scorched Documentation

Release 0.1

Josip Delic

October 24, 2016

Contents

Contents:

CHAPTER 1

First steps

1.1 Installing scorched

You can install scorched via setuptools, pip.

To use scorched, you'll need an Apache Solr installation. Scorched currently requires at least version 3.6.1 of Apache Solr.

1.1.1 Using pip

If you have pip installed, just type:

```
$ pip install scorched
```

If you've got an old version of scorched installed, and want to upgrade, then type:

```
$ pip install -U scorched
```

That's all you need to do; all dependencies will be pulled in automatically.

1.2 Configuring a connection

Whether you're querying or updating a Solr server, you need to set up a connection to the Solr server. Pass the URL of the Solr server to a SolrInterface object.

```
>>> import scorched
>>> si = scorched.SolrInterface("http://localhost:8983/solr/")
```

Note: Optional arguments to connection: *scorched.connection.SolrConnection*

1.3 Adding documents

To add data to the scorched instance use a Python dictionary.

```
>>> document = {"id":"0553573403",
...              "cat":"book",
...              "name":"A Game of Thrones",
...              "price":7.99,
...              "inStock": True,
...              "author_t":
...              "George R.R. Martin",
...              "series_t":"A Song of Ice and Fire",
...              "sequence_i":1,
...              "genre_s":"fantasy"}
>>> si.add(document)
```

You can add lists of dictionaries in the same way. Given the example "books.json" file, you could feed it to scorched like so:

```
>>> file = os.path.join(os.path.dirname(__file__), "dumps",
...                     "books.json")
>>> with open(file) as f:
...     datajson = f.read()
...     docs = json.loads(self.datajson)
>>> si.add(docs)
>>> si.commit()
```

Note: Optional arguments to add:

See http://wiki.apache.org/solr/UpdateXmlMessages for details. Or the api documentation: TODO link

1.4 Deleting documents

You can delete documents individually, or delete all documents resulting from a query.

To delete documents individually, you need to pass a list of the document ids to scorched.

```
>>> si.delete_by_ids([obj.id])
>>> si.delete_by_ids([x.id for x in objs])
```

To delete documents by query, you construct one or more queries from *Q* objects, in the same way that you construct a query as explained in optional-terms. You then pass those query into the `delete_by_query()` method:

```
>>> si.delete_by_query(query=si.Q("game"))
```

To clear the entire index, there is a shortcut which simply deletes every document in the index.

```
>>> si.delete_all()
```

Deletions, like additions, only take effect after a commit (or autocommit).

Note: Optional arguments to delete:

See http://wiki.apache.org/solr/UpdateXmlMessages for details. Or the api documentation: TODO link

1.5 Optimizing

After updating an index with new data, it becomes fragmented and performance suffers. This means that you need to optimize the index. When and how often you do this is something you need to decide on a case by case basis. If you only add data infrequently, you should optimize after every new update; if you trickle in data on a frequent basis, you need to think more about it. See http://wiki.apache.org/solr/SolrPerformanceFactors#Optimization_Considerations.

Either way, to optimize an index, simply call:

```
>>> si.optimize()
```

A Solr optimize also performs a commit, so if you're about to `optimize()` anyway, you can leave off the preceding `commit()`. It doesn't particularly hurt to do both though.

1.6 Rollback

If you haven't yet added/deleted documents since the last commit, you can issue a rollback to revert the index state to that of the last commit.

```
>>> si.rollback()
```

CHAPTER 2

Querying

For the examples in this chapter, I'll be assuming that you've loaded your server up with the books data supplied with the example Solr setup.

The data itself you can see at `$SOLR_SOURCE_DIR/example/exampledocs/books.json`. To load it into a server running with the example schema:

```
$ cd example/exampledocs
$ curl 'http://localhost:8983/solr/update/json?commit=true' --data-binary \
@exampledocs/books.json -H 'Content-type:application/json'
```

2.1 Searching solr

Scorched uses a chaining API, and will hopefully look quite familiar to anyone who has used the Django ORM.

The `books.json` data looked like this:

```
[
    {
        "id" : "978-0641723445",
        "cat" : ["book","hardcover"],
        "name" : "The Lightning Thief",
        "author" : "Rick Riordan",
        "series_t" : "Percy Jackson and the Olympians",
        "sequence_i" : 1,
        "genre_s" : "fantasy",
        "inStock" : true,
        "price" : 12.50,
        "pages_i" : 384
    }
...
]
```

Note: Dynamic fields.

Dynamic fields are named with a suffix (*_i, *_t, *_s).

A simple search for one word, in the default search field.

```
>>> si.query("thief")
```

Maybe you want to search in the (non-default) field author for authors called Martin

```
>>> si.query(author="rick")
```

Maybe you want to search for books with "thief" in their title, by an author called "rick".

```
>>> si.query(name="thief", author="rick")
```

Perhaps your initial, default, search is more complex, and has more than one word in it:

```
>>> si.query(name="lightning").query(name="thief")
```

A easy way to see what sunburnt is producing is to call `options`:

```
>>> si.query(name="lightning").query(name="thief").options()
{'q': u'name:lightning AND name:thief'}
```

2.2 Executing queries

Scorched is lazy in constructing queries. The examples in the previous section don't actually perform the query - they just create a "query object" with the correct parameters. To actually get the results of the query, you'll need to execute it:

```
>>> response = si.query("thief").execute()
```

This will return a `SolrResponse` object. If you treat this object as a list, then each member of the list will be a document, in the form of a Python dictionary containing the relevant fields:

For example, if you run the first example query above, you should see a response like this:

```
>>> for result in si.query("thief").execute():
...     print result
{
    u'name': u'The Lightning Thief',
    u'author': u'Rick Riordan',
    u'series_t': u'Percy Jackson and the Olympians',
    u'pages_i': 384,
    u'genre_s': u'fantasy',
    u'author_s': u'Rick Riordan',
    u'price': 12.5,
    u'price_c': u'12.5,USD',
    u'sequence_i': 1,
    u'inStock': True,
    u'_version_': 1462820023761371136,
    u'cat': [u'book', u'hardcover'],
    u'id': u'978-0641723445'
}
```

Of course, often you don't want your results in the form of a dictionary, you want an object. Perhaps you have the following class defined in your code:

```
>>> class Book:
...     def __init__(self, name, author, **other_kwargs):
...         self.title = name
...         self.author = author
...         self.other_kwargs = other_kwargs
...
```

```
...     def __repr__(self):
...         return 'Book("%s", "%s")' % (self.title, self.author)
```

You can tell scorched to give you `Book` instances back by telling `execute()` to use the class as a constructor.

```
>>> for result in si.query("game").execute(constructor=Book):
...     print result
Book("The Lightning Thief", "Rick Riordan")
```

The `constructor` argument most often will be a class, but it can be any callable; it will always be called as `constructor(**response_dict)`.

You can extract more information from the response than simply the list of results. The SolrResponse object has the following attributes:

- `response.status` : status of query. (status != 0 something went wrong).
- `response.QTime` : how long did the query take in milliseconds.
- `response.params` : the params that were used in the query.

and the results themselves are in the following attributes

- `response.result` : the results of your main query.
- `response.result.groups` : see **'Result greater'_** below.
- `response.facet_counts` : see *Faceting* below.
- `response.highlighting` : see *Highlighting* below.
- `response.more_like_these` : see *More Like This* below.

Finally, `response.result` itself has the following attributes

- `response.result.numFound` : total number of docs found in the index.
- `response.result.docs` : the actual results themselves.
- **`response.result.start`** [if the number of docs is less than numFound,] then this is the pagination offset.

2.3 Pagination

By default, Solr will only return the first 10 results (this is configurable in `schema.xml`). To get at more results, you need to tell solr to paginate further through the results. You do this by applying the `paginate()` method, which takes two parameters, `start` and `rows`:

```
>>> si.query("black").paginate(start=10, rows=30)
```

2.4 Cursors

If you want to get all / a huge number of results, you should use cursors to get the results in smaller chunks. Due to the way this is implemented in Solr, your sort needs to include your uniqueKey field. The `cursor()` method returns a cursor that you can iterate over. Like `execute()`, `cursor()` takes an optional `constructor` parameter. In addition you can pass `rows` to define how many results should be fetched from Solr at once.

```
>>> for item in si.query("black").sort_by('id').cursor(rows=100): ...
```

2.5 Returning different fields

By default, Solr will return all stored fields in the results. You might only be interested in a subset of those fields. To restrict the fields Solr returns, you apply the `field_limit()` methods.

```
>>> si.query("game").field_limit("id")
>>> si.query("game").field_limit(["id", "name"])
```

You can use the same option to get hold of the relevancy score that Solr has calculated for each document in the query:

```
>>> si.query("game").field_limit(score=True) # Return the score alongside each document
>>> si.query("game").field_limit("id", score=True") # return just the id and score.
```

The results appear just like the normal dictionary responses, but with a different selection of fields.

```
>>> for result in si.query("thief").field_limit("id", score=True"):
...     print result
{u'score': 0.6349302, u'id': u'978-0641723445'}
```

2.6 More complex queries

In our books example, there are two numerical fields - the `price` (which is a float) and `sequence_i` (which is an integer). Numerical fields can be queried:

- exactly
- by comparison (< / <= / >= / >)
- by range (between two values)

2.6.1 Exact queries

Don't try and query floats exactly unless you really know what you're doing (http://download.oracle.com/docs/cd/E19957-01/806-3568/ncg_goldberg.html). Solr will let you, but you almost certainly don't want to. Querying integers exactly is fine though.

```
>>> si.query(sequence_i=1)
```

2.6.2 Comparison queries

These use a new syntax:

```
>>> si.query(price__lt=7)
```

Notice the double-underscore separating "price" from "lt". It will search for all books whose price is less than 7. You can do similar searches on any float or integer field, and you can use:

- `gt` : greater than, >
- `gte` : greater than or equal to, >=
- `lt` : less than, <
- `lte` : less than or equal to, <=

2.6.3 Range queries

As an extension of a comparison query, you can query for values that are within a range, ie between two different numbers.

```
>>> si.query(price__range=(5, 7)) # all books with prices between 5 and 7.
```

This range query is *inclusive* - it will return prices of books which are priced at exactly 5 or exactly 7. You can also make an *exclusive* search:

```
>>> si.query(price__rangeexc=(5, 7))
```

Which will exclude books priced at exactly 5 or 7.

Finally, you can also do a completely open range search:

```
>>> si.query(price__any=True)
```

Will search for a book which has *any* price. Why would you do this? Well, if you had a schema where price was *optional*, then this search would return all books which had a price - and exclude any books which didn't have a price.

2.6.4 Date queries

You can query on dates the same way as you can query on numbers: exactly, by comparison, or by range.

Be warned, though, that exact searching on date suffers from similar problems to exact searching on floating point numbers. Solr stores all dates to microsecond precision; exact searching will fail unless the date requested is also correct to microsecond precision.

```
>>> si.query(date_dt=datetime.datetime(2006, 02, 13))
```

Will search for items whose manufacture date is *exactly* zero microseconds after midnight on the 13th February, 2006.

More likely you'll want to search by comparison or by range:

```
# all items after the 1st January 2006
>>> si.query(date_dt__gt=datetime.datetime(2006, 1, 1))

# all items in Q1 2006.
>>> si.query(date_dt__range=(datetime.datetime(2006, 1, 1), datetime.datetime(2006, 4, 1))
```

The argument to a date query can be any object that looks roughly like a Python `datetime` object or a string in W3C Datetime notation (http://www.w3.org/TR/NOTE-datetime)

```
>>> si.query(date_dt__gte="2006")
>>> si.query(date_dt__lt="2009-04-13")
>>> si.query(date_dt__range=("2010-03-04 00:34:21", "2011-02-17 09:21:44"))
```

2.6.5 Boolean fields

Boolean fields are flags on a document. In the example hardware specs, documents carry an `inStock` field. We can select on that by doing:

```
>>> si.query("thief", inStock=True)
```

2.7 Sorting results

Solr will return results in "relevancy" order. How Solr determines relevancy is a complex question, and can depend highly on your specific setup. However, it's possible to override this and sort query results by another field. This field must be sortable, so most likely your"d use a numerical or date field.

```
>>> si.query("thief").sort_by("price") # ascending price
>>> si.query("thief").sort_by("-price") # descending price
```

You can also sort on multiple factors:

```
>>> si.query("thief").sort_by("-price").sort_by("score")
```

This query will sort first by descending price, and then by increasing "score" (which is what Solr calls relevancy).

2.8 Complex queries

Scorched queries can be chained together in all sorts of ways, with query terms being applied.

What we do is construct two *query objects*, one for each condition, and `OR` them together.

```
>>> si.query(si.Q("thief") | si.Q("sea"))
```

The `Q` object can contain an arbitrary query, and can then be combined using Boolean logic (here, using `|`, the OR operator). The result can then be passed to a normal `si.query()` call for execution.

`Q` objects can be combined using any of the Boolean operators, so also `&` (`AND`) and `~` (`NOT`), and can be nested within each other.

A moderately complex query could be written:

```
>>> query = si.query(si.Q(si.Q("thief") & ~si.Q(author="ostein")) \
| si.Q(si.Q("foo") & ~si.Q(author="bui")))
```

Which will producse this query:

```
>>> query.options()
{'q': u'(thief AND (*:* AND NOT author:ostein)) OR (foo AND (*:* AND NOT author:bui))'}
```

2.9 Excluding results from queries

If we want to *exclude* results by some criteria we use the `~si.Q()`.

```
>>> si.query(~si.Q(author="Rick Riordan"))
```

2.10 Wildcard searching

You can use asterisks and question marks in the normal way, except that you may not use leading wildcards - ie no wildcards at the beginning of a term.

Search for book with "thie" in the name:

```
>>> si.query(name=scorched.strings.WildcardString("thie*"))
```

If, for some reason, you want to search exactly for a string with an asterisk or a question mark in it then you need to tell Solr to special case it:

```
>>> si.query(id=RawString("055323933?*"))
```

This will search for a document whose id contains *exactly* the string given, including the question mark and asterisk.

2.11 Filter queries

Solr implements several internal caching layers, and to some extent you can control when and how they're used.

Often, you find that you can partition your query; one part is run many times without change, or with very limited change, and another part varies much more. (See http://wiki.apache.org/solr/FilterQueryGuidance for more guidance.)

If you taking search input from the user, you would write:

```
>>> si.query(name=user_input).filter(price__lt=7.5)
>>> si.query(name=user_input).filter(price__gte=7.5)
```

Adding multiple filter:

```
>>> si.query(name="bla").filter(price__lt=7.5).filter(author="hans").options()
{'fq': [u'author:hans', u'price:{* TO 7.5}'], 'q': u'name:bla'}
```

You can filter any sort of query, simply by using `filter()` instead of `query()`. And if your filtering involves an exclusion, then simple use `~si.Q(author="lloyd")`.

```
>>> si.query(title="black").filter(~si.Q(author="lloyd")).options()
{'fq': u'NOT author:lloyd', 'q': u'title:black'}
```

It's possible to mix and match `query()` and `filter()` calls as much as you like while chaining. The resulting filter queries will be combined and cached together. The argument to a `filter()` call can be an combination of `si.Q` objects.

```
>>> si.query(title="black").filter(
...     si.Q(si.Q(name="thief") & ~si.Q(author="ostein"))
...         ).filter(si.Q(si.Q(title="foo") & ~si.Q(author="bui"))
... ).options()
{'fq': [u'name:thief', u'title:foo', u'NOT author:ostein', u'NOT author:bui'],
 'q': u'title:black'}
```

2.12 Boosting

Solr provides a mechanism for "boosting" results according to the values of various fields (See http://wiki.apache.org/solr/SolrRelevancyCookbook#Boosting_Ranking_Terms for a full explanation).

Boosts the importance of the author field by 3.

```
>>> si.query(si.Q("black") | si.Q(author="lloyd")**3).options()
{'q': u'black OR author:lloyd^3'}
```

A more common pattern is that you want all books with "black" in the title *and you have a preference for those authored by Lloyd Alexander*. This is different from the last query; the last query would return books by Lloyd

Alexander which did not have "black" in the title. Achieving this in Solr is possible, but a little awkward; scorched provides a shortcut for this pattern.

```
>>> si.query("black").boost_relevancy(3, author_t="lloyd").options()
{'q': u'black OR (black AND author_t:lloyd^3)'}
```

This is fully chainable, and `boost_relevancy` can take an arbitrary collection of query objects.

2.13 Faceting

For background, see http://wiki.apache.org/solr/SimpleFacetParameters.

Scorched lets you apply faceting to any query, with the `facet_by()` method, chainable on a query object. The `facet_by()` method needs, at least, a field (or list of fields) to facet on:

```
>>> facet_query = si.query("thief").facet_by("sequence_i").paginate(rows=0)
```

The above fragment will search for game with "thrones" in the title, and facet the results according to the value of `sequence_i`. It will also return zero results, just the facet output.

```
>>> print facet_query.execute().facet_counts.facet_fields
{u'sequence_i': [(u'1', 1), (u'2', 0)]}
```

The `facet_counts` objects contains several sets of results - here, we're only interested in the `facet_fields` object. This contains a dictionary of results, keyed by each field where faceting was requested. The dictionary value is a list of two-tuples, mapping the value of the faceted field.

You can facet on more than one field at a time:

```
>>> si.query(...).facet_by(fields=["field1", "field2, ...])
```

The `facet_fields` dictionary will have more than one key.

Solr supports a number of parameters to the faceting operation. All of the basic options are exposed through scorched:

```
fields, prefix, sort, limit, offset, mincount, missing, method,
enum.cache.minDf
```

All of these can be used as keyword arguments to the `facet()` call, except of course the last one since it contains periods. To pass keyword arguments with periods in them, you can use ** syntax:

You can facet by ranges. The following query will return range facets over `field1`: 0-10, 11-20, 21-30, etc. The `mincount` parameter can be used to return only those facets which contain a minimum number of results.

```
>>> si.query(...).facet_range(fields='field1', start=0, gap=10, end=100, \
                              limit=10, mincount=1)
```

Alternatively, you create ranges of dates using Solr's *date math* syntax. This next example creates a facet for each of the last 12 months.

```
>>> si.query(...).facet_range(fields='field1', start='NOW-12MONTHS/MONTH', \
                              gap='+1MONTHS', end='NOW/MONTH')
```

See https://cwiki.apache.org/confluence/display/solr/Working+with+Dates#WorkingwithDates-DateMath for more details on *date math* syntax.

```
>>> facet(**{"enum.cache.minDf":25})
```

You can also facet on the result of one or more queries, using the `facet_query()` method. For example:

```
>>> fquery = si.query("game").facet_query(price__lt=7).facet_query(price__gte=7)
>>> print fquery.execute().facet_counts.facet_queries
[('price:[7.0 TO *]', 1), ('price:{* TO 7.0}', 1)]
```

This will facet the results according to the two queries specified, so you can see how many of the results cost less than 7, and how many cost more.

The results come back this time in the `facet_queries` object, but have the same form as before. The facets are shown as a list of tuples, mapping query to number of results.

Facet pivot TODO https://wiki.apache.org/solr/HierarchicalFaceting#Pivot_Facets

2.14 Result grouping

For background, see http://wiki.apache.org/solr/FieldCollapsing.

Solr 3.3 added support for result grouping.

An example call looks like this:

```
>>> resp = si.query().group_by('genre_s', limit=10).execute()
>>> for g in resp.groups['genre_s']['groups']:
...     print "%s #%s" % (g['groupValue'], len(g['doclist']['docs']))
...     for d in  g['doclist']['docs']:
...         print "\t%s" % d['name']
fantasy #3
    The Lightning Thief
    The Sea of Monsters
    Sophie's World : The Greek Philosophers
IT #1
    Lucene in Action, Second Edition
```

2.15 Highlighting

For background, see http://wiki.apache.org/solr/HighlightingParameters.

Alongside the normal search results, you can ask Solr to return fragments of the documents, with relevant search terms highlighted. You do this with the chainable `highlight()` method.

Specify which field we would like to see highlighted:

```
>>> resp = si.query('thief').highlight('name').execute()
>>> resp.highlighting
{u'978-0641723445': {u'name': [u'The Lightning <em>Thief</em>']}}
```

It is also possible to specify a array of fields:

```
>>> si.query('thief').highlight(['name', 'title']).options()
{'hl': True, 'hl.fl': 'name,title', 'q': u'thief'}
```

Highlighting values will also be included in ‘‘response.result.doc‘ and grouped results as a ‘‘solr_highlights‘ attribute so that they can be accessed during result iteration.

2.16 PostingsHighlighter

For background, see https://wiki.apache.org/solr/PostingsHighlighter.

PostingsHighlighter is a new highlighter in Solr4.3 to summarize documents for summary results. You do this with the chainable `postings_highlight()` method.

Specify which field we would like to see highlighted:

```
>>> resp = si.query('thief').postings_highlight('name').execute()
>>> resp.highlighting
{u'978-0641723445': {u'name': [u'The Lightning <em>Thief</em>']}}
```

It is also possible to specify a array of fields:

```
>>> si.query('thief').postings_highlight(['name', 'title']).options()
{'hl': True, 'hl.fl': 'name,title', 'q': u'thief'}
```

2.17 Term Vectors

For background, see https://wiki.apache.org/solr/TermVectorComponent.

Alongside the normal search results, you can ask solr to return the term vector, the term frequency, inverse document frequency, and position and offset information for the documents. You do this with the chainable `term_vector()` method.

```
>>> resp = si.query('thief').term_vector(all=True).execute()
```

You can also specify for which fields you would like to get information:

```
>>> resp = si.query('thief').term_vector('name').execute()
```

It is also possible to specify a array of fields:

```
>>> si.query('thief').term_vector(['name', 'title'], all=True).execute()
```

2.18 More Like This

For background, see http://wiki.apache.org/solr/MoreLikeThis. Alongside a set of search results, Solr can suggest other documents that are similar to each of the documents in the search result.

More-like-this searches are accomplished with the `mlt()` chainable option. Solr needs to know which fields to consider when deciding similarity.

```
>>> resp = si.query(id="978-0641723445").mlt("genre_s", mintf=1, mindf=1).execute()
>>> resp.more_like_these
{u'978-0641723445': <scorched.response.SolrResult at 0x28b6350>}

>>> resp.more_like_these['978-0641723445'].docs
[{u'_version_': 1462820023772905472,
  u'author': u'Rick Riordan',
  u'author_s': u'Rick Riordan',
  u'cat': [u'book', u'paperback'],
  u'genre_s': u'fantasy',
  u'id': u'978-1423103349',
```

```
  u'inStock': True,
  u'name': u'The Sea of Monsters',
  u'pages_i': 304,
  u'price': 6.49,
  u'price_c': u'6.49,USD',
  u'sequence_i': 2,
  u'series_t': u'Percy Jackson and the Olympians'},
 {u'_version_': 1462820023776051200,
  u'author': u'Jostein Gaarder',
  u'author_s': u'Jostein Gaarder',
  u'cat': [u'book', u'paperback'],
  u'genre_s': u'fantasy',
  u'id': u'978-1857995879',
  u'inStock': True,
  u'name': u"Sophie's World : The Greek Philosophers",
  u'pages_i': 64,
  u'price': 3.07,
  u'price_c': u'3.07,USD',
  u'sequence_i': 1}]
```

Here we used `mlt()` options to alter the default behaviour (because our corpus is so small that Solr wouldn't find any similar documents with the standard behaviour.

The `SolrResponse` object has a `more_like_these` attribute. This is a dictionary of `SolrResult` objects, one dictionary entry for each result of the main query. Here, the query only produced one result (because we searched on the `uniqueKey`. Inspecting the `SolrResult` object, we find that it contains only one document.

We can read the above result as saying that under the `mlt()` parameters requested, there was only one document similar to the search result.

To avoid having to do the extra dictionary lookup.

`mlt()` also takes a list of options (see the Solr documentation for a full explanation);

```
fields, count, mintf, mindf, minwl, mawl, maxqt, maxntp, boost
```

2.19 Alternative parser

Scorched supports the *dismax* and *edismax* parser. These can be added by simply calling `alt_parser`.

Example:

```
>>> si.query().alt_parser('edismax', mm=2).options()
{'defType': 'edismax', 'mm': 2, 'q': '*:*'}
```

The *edismax* parser also supports field aliases. Here is an example where `foo` is aliased to the fields `bar` and `baz`.

Example:

```
>>> si.query().alt_parser('edismax', f={'foo':['bar', 'baz']}).options()
{'defType': 'edismax', 'q': '*:*', 'f.foo.qf': 'bar baz'}
```

2.20 Set request handler

For background, see https://wiki.apache.org/solr/SolrRequestHandler. It is possible to set the request handler. To set a different request handler use `set_requesthandler`.

Example:

```
>>> si.query().set_requesthandler('foo').options()
{u'q': u'*:*', u'qt': 'foo'}
```

2.21 Set debug

For background, see https://wiki.apache.org/solr/CommonQueryParameters#Debugging. To see what Solr is doing with our query we need sometimes more info. To get this additional information we set `debug`.

Example:

```
>>> si.query().debug().options()
{u'debugQuery': True, u'q': u'*:*'}
>>>  si.query().debug().execute().debug
{u'QParser': u'LuceneQParser',
u'explain': {u'978-1423103349': u'\n1.0 = (MATCH) MatchAllDocsQuery, product of:\n  1.0 = queryNorm\n',
 u'978-1857995879': u'\n1.0 = (MATCH) MatchAllDocsQuery, product of:\n  1.0 = queryNorm\n',
 u'978-1933988177': u'\n1.0 = (MATCH) MatchAllDocsQuery, product of:\n  1.0 = queryNorm\n'},
u'parsedquery': u'MatchAllDocsQuery(*:*)',
u'parsedquery_toString': u'*:*',
u'querystring': u'*:*',
u'rawquerystring': u'*:*',
u'timing': {u'prepare': {u'debug': {u'time': 0.0},
  u'facet': {u'time': 0.0},
  u'highlight': {u'time': 0.0},
  u'mlt': {u'time': 0.0},
  u'query': {u'time': 0.0},
  u'stats': {u'time': 0.0},
  u'time': 0.0},
 u'process': {u'debug': {u'time': 0.0},
  u'facet': {u'time': 0.0},
  u'highlight': {u'time': 0.0},
  u'mlt': {u'time': 0.0},
  u'query': {u'time': 1.0},
  u'stats': {u'time': 0.0},
  u'time': 1.0},
 u'time': 1.0}}
```

2.22 Enable spellchecking

For background, see http://wiki.apache.org/solr/SpellCheckComponent. It is possible to activate spellchecking in yout query. To do that, use `spellcheck`.

Example:

```
>>> si.query().spellcheck().options()
{u'q': u'*:*', u'spellcheck': 'true'}
```

2.23 Realtime Get

For background, see https://wiki.apache.org/solr/RealTimeGet

Solr 4.0 added support for retrieval of documents that are not yet commited. The retrieval can only by done by id:

```
>>> resp = si.get("978-1423103349")
```

You can also pass multiple ids:

```
>>> resp = si.get(["978-0641723445", "978-1423103349"])
```

The return value is the same as for a normal search

2.24 Stats

For background, see https://wiki.apache.org/solr/StatsComponent

Solr can return simple statistics for indexed numeric fields:

```
>>> resp = solr.query().stats('int_field')
```

You can also pass multiple fields:

```
>>> resp = solr.query().stats(['int_field', 'float_field'])
```

The resulting statistics are available on the response at `resp.stats.stats_fields`.

CHAPTER 3

More Like This queries

More Like This (MLT) is a feature of Solr which provides for comparisons of documents; you can ask Solr to tell you about any More documents it has that are Like This one.

An MLT query can be part of a standard query (see standard-query-more-like-this.), in which case you're asking Solr to tell you not only about immediate query results, but also about any other results which are similar to the results you've got.

Alternatively, you can feed Solr an entire document that is not already in its index, and ask to do an MLT query on that document.

The first case is covered above in standard-query-more-like-this; the second case we'll show here.

3.1 Basic MLT query

Instead of calling the `query` method on the interface, we call the `mlt_query` method.

```
>>> si.mlt_query(fields="name", content=open("localfile").read())
```

We give the MLT handler some content (sourced in this case from a local file); the MLT query will take this text, analyze it, and retrieve documents that are similar according to the results of its analysis.

The results are returned in the same format as illustrated in the `mlt()` method.

3.2 Further MLT query options

If we wanted similarity to be calculated with respect to a different field or fields.:

```
>>> si.mlt_query(content=open("localfile").read(),
...              fields=["name", "author_t"])
```

We can understand a little more about why we get the results we do by asking for the result of the MLT document analysis.

```
>>> si.mlt_query(fields="name", content=open("localfile").read(),
...              interestingTerms="list")
>>> si.mlt_query(fields="name", content=open("localfile").read(),
...              interestingTerms="details")
```

"list" will return a list of the interesting terms extracted; "details" will also provide details of the boost used for each term.

If the document you're supplying is not encoded in UTF-8 (or equivalently ASCII) format, then you need to specify the charset in use (using the list available at http://docs.python.org/library/codecs.html#standard-encodings:

```
>>> si.mlt_query(fields="name", content=open("localfile").read(),
...              content_charset="iso-8859-1")
```

3.3 Sourcing content from the web

You can also choose to tell Solr to source the document from the web, by giving the URL for the content rather than supplying it yourself:

```
>>> si.mlt_query(fields="name", url="http://example.com/document")
```

All the other options above still apply to URL-sourced content, except for "content_charset"; that's up to the webserver where the content is stored.

In all the cases above, you can also specify any of the other options shown in `mlt()`, apart from "count".

3.4 MLT queries on indexed content

You can perform an MLT query on indexed content in the following way:

```
>>> res = si.mlt_query("genre_s", interestingTerms="details",
...                    mintf=1, mindf=1).query(
...                    id="978-0641723445").execute()
>>> res.result.docs
[{u'_version_': 1462917302263480320,
  u'author': u'Rick Riordan',
  u'author_s': u'Rick Riordan',
  u'cat': [u'book', u'paperback'],
  u'genre_s': u'fantasy',
  u'id': u'978-1423103349',
  u'inStock': True,
  u'name': u'The Sea of Monsters',
  u'pages_i': 304,
  u'price': 6.49,
  u'price_c': u'6.49,USD',
  u'sequence_i': 2,
  u'series_t': u'Percy Jackson and the Olympians'},
 {u'_version_': 1462917302263480321,
  u'author': u'Jostein Gaarder',
  u'author_s': u'Jostein Gaarder',
  u'cat': [u'book', u'paperback'],
  u'genre_s': u'fantasy',
  u'id': u'978-1857995879',
  u'inStock': True,
  u'name': u"Sophie's World : The Greek Philosophers",
  u'pages_i': 64,
  u'price': 3.07,
  u'price_c': u'3.07,USD',
  u'sequence_i': 1}]
>>> res.interesting_terms
>>> [u'genre_s:fantasy', 1.0]
```

ie - initialize an otherwise empty mlt_query object, and then run queries on it as you would run normal queries. The full range of query operations is supported when composing the query for indexed content:

```
>>> si.mlt_query("name").query(title='Whale').query(~si.Q(
...     author='Melville').query(si.Q('Moby') | si.Q('Dick'))
```

3.5 Chaining MLT queries

The `mlt_query()` method is chainable in the same way as the `query` method. There are a fre differences to note.

- You can't chain a `query()` onto an `mlt_query()` call if the MLT query is based on supplied `content` or `url`.
- You can't chain multiple `mlt_query()` methods together - only one content source can be considered at a time.

The `mlt_query()` method takes all of the mlt() options except "count".

CHAPTER 4

Scorched API

4.1 API

scorched.connection.**grouper**(*iterable*, *n*)
grouper('ABCDEFG', 3) -> [['ABC'], ['DEF'], ['G']]

class scorched.connection.**SolrConnection**(*url*, *http_connection*, *mode*, *retry_timeout*, *max_length_get_url*, *search_timeout=()*)

__init__(*url*, *http_connection*, *mode*, *retry_timeout*, *max_length_get_url*, *search_timeout=()*)

Parameters

- **url** (*str*) – url to Solr
- **http_connection** (*requests connection*) – already existing connection TODO
- **mode** (*str*) – mode (readable, writable) Solr
- **retry_timeout** (*int*) – timeout until retry
- **max_length_get_url** (*int*) – max length until switch to post
- **search_timeout** (*float or tuple*) – (optional) How long to wait for the server to send data before giving up, as a float, or a (connect timeout, read timeout) tuple.

get(*ids*, *fl=None*)
Perform a RealTime Get

mlt(*params*, *content=None*)

Parameters **params** (*dict*) – LuceneQuery converted to a dictionary with search queries

Returns json – json string

Perform a MoreLikeThis query using the content specified There may be no content if stream.url is specified in the params.

request(**args*, ***kwargs*)

Parameters

- **args** (*tuple*) – arguments
- **kwargs** (*dict*) – key word arguments

select(*params*)

Parameters **params** (*dict*) – LuceneQuery converted to a dictionary with search queries

Returns json – json string

We perform here a search on the *select* handler of Solr.

update(*update_doc*, ***kwargs*)

Parameters **update_doc** (*json data*) – data send to Solr

Returns json – json string

Send json to Solr

url_for_update(*commit=None*, *commitWithin=None*, *softCommit=None*, *optimize=None*, *waitSearcher=None*, *expungeDeletes=None*, *maxSegments=None*)

Parameters

- **commit** (*bool*) – optional – commit actions
- **commitWithin** (*int*) – optional – document will be added within that time
- **softCommit** (*bool*) – optional – performant commit without "on-disk" guarantee
- **optimize** – optional – optimize forces all of the index segments to be merged into a single segment first.
- **waitSearcher** (*bool*) – optional – block until a new searcher is opened and registered as the main query searcher,
- **expungeDeletes** (*bool*) – optional – merge segments with deletes away
- **maxSegments** (*int*) – optional – optimizes down to at most this number of segments

Returns str – url with all extra paramters set

This functions sets all extra parameters for the `optimize` and `commit` function.

class scorched.connection.**SolrInterface**(*url*, *http_connection=None*, *mode=u''*, *retry_timeout=-1*, *max_length_get_url=2048*, *search_timeout=()*)

__init__(*url*, *http_connection=None*, *mode=u''*, *retry_timeout=-1*, *max_length_get_url=2048*, *search_timeout=()*)

Parameters

- **url** (*str*) – url to Solr
- **http_connection** (*requests connection*) – optional – already existing connection TODO
- **mode** (*str*) – optional – mode (readable, writable) Solr
- **retry_timeout** (*int*) – optional – timeout until retry
- **max_length_get_url** (*int*) – optional – max length until switch to post
- **search_timeout** (*float or tuple*) – (optional) How long to wait for the server to send data before giving up, as a float, or a (connect timeout, read timeout) tuple.

add(*docs*, *chunk=100*, ***kwargs*)

Parameters

- **docs** (*dict*) – documents to be added
- **chunk** – optional – size of chunks in which the add command

should be split :type chunk: int :param kwargs: optinal – additional arguments :type kwargs: dict :returns: list of SolrUpdateResponse – A Solr response object.

Add a document or a list of document to Solr.

commit(*waitSearcher=None*, *expungeDeletes=None*, *softCommit=None*)

Parameters

- **waitSearcher** (*bool*) – optional – block until a new searcher is opened and registered as the main query searcher, making the changes visible
- **expungeDeletes** (*bool*) – optional – merge segments with deletes away
- **softCommit** (*bool*) – optional – perform a soft commit - this will refresh the 'view' of the index in a more performant manner, but without "on-disk" guarantees.

Returns SolrUpdateResponse – A Solr response object.

A commit operation makes index changes visible to new search requests.

delete_all()

Returns SolrUpdateResponse – A Solr response object.

Delete everything

delete_by_ids(*ids*, ***kwargs*)

Parameters **ids** (*list*) – ids of entries that should be deleted

Returns SolrUpdateResponse – A Solr response object.

Delete entries by a given id

delete_by_query(*query*, ***kwargs*)

Parameters **query** (*LuceneQuery*) – criteria how witch entries should be deleted

Returns SolrUpdateResponse – A Solr response object.

Delete entries by a given query

get(*ids*, *fields=None*)

RealTime Get document(s) by id(s)

Parameters

- **ids** (*list, string or int*) – id(s) of the document(s)
- **fields** – optional – list of fields to return

mlt_query(*fields*, *content=None*, *content_charset=None*, *url=None*, *query_fields=None*, ***kwargs*)

Parameters

- **fields** (*list*) – field names to compute similarity upon
- **content** (*str*) – optional – string on witch to find similar documents
- **content_charset** (*str*) – optional – charset e.g. (iso-8859-1)
- **url** (*str*) – optional – like content but retrive directly from url
- **query_fields** (*dict e.g. {"a": 0.25, "b": 0.75}*) – optional – adjust boosting values for `fields`

Returns MltSolrSearch

Perform a similarity query on MoreLikeThisHandler

The MoreLikeThisHandler is expected to be registered at the '/mlt' endpoint in the solrconfig.xml file of the server.

Other MoreLikeThis specific parameters can be passed as kwargs without the 'mlt.' prefix.

mlt_search(*content=None*, ***kwargs*)

Returns SolrResponse – A Solr response object.

More like this search Solr

optimize(*waitSearcher=None*, *maxSegments=None*)

Parameters

- **waitSearcher** (*bool*) – optional – block until a new searcher is opened and registered as the main query searcher, making the changes visible
- **maxSegments** (*int*) – optional – optimizes down to at most this number of segments

Returns SolrUpdateResponse – A Solr response object.

An optimize is like a hard commit except that it forces all of the index segments to be merged into a single segment first.

query(**args*, ***kwargs*)

Returns SolrSearch – A solrsearch.

Build a Solr query

rollback()

Returns SolrUpdateResponse – A Solr response object.

The rollback command rollbacks all add/deletes made to the index since the last commit

search(***kwargs*)

Returns SolrResponse – A Solr response object.

Search solr

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

S

Symbols

A

C

D

G

M

O

Q

R

S

U