SolrClient Documentation

Release 0.2.0

Nikita Vasilyev

May 19, 2018

Contents

SolrClient is a basic python library for Solr with several helper modules. Built in python3 with support for latest features of Solr 5. Development is heavily focused on indexing as well as parsing various query responses and returning them in native python data structures. Several helper classes will be built to automate querying and management of Solr clusters.

CHAPTER 1

Requirements

- python 3
- requests library (http://docs.python-requests.org/en/latest/)
- Solr

CHAPTER 2

Features

- Response Object to easily extract data from Solr Response
- Cursor Mark support
- Indexing: (raw JSON, JSON Files, gzipped JSON)
- Specify multiple hosts/IPs for SolrCloud for redundancy
- Basic Managed Schema field management
- Collection Reindexing/FS Export with cursorMark (Solr 4.9+)
- Tested against a real Solr instance

CHAPTER 3

Getting Started

Basic usage:

```
>>> from SolrClient import SolrClient
>>> solr = SolrClient('http://localhost:8983/solr')
>>> res = solr.query('SolrClient_unittest',{
        'q':'product_name:Lorem',
        'facet':True,
        'facet.field':'facet_test',
})
>>> res.get_results_count()
4
>>> res.get_facets()
{'facet_test': {'ipsum': 0, 'sit': 0, 'dolor': 2, 'amet,': 1, 'Lorem': 1}}
>>> res.get_facet_keys_as_list('facet_test')
['ipsum', 'sit', 'dolor', 'amet,', 'Lorem']
>>> res.docs
[{'product_name_exact': 'orci. Morbi ipsum ullamcorper, quam', '_version_':␣
↪15149272615480197 12, 'facet_test': ['dolor'], 'date': '2015-10-13T14:40:20.492Z',
↪'id': 'cb666bd1-ab8e-4951-98 29-5ccd4c12d10b', 'price': 10, 'product_name':
↪'ullamcorper, nulla. Vestibulum Lorem orci,'},  {'product_name_exact': 'enim␣
↪aliquet orci. sapien, mattis,', '_version_': 151492726156689408 0, 'facet_test': [
↪'dolor'], 'date': '2015-10-13T14:40:20.492Z', 'id': '8cb40255-ea07-4ab2-a30 f-
↪6e843781a043', 'price': 22, 'product_name': 'dui. Lorem ullamcorper, lacus.␣
↪hendrerit'}, {' product_name_exact': 'arcu In Nunc vel Nunc', '_version_':␣
↪15149272615689912 34, 'facet_test':  ['Lorem'], 'date': '2015-10-13T14:40:20.493Z',
↪'id': '287702d2-90b8-4dce-8e66-00a016e51bdd',  'price': 93, 'product_name': 'ipsum␣
↪vel. Lorem dui. risus'}, {'product_name_exact': 'Vivamus  sem ac dolor neque', '_
↪version_': 151492726165602304 0, 'facet_test': ['amet,'], 'date': '201 5-10-
↪13T14:40:20.494Z', 'id': 'f3c396f0-1fc2-4847-a966-1ebe055b8bd7', 'price': 60,
↪'product_n ame': 'consectetur Mauris dolor Lorem adipiscing'}]
```

See, so easy a caveman can do it.

CHAPTER 4

Components

4.1 SolrClient package

This is the primary module for interacting with Solr. All other components are accessed through a SolrClient instance. Basic usage:

```
>>> from SolrClient import SolrClient
>>> solr = SolrClient('http://localhost:8983/solr') #Solr URL
>>> res = solr.query('SolrClient_unittest',{ #Query params are sent as a python dict
        'q':'product_name:Lorem',
        'facet':True,
        'facet.field':'facet_test',
})
>>> res.get_results_count() #Number of items returned
4
>>> res.get_facets() #If you asked for facets it will give you facets as a python dict
{'facet_test': {'ipsum': 0, 'sit': 0, 'dolor': 2, 'amet,': 1, 'Lorem': 1}}
>>> res.docs #Returns documents as a list
[{'product_name_exact': 'orci. Morbi ipsum ullamcorper, quam', '_version_':␣
↪15149272615480197 12, 'facet_test': ['dolor'], 'date': '2015-10-13T14:40:20.492Z',
↪'id': 'cb666bd1-ab8e-4951-98 29-5ccd4c12d10b', 'price': 10, 'product_name':
↪'ullamcorper, nulla. Vestibulum Lorem orci,'},  {'product_name_exact': 'enim␣
↪aliquet orci. sapien, mattis,', '_version_': 151492726156689408 0, 'facet_test': [
↪'dolor'], 'date': '2015-10-13T14:40:20.492Z', 'id': '8cb40255-ea07-4ab2-a30 f-
↪6e843781a043', 'price': 22, 'product_name': 'dui. Lorem ullamcorper, lacus.␣
↪hendrerit'}, {' product_name_exact': 'arcu In Nunc vel Nunc', '_version_':␣
↪1514927261568991234, 'facet_test':  ['Lorem'], 'date': '2015-10-13T14:40:20.493Z',
↪'id': '287702d2-90b8-4dce-8e66-00a016e51bdd',  'price': 93, 'product_name': 'ipsum␣
↪vel. Lorem dui. risus'}, {'product_name_exact': 'Vivamus  sem ac dolor neque', '_
↪version_': 1514927261656023040, 'facet_test': ['amet,'], 'date': '201 5-10-
↪13T14:40:20.494Z', 'id': 'f3c396f0-1fc2-4847-a966-1ebe055b8bd7', 'price': 60,
↪'product_n ame': 'consectetur Mauris dolor Lorem adipiscing'}]
```

4.1.1 SolrClient.SolrClient module

class SolrClient.**SolrClient**(*host='http://localhost:8983/solr'*, *transport=<class SolrClient.transport.transportrequests.TransportRequests>*, *devel=False*, *auth=None*, *log=None*, ***kwargs*)

Bases: object

Creates a new SolrClient.

Parameters

- **host** – Specifies the location of Solr Server. ex 'http://localhost:8983/solr'. Can also take a list of host values in which case it will use the first server specified, but will switch over to the second one if the first one is not available.
- **transport** – Transport class to use. So far only requests is supported.
- **devel** (*bool*) – Can be turned on during development or debugging for a much greater logging. Requires logging to be configured with DEBUG level.

commit(*collection*, *openSearcher=False*, *softCommit=False*, *waitSearcher=True*, *commit=True*, ***kwargs*)

Parameters

- **collection** (*str*) – The name of the collection for the request
- **openSearcher** (*bool*) – If new searcher is to be opened
- **softCommit** (*bool*) – SoftCommit
- **waitServer** (*bool*) – Blocks until the new searcher is opened
- **commit** (*bool*) – Commit

Sends a commit to a Solr collection.

cursor_query(*collection*, *query*)

Parameters

- **collection** (*str*) – The name of the collection for the request.
- **query** (*dict*) – Dictionary of solr args.

Will page through the result set in increments using cursorMark until it has all items. Sort is required for cursorMark queries, if you don't specify it, the default is 'id desc'.

Returns an iterator of SolrResponse objects. For Example:

```
>>> for res in solr.cursor_query('SolrClient_unittest',{'q':'*:*'}):
        print(res)
```

delete_doc_by_id(*collection*, *doc_id*, ***kwargs*)

Parameters

- **collection** (*str*) – The name of the collection for the request
- **id** (*str*) – ID of the document to be deleted. Can specify '*' to delete everything.

Deletes items from Solr based on the ID.

```
>>> solr.delete_doc_by_id('SolrClient_unittest','changeme')
```

delete_doc_by_query(*collection*, *query*, ***kwargs*)

Parameters

- **collection** (*str*) – The name of the collection for the request
- **query** (*str*) – Query selecting documents to be deleted.

Deletes items from Solr based on a given query.

```
>>> solr.delete_doc_by_query('SolrClient_unittest','*:*')
```

get(*collection*, *doc_id*, ***kwargs*)

Parameters

- **collection** (*str*) – The name of the collection for the request
- **doc_id** (*str*) – ID of the document to be retrieved.

Retrieve document from Solr based on the ID.

```
>>> solr.get('SolrClient_unittest','changeme')
```

get_zk()

index(*collection*, *docs*, *params=None*, *min_rf=None*, ***kwargs*)

Parameters

- **collection** (*str*) – The name of the collection for the request.
- **list docs** (*docs*) – List of dicts. ex: [{"title": "testing solr indexing", "id": "test1"}]
- **int min_rf** (*min_rf*) – Required number of replicas to write to'

Sends supplied list of dicts to solr for indexing.

```
>>> docs = [{'id':'changeme','field1':'value1'}, {'id':'changeme1','field2':
↪'value2'}]
>>> solr.index('SolrClient_unittest', docs)
```

index_json(*collection*, *data*, *params=None*, *min_rf=None*, ***kwargs*)

Parameters

- **collection** (*str*) – The name of the collection for the request.
- **str data** (*data*) – Valid Solr JSON as a string. ex: '[{"title": "testing solr indexing", "id": "test1"}]'
- **int min_rf** (*min_rf*) – Required number of replicas to write to'

Sends supplied json to solr for indexing, supplied JSON must be a list of dictionaries.

```
>>> docs = [{'id':'changeme','field1':'value1'},
            {'id':'changeme1','field2':'value2'}]
>>> solr.index_json('SolrClient_unittest',json.dumps(docs))
```

local_index(*collection*, *filename*, ***kwargs*)

Parameters

- **collection** (*str*) – The name of the collection for the request
- **filename** (*str*) – String file path of the file to index.

Will index specified file into Solr. The *file* must be local to the server, this is faster than other indexing options. If the files are already on the servers I suggest you use this. For example:

```
>>> solr.local_index('SolrClient_unittest',
                           '/local/to/server/temp_file.json')
```

mget(*collection*, *doc_ids*, ***kwargs*)

Parameters

- **collection** (*str*) – The name of the collection for the request
- **doc_ids** (*tuple*) – ID of the document to be retrieved.

Retrieve documents from Solr based on the ID.

```
>>> solr.get('SolrClient_unittest','changeme')
```

paging_query(*collection*, *query*, *rows=1000*, *start=0*, *max_start=200000*)

Parameters

- **collection** (*str*) – The name of the collection for the request.
- **query** (*dict*) – Dictionary of solr args.
- **rows** (*int*) – Number of rows to return in each batch. Default is 1000.
- **start** (*int*) – What position to start with. Default is 0.
- **max_start** (*int*) – Once the start will reach this number, the function will stop. Default is 200000.

Will page through the result set in increments of *row* WITHOUT using cursorMark until it has all items or until *max_start* is reached. Use max_start to protect your Solr instance if you are not sure how many items you will be getting. The default is 200,000, which is still a bit high.

Returns an iterator of SolrResponse objects. For Example:

```
>>> for res in solr.paging_query('SolrClient_unittest',{'q':'*:*'}):
        print(res)
```

query(*collection*, *query*, *request_handler='select'*, ***kwargs*)

Parameters

- **collection** (*str*) – The name of the collection for the request
- **request_handler** (*str*) – Request handler, default is 'select'
- **query** (*dict*) – Python dictonary of Solr query parameters.

Sends a query to Solr, returns a SolrResults Object. *query* should be a dictionary of solr request handler arguments. Example:

```
res = solr.query('SolrClient_unittest',{
    'q':'*:*',
    'facet':True,
    'facet.field':'facet_test',
})
```

query_raw(*collection*, *query*, *request_handler='select'*, ***kwargs*)

Parameters

- **collection** (*str*) – The name of the collection for the request
- **request_handler** (*str*) – Request handler, default is 'select'
- **query** (*dict*) – Python dictionary of Solr query parameters.

Sends a query to Solr, returns a dict. *query* should be a dictionary of solr request handler arguments. Example:

```
res = solr.query_raw('SolrClient_unittest',{
    'q':'*:*',
    'facet':True,
    'facet.field':'facet_test',
})
```

stream_file(*collection*, *filename*, ***kwargs*)

Parameters

- **collection** (*str*) – The name of the collection for the request
- **filename** (*str*) – Filename of json file to index.

Will open the json file, uncompressing it if necessary, and submit it to specified solr collection for indexing.

```
>>> solr.local_index('SolrClient_unittest',
                           '/local/to/script/temp_file.json')
```

4.2 SolrClient.SolrResponse module

SolrResponse object is returned on queries to Solr. It provides several handy methods for getting the data back.

class SolrClient.**SolrResponse**(*data*)

get_cursor()

If you asked for the cursor in your query, this will return the next cursor mark.

get_facet_keys_as_list(*field*)

Parameters **field** (*str*) – Name of facet field to retrieve keys from.

Similar to get_facet_values_as_list but returns the list of keys as a list instead. Example:

```
>>> r.get_facet_keys_as_list('facet_test')
['Lorem', 'ipsum', 'amet,', 'dolor', 'sit']
```

get_facet_pivot()

Parses facet pivot response. Example::

```
>>> res = solr.query('SolrClient_unittest',{
'q':'*:*',
'fq':'price:[50 TO *]',
'facet':True,
'facet.pivot':'facet_test,price' #Note how there is no space between␣
↪fields. They are just separated by commas
})
>>> res.get_facet_pivot()
{'facet_test,price': {'Lorem': {89: 1, 75: 1}, 'ipsum': {53: 1, 70: 1,␣
↪55: 1, 89: 1, 74: 1, 93: 1, 79: 1}, 'dolor': {61: 1, 94: 1}, 'sit':␣
↪{99: 1, 50: 1, 67: 1, 52: 1, 54: 1, 71: 1, 72: 1, 84: 1, 62: 1}, 'amet,
↪': {68: 1}}}
```

(continues on next page)

(continued from previous page)

This method has built in recursion and can support indefinite number of facets. However, note that the output format is significantly massaged since Solr by default outputs a list of fields in each pivot field.

get_facet_values_as_list(*field*)

Parameters **field** (*str*) – Name of facet field to retrieve values from.

Returns facet values as list for a given field. Example:

```
>>> res = solr.query('SolrClient_unittest',{
    'q':'*:*',
    'facet':'true',
    'facet.field':'facet_test',
})
>>> res.get_facet_values_as_list('facet_test')
[9, 6, 14, 10, 11]
>>> res.get_facets()
{'facet_test': {'Lorem': 9, 'ipsum': 6, 'amet,': 14, 'dolor': 10, 'sit': 11}}
```

get_facets()

Returns a dictionary of facets:

```
>>> res = solr.query('SolrClient_unittest',{
        'q':'product_name:Lorem',
        'facet':True,
        'facet.field':'facet_test',
})... ... ... ...
>>> res.get_results_count()
4
>>> res.get_facets()
{'facet_test': {'ipsum': 0, 'sit': 0, 'dolor': 2, 'amet,': 1, 'Lorem': 1}}
```

get_facets_ranges()

Returns query facet ranges

```
>>> res = solr.query('SolrClient_unittest',{
    'q':'*:*',
    'facet':True,
    'facet.range':'price',
    'facet.range.start':0,
    'facet.range.end':100,
    'facet.range.gap':10
    })
>>> res.get_facets_ranges()
{'price': {'80': 9, '10': 5, '50': 3, '20': 7, '90': 3, '70': 4, '60': 7, '0
↪': 3, '40': 5, '30': 4}}
```

get_field_values_as_list(*field*)

Parameters **field** (*str*) – The name of the field for which to pull in values.

Will parse the query results (must be ungrouped) and return all values of 'field' as a list. Note that these are not unique values. Example:

```
>>> r.get_field_values_as_list('product_name_exact')
['Mauris risus risus lacus. sit', 'dolor auctor Vivamus fringilla. vulputate',
↪ 'semper nisi lacus nulla sed', 'vel amet diam sed posuere', 'vitae neque␣
↪ultricies, Phasellus ac', 'consectetur nisi orci, eu diam', 'sapien, nisi␣
↪accumsan accumsan In', 'ligula. odio ipsum sit vel', 'tempus orci. elit, Ut␣
↪nisl.', 'neque nisi Integer nisi Lorem']
```

(continues on next page)

(continued from previous page)

get_first_field_values_as_list(*field*)

Parameters **field** (*str*) – The name of the field for lookup.

Goes through all documents returned looking for specified field. At first encounter will return the field's value.

get_flat_groups(*field=None*)
Flattens the group response and just returns a list of documents.

get_groups_count(*field=None*)
Returns 'matches' from group response.

If grouping on more than one field, provide the field argument to specify which count you are looking for.

get_json()
Returns json from the original response.

get_jsonfacet_counts_as_dict(*field*, *data=None*)
EXPERIMENTAL Takes facets and returns then as a dictionary that is easier to work with, for example, if you are getting something this:

```
{'facets': {'count': 50,
  'test': {'buckets': [{'count': 10,
     'pr': {'buckets': [{'count': 2, 'unique': 1, 'val': 79},
       {'count': 1, 'unique': 1, 'val': 9}]},
     'pr_sum': 639.0,
     'val': 'consectetur'},
    {'count': 8,
     'pr': {'buckets': [{'count': 1, 'unique': 1, 'val': 9},
       {'count': 1, 'unique': 1, 'val': 31},
       {'count': 1, 'unique': 1, 'val': 33}]},
     'pr_sum': 420.0,
     'val': 'auctor'},
    {'count': 8,
     'pr': {'buckets': [{'count': 2, 'unique': 1, 'val': 94},
       {'count': 1, 'unique': 1, 'val': 25}]},
     'pr_sum': 501.0,
     'val': 'nulla'}]}}}
```

This should return you something like this:

```
{'test': {'auctor': {'count': 8,
                     'pr': {9: {'count': 1, 'unique': 1},
                            31: {'count': 1, 'unique': 1},
                            33: {'count': 1, 'unique': 1}},
                     'pr_sum': 420.0},
          'consectetur': {'count': 10,
                          'pr': {9: {'count': 1, 'unique': 1},
                                 79: {'count': 2, 'unique': 1}},
                          'pr_sum': 639.0},
          'nulla': {'count': 8,
                    'pr': {25: {'count': 1, 'unique': 1},
                           94: {'count': 2, 'unique': 1}},
                    'pr_sum': 501.0}}}
```

get_ngroups (*field=None*)
Returns ngroups count if it was specified in the query, otherwise ValueError.

If grouping on more than one field, provide the field argument to specify which count you are looking for.

get_num_found()
Returns number of documents found on an ungrounded query.

```
>>> res = solr.query('SolrClient_unittest',{
        'q':'*:*',
        'facet':True,
        'facet.field':'facet_test',
})
>>> res.get_num_found()
50
```

get_results_count()
Returns the number of documents returned in current query.

```
>>> res = solr.query('SolrClient_unittest',{
    'q':'*:*',
    'facet':True,
    'facet.field':'facet_test',
    })
>>>
>>> res.get_results_count()
10
>>> res.get_num_found()
50
```

json_facet (*field=None*)
EXPERIMENTAL

Tried to kick back the json.fact output.

4.3 SolrClient.IndexQ module

Really simple filesystem based queue for de-coupling data getting/processing from indexing into solr. All in all, this module is nothing special, but I suspect a lot of people write something similar; so maybe it will save you some time.

For example, lets say you are working with some data processing that exhibits the following:

- Outputs a large number of items
- These items are possibly small
- You want to process them as fast as possible
- They don't have to be indexed right away

Log parsing is a good example; if you wanted to parse a log file and index that data into Solr you would not send an individual update request for each line and instead aggregate them into something more substantial. This can especially become a problem if you are parsing log files with some parallelism.

This is the issue that this sub module resolves. It allows you to create a quick file system based queue of items and then index them into Solr later. It will also maintain an internal buffer and add items to it until a specific size is reached before writing it out to the file system.

Here is the really basic example to illustrate the concept.:

```
>>> from SolrClient import SolrClient, IndexQ
>>> index = IndexQ('.','testq',size=1)
>>> index.add({'id':'test1'})
17 #By default it returns the buffer offset
>>> index.get_all_as_list()
[]
>>> index.add(finalize=True)
'./testq/todo/testq_2015-10-20-19-7-58-5219.json' #If file was written it will return␣
↪the filename
>>> index.get_all_as_list()
['./testq/todo/testq_2015-10-20-19-7-58-5219.json']
>>> solr = SolrClient('http://localhost:7050/solr')
>>> index.index(solr,'SolrClient_unittest')
```

Note that you don't have to track the output of add method, it is just there to give you a better idea of what it is doing. You can also specify threads to index method to run this quicker, by default it will use one thread. There is also some logging to provide you a better idea of what it is doing.

class SolrClient.**IndexQ**(*basepath*, *queue*, *compress=False*, *compress_complete=False*, *size=0*, *devel=False*, *threshold=0.9*, *log=None*, *rotate_complete=None*, ***kwargs*)

IndexQ sub module will help with indexing content into Solr. It can be used to de-couple data processing from indexing.

Each queue is set up with the following directory structure queue_name/

- todo/
- done/

Items get saved to the todo directory and once an item is processed it gets moved to the done directory. Items are processed in chronological order.

add(*item=None*, *finalize=False*, *callback=None*)

Takes a string, dictionary or list of items for adding to queue. To help troubleshoot it will output the updated buffer size, however when the content gets written it will output the file path of the new file. Generally this can be safely discarded.

Parameters

- **item** (*<dict, list>*) – Item to add to the queue. If dict will be converted directly to a list and then to json. List must be a list of dictionaries. If a string is submitted, it will be written out as-is immediately and not buffered.
- **finalize** (*bool*) – If items are buffered internally, it will flush them to disk and return the file name.
- **callback** – A callback function that will be called when the item gets written to disk. It will be passed one position argument, the file path of the file written. Note that errors from the callback method will not be re-raised here.

complete(*filepath*)

Marks the item as complete by moving it to the done directory and optionally gzipping it.

get_all_as_list(*dir='_todo_dir'*)

Returns a list of the the full path to all items currently in the todo directory. The items will be listed in ascending order based on filesystem time. This will re-scan the directory on each execution.

Do not use this to process items, this method should only be used for troubleshooting or something axillary. To process items use get_todo_items() iterator.

get_all_json_from_indexq()

Gets all data from the todo files in indexq and returns one huge list of all data.

get_multi_q(*sentinel='STOP'*)

This helps indexq operate in multiprocessing environment without each process having to have it's own IndexQ. It also is a handy way to deal with thread / process safety.

This method will create and return a JoinableQueue object. Additionally, it will kick off a back end process that will monitor the queue, de-queue items and add them to this indexq.

The returned JoinableQueue object can be safely passed to multiple worker processes to populate it with data.

To indicate that you are done writing the data to the queue, pass in the sentinel value ('STOP' by default).

Make sure you call join_indexer() after you are done to close out the queue and join the worker.

get_todo_items(***kwargs*)

Returns an iterator that will provide each item in the todo queue. Note that to complete each item you have to run complete method with the output of this iterator.

That will move the item to the done directory and prevent it from being retrieved in the future.

index(*solr*, *collection*, *threads=1*, *send_method='stream_file'*, ***kwargs*)

Will index the queue into a specified solr instance and collection. Specify multiple threads to make this faster, however keep in mind that if you specify multiple threads the items may not be in order. Example:

```
solr = SolrClient('http://localhost:8983/solr/')
for doc in self.docs:
    index.add(doc, finalize=True)
index.index(solr,'SolrClient_unittest')
```

Parameters

- **solr** (*object*) – SolrClient object.
- **collection** (*string*) – The name of the collection to index document into.
- **threads** (*int*) – Number of simultaneous threads to spin up for indexing.
- **send_method** (*string*) – SolrClient method to execute for indexing. Default is stream_file

join_indexer()

4.4 SolrClient.Schema module

Solr Schema component for basic interations:

```
>>> field = {'name':'fieldname','stored':True,'indexed':True,'type':'tdate'}
>>> solr.schema.create_field('SolrClient_unittest', field)
{'responseHeader': {'status': 0, 'QTime': 85}}
>>> solr.schema.does_field_exist('SolrClient_unittest','fieldname')
True
>>> res = solr.schema.create_field('SolrClient_unittest', field)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "C:\Users\Nick\Documents\GitHub\SolrClient\SolrClient\schema.py", line 43, in␣
↪create_field
        raise ValueError("Field {} Already Exists in Solr Collection {}".format(field_
↪dict['name'],collection))
ValueError: Field fieldname Already Exists in Solr Collection SolrClient_unittest
```

class SolrClient.**Schema**(*solr*)

Class for interacting with Solr collections that are using data driven schemas. At this point there are basic methods for creating/deleting fields, contributions to this class are very welcome.

More info on Solr can be found here: https://cwiki.apache.org/confluence/display/solr/Schema+API

create_copy_field(*collection*, *copy_dict*)

Creates a copy field.

copy_dict should look like

```
{'source':'source_field_name','dest':'destination_field_name'}
```

Parameters

- **collection** (*string*) – Name of the collection for the action
- **copy_field** (*dict*) – Dictionary of field info

Reference: https://cwiki.apache.org/confluence/display/solr/Schema+API#SchemaAPI-AddaNewCopyFieldRule

create_field(*collection*, *field_dict*)

Creates a new field in managed schema, will raise ValueError if the field already exists. field_dict should look like this:

```
{
     "name":"sell-by",
     "type":"tdate",
     "stored":True
}
```

Reference: https://cwiki.apache.org/confluence/display/solr/Defining+Fields

delete_copy_field(*collection*, *copy_dict*)

Deletes a copy field.

copy_dict should look like

```
{'source':'source_field_name','dest':'destination_field_name'}
```

Parameters

- **collection** (*string*) – Name of the collection for the action
- **copy_field** (*dict*) – Dictionary of field info

delete_field(*collection*, *field_name*)

Deletes a field from the Solr Collection. Will raise ValueError if the field doesn't exist.

Parameters

- **collection** (*string*) – Name of the collection for the action
- **field_name** (*string*) – String name of the field.

does_field_exist(*collection*, *field_name*)

Checks if the field exists will return a boolean True (exists) or False(doesn't exist).

Parameters

- **collection** (*string*) – Name of the collection for the action

- **field_name** (*string*) – String name of the field.

get_schema_copyfields(*collection*)

get_schema_fields(*collection*)
Returns Schema Fields from a Solr Collection

replace_field(*collection*, *field_dict*)
Replace a field in managed schema, will raise ValueError if the field does not exist. field_dict as in create_field(...).

Parameters

- **collection** (*string*) –
- **field_dict** (*dict*) –

4.5 SolrClient.Reindexer module

This helper class uses Solr's cursorMark (Solr 4.9+) to re-index collections or to dump out your collection to filesystem. It is useful if you want to get an offline snapshot of your data. Additionally, you will need to re-index your data to upgrade lucene indexes and this is a handy way to do it.

In the most basic way; it will sort your items by id and page through the results in batches of *rows* and concurrently send data to the destiation. Destination can be either another Solr collection or an IndexQ instance. If it is another Solr collection you have to make sure that it is configured exactly as the first one. Keep in mind that if items are added or modified while you are performing this operation; they may not be captured. So it is advised to stop indexing while you are running it.

If you are keeping the document's index timestamp, with something like:

```
<field name="last_update" type="date" indexed="true" stored="true" default="NOW" />
```

You can specify that field through *date_field* parameter. If it is supplied, the Reindexer will include the date_field in the sort and start re-indexing starting with the oldest documents. This way new items will also be picked up. Note that deletions will not be carried over so it is still adviced to stop indexing.

Using this will also allows you to resume the reindexing if it gets interrupted for some reason through the *resume* method.

On the resume, it will run several range facet queries to compare the counts based on date ranges and only re-process the ranges that have missing documents.

class SolrClient.helpers.**Reindexer**(*source*, *dest*, *source_coll=None*, *dest_coll=None*, *rows=1000*, *date_field=None*, *devel=False*, *per_shard=False*, *ignore_fields=['_version_']*)

Initiates the re-indexer.

Parameters

- **source** – An instance of SolrClient.
- **dest** – An instance of SolrClient or an instance of IndexQ.
- **source_coll** (*string*) – Source collection name.
- **dest_coll** (*string*) – Destination collection name; only required if destination is SolrClient.
- **rows** (*int*) – Number of items to get in each query; default is 1000, however you will probably want to increase it.

- **date_field** (*string*) – String name of a Solr date field to use in sort and resume.
- **devel** (*bool*) – Whenever to turn on super verbouse logging for development. Standard DEBUG should suffice for most developemnt.
- **per_shard** (*bool*) – Will add distrib=false to each query to get the data. Use this only if you will be running multiple instances of this to get the rest of the shards.
- **ignore_fields** (*list*) – What fields to exclude from Solr queries. This is important since if you pull them out, you won't be able to index the documents in.

By default, it will try to determine and exclude copy fields as well as _version_. Pass in your own list to override or set it to False to prevent it from doing anything.

reindex(*fq=[]*, ***kwargs*)

Starts Reindexing Process. All parameter arguments will be passed down to the getter function. :param string fq: FilterQuery to pass to source Solr to retrieve items. This can be used to limit the results.

resume(*start_date=None*, *end_date=None*, *timespan='DAY'*, *check=False*)

This method may help if the original run was interrupted for some reason. It will only work under the following conditions * You have a date field that you can facet on * Indexing was stopped for the duration of the copy

The way this tries to resume re-indexing is by running a date range facet on the source and destination collections. It then compares the counts in both collections for each timespan specified. If the counts are different, it will re-index items for each range where the counts are off. You can also pass in a start_date to only get items after a certain time period. Note that each date range will be indexed in it's entirety, even if there is only one item missing.

Keep in mind this only checks the counts and not actual data. So make the indexes weren't modified between the reindexing execution and running the resume operation.

Parameters **start_date** – Date to start indexing from. If not specified there will be no restrictions and all data will be processed. Note that

this value will be passed to Solr directly and not modified. :param end_date: The date to index items up to. Solr Date Math compliant value for faceting; currenlty only DAY is supported. :param timespan: Solr Date Math compliant value for faceting; currenlty only DAY is supported. :param check: If set to True it will only log differences between the two collections without actually modifying the destination.

4.6 SolrClient.Collections module

Solr Collections api interface.

class SolrClient.**Collections**(*solr*, *log*)

Provides an interface to Solr Collections API.

api(*action*, *args=None*)

Sends a request to Solr Collections API. Documentation is here: https://cwiki.apache.org/confluence/display/solr/Collections+API

Parameters

- **action** (*string*) – Name of the collection for the action
- **args** (*dict*) – Dictionary of specific parameters for action

check_status(*ignore=()*, *status=None*)

Checks status of each collection and shard to make sure that:

1. Cluster state is active
2. Number of docs matches across replicas for a given shard.

Returns a dict of results for custom alerting.

cluster_status_raw(***kwargs*)
Returns raw output of the clusterstatus api command.

clusterstatus()
Returns a slightly slimmed down version of the clusterstatus api command. It also gets count of documents in each shard on each replica and returns it as doc_count key for each replica.

create(*name*, *numShards*, *params=None*)
Create a new collection.

exists(*collection*)
Return True if a collection exists.

list()
Returns a list[string] of all collection names on the cluster.

4.7 SolrClient.ZK module

Solr ZK api interface.

class SolrClient.**ZK**(*solr*, *log*)

check_zk()
Will attempt to telnet to each zookeeper that is used by SolrClient and issue 'mntr' command. Response is parsed to check to see if the zookeeper node is a leader or a follower and returned as a dict.

If the telnet collection fails or the proper response is not parsed, the zk node will be listed as 'down' in the dict. Desired values are either follower or leader.

copy_config(*original*, *new*)
Copies collection configs into a new folder. Can be used to create new collections based on existing configs.

Basically, copies all nodes under /configs/original to /configs/new.

Parameters

- **str** (*new*) – ZK name of original config
- **str** – New name of the ZK config.

download_collection_configs(*collection*, *fs_path*)
Downloads ZK Directory to the FileSystem.

Parameters

- **str** (*fs_path*) – Name of the collection (zk config name)
- **str** – Destination filesystem path.

get_item(*path*)
Returns a specified zookeeper node.

upload_collection_configs(*collection*, *fs_path*)
Uploads collection configurations from a specified directory to zookeeper.

CHAPTER 5

Roadmap

- Better test coverage
- HTTPS Support
- urllib support
- Collection Alias Management
- IndexManager for storing indexing documents off-line and batch indexing them
- More Schema API Action Calls
- Collections API Support

CHAPTER 6

Contributing

I've realized that that there isn't really a well maintained Solr Python library I liked so I put this together. Contributions (code, tests, documentation) are definitely welcome; if you have a question about development please open up an issue on github page. If you have a pull request, please make sure to add tests and that all of them pass before submitting.

Python Module Index

S

Index