Maven Repository

elasticsearch is hosted on Sonatype, with both a releases repo and a snapshots repo.

Anatomy

Once a *GClient* has been obtained, all of ElasticSearch APIs can be executed on it. Each Groovy API is exposed using three different mechanisms.

Closure Request

The first type is to simply provide the request as a Closure, which automatically gets resolved into the respective request instance (for the index API, its the **IndexRequest** class). The API returns a special future, called **GActionFuture**. This is a groovier version of elasticsearch Java **ActionFuture** (in turn a nicer extension to Java own **Future**) which allows to register listeners (closures) on it for success and failures, as well as blocking for the response. For example:

```
def indexR = client.index {
    index "test"
    type "type1"
    id "1"
    source {
        test = "value"
        complex {
            value1 = "value1"
            value2 = "value2"
        }
    }
}

println "Indexed $indexR.response.id into $indexR.response.index/$indexR.response.type
↪"
```

In the above example, calling **indexR.response** will simply block for the response. We can also block for the response for a specific timeout:

```
IndexResponse response = indexR.response "5s" // block for 5 seconds, same as:
response = indexR.response 5, TimeValue.SECONDS //
```

We can also register closures that will be called on success and on failure:

```
indexR.success = {IndexResponse response ->
    pritnln "Indexed $response.id into $response.index/$response.type"
}
indexR.failure = {Throwable t ->
    println "Failed to index: $t.message"
}
```

Request

This option allows to pass the actual instance of the request (instead of a closure) as a parameter. The rest is similar to the closure as a parameter option (the **GActionFuture** handling). For example:

```
def indexR = client.index (new IndexRequest(
        index: "test",
        type: "type1",
        id: "1",
        source: {
            test = "value"
            complex {
                value1 = "value1"
                value2 = "value2"
            }
        }))

println "Indexed $indexR.response.id into $indexR.response.index/$indexR.response.type
↪"
```

Java Like

The last option is to provide an actual instance of the API request, and an **ActionListener** for the callback. This is exactly like the Java API with the added **gexecute** which returns the **GActionFuture**:

```
def indexR = node.client.prepareIndex("test", "type1", "1").setSource({
    test = "value"
    complex {
        value1 = "value1"
        value2 = "value2"
    }
}).gexecute()
```

Client

Obtaining an elasticsearch Groovy **GClient** (a **GClient** is a simple wrapper on top of the Java **Client**) is simple. The most common way to get a client is by starting an embedded **Node** which acts as a node within the cluster.

Node Client

A Node based client is the simplest form to get a **GClient** to start executing operations against elasticsearch.

```
import org.elasticsearch.groovy.node.GNode
import org.elasticsearch.groovy.node.GNodeBuilder
import static org.elasticsearch.groovy.node.GNodeBuilder.*

// on startup

GNode node = nodeBuilder().node();
GClient client = node.client();

// on shutdown

node.close();
```

Since elasticsearch allows to configure it using JSON based settings, the configuration itself can be done using a closure that represent the JSON:

```
import org.elasticsearch.groovy.node.GNode
import org.elasticsearch.groovy.node.GNodeBuilder
import static org.elasticsearch.groovy.node.GNodeBuilder.*

// on startup

GNodeBuilder nodeBuilder = nodeBuilder();
nodeBuilder.settings {
    node {
        client = true
    }
    cluster {
        name = "test"
    }
}

GNode node = nodeBuilder.node()

// on shutdown

node.stop().close()
```

Count

The count API is very similar to the *Java count API*. The Groovy extension allows to provide the query to execute as a **Closure** (similar to GORM criteria builder):

```
def count = client.count {
    indices "test"
    types "type1"
    query {
        term {
            test = "value"
        }
    }
}
```

The query follows the same *Query DSL*.

Delete

The delete API is very similar to the *Java delete API*, here is an example:

```
def deleteF = node.client.delete {
    index "test"
    type "type1"
    id "1"
}
```

Get

The get API is very similar to the *Java get API*. The main benefit of using groovy is handling the source content. It can be automatically converted to a **Map** which means using Groovy to navigate it is simple:

```
def getF = node.client.get {
    index "test"
    type "type1"
    id "1"
}

println "Result of field2: $getF.response.source.complex.field2"
```

Index

The index API is very similar to the *Java index API*. The Groovy extension to it is the ability to provide the indexed source using a closure. For example:

```
def indexR = client.index {
    index "test"
    type "type1"
    id "1"
    source {
        test = "value"
        complex {
            value1 = "value1"
            value2 = "value2"
        }
    }
}
```

In the above example, the source closure itself gets transformed into an XContent (defaults to JSON). In order to change how the source closure is serialized, a global (static) setting can be set on the **GClient** by changing the **indexContentType** field.

Note also that the **source** can be set using the typical Java based APIs, the **Closure** option is a Groovy extension.

Search

The search API is very similar to the *Java search API*. The Groovy extension allows to provide the search source to execute as a **Closure** including the query itself (similar to GORM criteria builder):

```
def search = node.client.search {
    indices "test"
    types "type1"
    source {
        query {
            term(test: "value")
        }
    }
}

search.response.hits.each {SearchHit hit ->
    println "Got hit $hit.id from $hit.index/$hit.type"
}
```

It can also be execute using the "Java API" while still using a closure for the query:

```
def search = node.client.prepareSearch("test").setQuery({
        term(test: "value")
```

```
}).gexecute();

search.response.hits.each {SearchHit hit ->
    println "Got hit $hit.id from $hit.index/$hit.type"
}
```

The format of the search **Closure** follows the same JSON syntax as the *Search API* request.

CHAPTER 7

ES Appendix

Appendix:

Clients

—

Clients

- ElasticSearch.pm: Perl client.
- pyes: Python client.
- tire: Ruby API & DSL, with full Rails ActiveModel compatibility and mongoid integration thro‘mebla <https://github.com/cousine/mebla>‘_. la.
- Elastica: PHP client.
- rubberband: Ruby client.
- em-elasticsearch: elasticsearch library for eventmachine.
- elastic_searchable: Ruby client + Rails integration.
- erlastic_search: Erlang client.
- elasticsearch : PHP client.
- NEST: .NET client.
- ElasticSearch.NET: .NET client.
- pyelasticsearch: Python client.
- Elastisch: Clojure client.

Integrations

- Grails: ElasticSearch Grails plugin.
- escargot: ElasticSearch connector for Rails (WIP).
- Catalyst: ElasticSearch and Catalyst integration.
- django-elasticsearch: Django ElasticSearch Backend.
- elasticflume: Flume ume sink implementation.
- Terrastore Search: Terrastore re/ integration module with elasticsearch.
- Wonderdog: Hadoop bulk loader into elasticsearch.
- Play! Framework: Integrate with Play! Framework Application.
- ElasticaBundle: Symfony2 Bundle wrapping Elastica.
- Drupal: Drupal ElasticSearch integration.
- couch_es: elasticsearch helper for couchdb based products (apache couchdb, bigcouch & refuge)
- Jetty: Jetty HTTP Transport

Misc

- Puppet: elasticsearch puppet module.
- Chef: elasticsearch chef recipe.
- elasticsearch-rpms: RPMs for elasticsearch.
- daikon: Daikon ElasticSearch CLI
- Scrutineer: A high performance consistency checker to compare what you've indexed with your source of truth content (e.g. DB)

Front Ends

- elasticsearch-head: A web front end for an elastic search cluster.
- bigdesk: Live charts and statistics for elasticsearch cluster.

GitHub

GitHub is a place where a lot of development is done around *elasticsearch*, here is a simple search for repos.

Building From Source

elasticsearch codebase is hosted on github, and its just a **fork** away. To get started with the code, start by either forking or cloning the repo. One can also just download the master code in either zip or tar.gz.

Once downloaded, building an elasticsearch distribution is simple. You'll need Apache Maven or see below for IDE integration. Then compile ElasticSearch via:

```
$ mvn package -DskipTests
```

If you are running it for the first time, go get a cup of coffee (or better yet, a beer), it will take some time to download all the dependencies *elasticsearch* has. Once finished, a full distribution of the elasticsearch will be created under **target/releases/**.

In order to use it, just get either the **deb** package, the **zip** or **tar.gz** installation, extract it, and fire up **elasticsearch -f**. You now have a fully functional master based *elasticsearch* version running.

Hacking

Maven is supported by all major IDEs these days:

- IntelliJ IDEA * NetBeans * Eclipse - You'll need a maven plugin

CHAPTER 8

Glossary

analysis Analysis is the process of converting full *text* to *terms*. Depending on which analyzer is used, these phrases: "**FOO BAR**", "**Foo-Bar**", "**foo,bar**" will probably all result in the terms "**foo**" and "**bar**". These terms are what is actually stored in the index.

A full text query (not a *term* query) for "**FoO:bAR**" will also be analyzed to the terms "**foo**","**bar**" and will thus match the terms stored in the index.

It is this process of analysis (both at index time and at search time) that allows elasticsearch to perform full text queries.

Also see *text* and *term*.

cluster A cluster consists of one or more *nodes* which share the same cluster name. Each cluster has a single master node which is chosen automatically by the cluster and which can be replaced if the current master node fails.

document A document is a JSON document which is stored in elasticsearch. It is like a row in a table in a relational database. Each document is stored in an *index* and has a *type* and an *id*.

A document is a JSON object (also known in other languages as a hash / hashmap / associative array) which contains zero or more *fields*, or key-value pairs.

The original JSON document that is indexed will be stored in the **_source** *field*, which is returned by default when getting or searching for a document.

id The ID of a *document* identifies a document. The **index/type/id** of a document must be unique. If no ID is provided, then it will be auto-generated. (also see *routing*)

field A *document* contains a list of fields, or key-value pairs. The value can be a simple (scalar) value (eg a string, integer, date), or a nested structure like an array or an object. A field is similar to a column in a table in a relational database.

The *mapping* for each field has a field 'type' (not to be confused with document *type*) which indicates the type of data that can be stored in that field, eg **integer**, **string**, **object**. The mapping also allows you to define (amongst other things) how the value for a field should be analyzed.

index An index is like a 'database' in a relational database. It has a *mapping* which defines multiple *types*.

An index is a logical namespace which maps to one or more primary *shards* and can have zero or more replica *shards*.

mapping A mapping is like a 'schema definition' in a relational database. Each *index* has a mapping, which defines each *type* within the index, plus a number of index-wide settings.

A mapping can either be defined explicitly, or it will be generated automatically when a document is indexed.

node A node is a running instance of elasticsearch which belongs to a *cluster*. Multiple nodes can be started on a single server for testing purposes, but usually you should have one node per server.

At startup, a node will use unicast (or multicast, if specified) to discover an existing cluster with the same cluster name and will try to join that cluster.

primary shard Each document is stored in a single primary *shard*. When you index a document, it is indexed first on the primary shard, then on all *replicas* of the primary shard.

By default, an *index* has 5 primary shards. You can specify fewer or more primary shards to scale the number of *documents* that your index can handle.

You cannot change the number of primary shards in an index, once the index is created.

See also *routing*

replica shard Each primary *shard* can have zero or more replicas. A replica is a copy of the primary shard, and has two purposes:

increase failover: a replica shard can be promoted to a primary shard if the primary fails

increase performance: get and search requests can be handled by primary or replica shards.

By default, each primary shard has one replica, but the number of replicas can be changed dynamically on an existing index. A replica shard will never be started on the same node as its primary shard.

routing When you index a document, it is stored on a single primary *shard*. That shard is chosen by hashing the **routing** value. By default, the **routing** value is derived from the ID of the document or, if the document has a specified parent document, from the ID of the parent document (to ensure that child and parent documents are stored on the same shard).

This value can be overridden by specifying a **routing** value at index time, or a *routing field* in the *mapping*.

shard A shard is a single Lucene instance. It is a low-level "worker" unit which is managed automatically by elasticsearch. An index is a logical namespace which points to *primary* and *replica* shards.

Other than defining the number of primary and replica shards that an index should have, you never need to refer to shards directly. Instead, your code should deal only with an index.

Elasticsearch distributes shards amongst all *nodes* in the *cluster*, and can be move shards automatically from one node to another in the case of node failure, or the addition of new nodes.

source field By default, the JSON document that you index will be stored in the **_source** field and will be returned by all get and search requests. This allows you access to the original object directly from search results, rather than requiring a second step to retrieve the object from an ID.

Note: the exact JSON string that you indexed will be returned to you, even if it contains invalid JSON. The contents of this field do not indicate anything about how the data in the object has been indexed.

term A term is an exact value that is indexed in elasticsearch. The terms **foo**, **Foo**, **FOO** are NOT equivalent. Terms (ie exact values) can be searched for using 'term' queries.

See also *text* and *analysis*.

text Text (or full text) is ordinary unstructured text, such as this paragraph. By default, text will by *analyzed* into *terms*, which is what is actually stored in the index.

Text *fields* need to be analyzed at index time in order to be searchable as full text, and keywords in full text queries must be analyzed at search time to produce (and search for) the same terms that were generated at index time.

See also *term* and *analysis*.

type A type is like a 'table' in a relational database. Each type has a list of *fields* that can be specified for *documents* of that type. The *mapping* defines how each field in the document is analyzed.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W