redis-completion Documentation

Release 0.4.0

charles leifer

June 16, 2016

Contents

Warning: This project is now deprecated. The code has a new home as a part of walrus, a set of Python utilities for working with Redis.

autocompletion with redis based on:

- http://antirez.com/post/autocomplete-with-redis.html
- http://stackoverflow.com/questions/1958005/redis-autocomplete/1966188

redis-completion is capable of storing a large number of phrases and quickly searching them for matches. Rich data can be stored and retrieved, helping you avoid trips to the database when retrieving search results.

CHAPTER 1

usage

If you just want to store really simple things, like strings:

```
engine = RedisEngine()
titles = ['python programming', 'programming c', 'unit testing python',
          'testing software', 'software design']
map(engine.store, titles)

>>> engine.search('pyt')
['python programming', 'unit testing python']

>>> engine.search('test')
['testing software', 'unit testing python']
```

If you want to store more complex data, like blog entries:

```
Entry.create(title='an entry about python', published=True)
Entry.create(title='all about redis', published=True)
Entry.create(title='using redis with python', published=False)

for entry in Entry.select():
    engine.store_json(entry.id, entry.title, {
        'published': entry.published,
        'title': entry.title,
        'url': entry.get_absolute_url(),
    })

>>> engine.search_json('pytho')
[{'published': True, 'title': 'an entry about python', 'url': '/blog/1/'},
 {'published': False, 'title': 'using redis with python', 'url': '/blog/3/'}]

# just published entries, please
>>> engine.search_json('redis', filters=[lambda i: i['published'] == True])
[{u'published': True, u'title': u'all about redis', u'url': u'/blog/2/'}]
```

Contents:

1.1 Installing

Install with pip:

```
pip install redis-completion
```

Install via git:

```
git clone https://github.com/coleifer/redis-completion.git
cd redis-completion
python setup.py install
```

1.1.1 Dependencies

redis-completion depends on the following libraries:

- redis-py

Also you will need to set up a Redis server if you do not have one running somewhere already. Information on that can be found on the project's homepage.

1.2 Getting Started

redis-completion was designed to make type-ahead search easy to integrate with your existing app. Assuming you have followed the *installation notes*, lets get started building a simple stock symbol lookup. I've gone ahead and uploaded a list of symbols and company names, which can be found here:

http://media.charlesleifer.com/downloads/misc/NYSE.txt

Let's get started by writing a script to populate our index by pulling the file down, reading its contents, then storing a mapping of company name -> symbol so we can easily search for companies we're interested in.

Note: This code can be found in the examples

```
import urllib2
from redis_completion import RedisEngine

engine = RedisEngine(prefix='stocks')

def load_data():
    url = 'http://media.charlesleifer.com/downloads/misc/NYSE.txt'
    contents = urllib2.urlopen(url).read()
    for row in contents.splitlines()[1:]:
        ticker, company = row.split('\t')
        # id, search phrase, data
        engine.store_json(ticker, company, {'ticker': ticker, 'company': company})

def search(p, **kwargs):
    return engine.search_json(p, **kwargs)
```

Save this script to a file and open up an interactive shell:

```
>>> from stocks import *
>>> load_data() # this may take a few seconds
```

Excellent, we've loaded all the data and can now perform searches on it:

```
>>> search('uni sta')
[{u'company': u'Strats Sm Trust For United States Cellular Corp',
  u'ticker': u'GJH'},
 {u'company': u'United States Cellular Corp.', u'ticker': u'USM'},
 {u'company': u'United States Cellular Corp.', u'ticker': u'UZA'},
 {u'company': u'United States Steel Corp.', u'ticker': u'X'}]

>>> search('shi co')
[{u'company': u'International Shipholding Corp.', u'ticker': u'ISH'},
 {u'company': u'Shinhan Financial Group Co Ltd', u'ticker': u'SHG'},
 {u'company': u'Teekay Shipping Corp.', u'ticker': u'TK'}]

>>> search('prog')
[{u'company': u'Progress Energy Inc.', u'ticker': u'PGN'},
 {u'company': u'Progressive Corp.', u'ticker': u'PGR'}]
```

1.3 Schema

1.3.1 Term definitions

The schema is composed of 3 pieces of data, which are the parameters for the *store()* method:

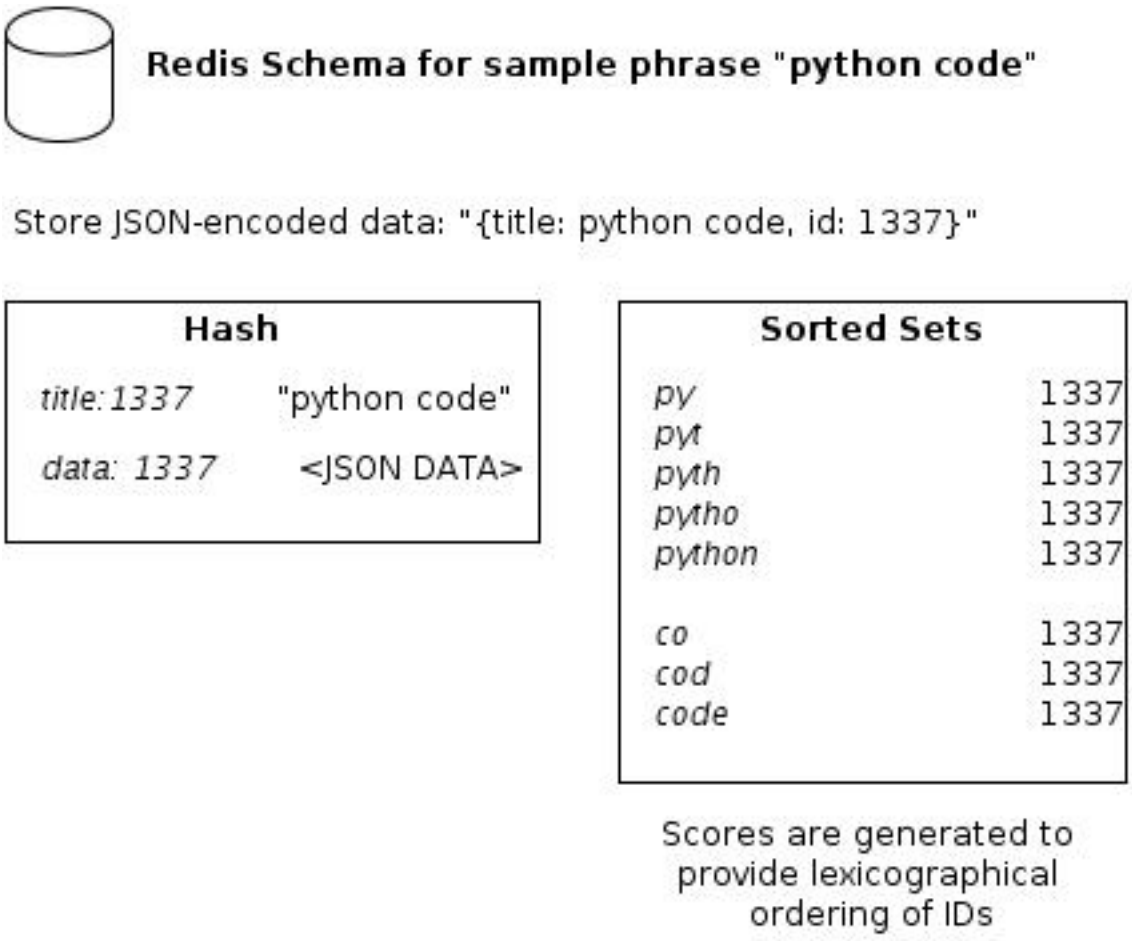
object_id A unique identifier for the given data, such as a primary key

title A title that will be used to generate a searchable index

data Any data you wish to associate with an object_id. This data will be returned when searches are performed.

1.3.2 Redis schema

redis-completion uses multiple sorted sets to perform searches for partial phrases. When indexing a term, it is split up into pieces of increasing length, so *"test"* becomes `[te, tes, test]`. Each of these partial phrases becomes the key to a sorted set. Each sorted set contains the IDs of data that matched that particular phrase. After finding the IDs, the associated data is pulled from a hash keyed by the object's id.



1.4 API

class **RedisEngine** (*prefix='ac'*, *stop_words=None*, *cache_timeout=300*, ***conn_kwargs*)

Parameters

- **prefix** (*string*) – a prefix used for all keys stored in Redis to allow multiple "indexes" to exist and to make deletion easier.
- **stop_words** (*set*) – a `set` of stop words to remove from index/search data
- **cache_timeout** (*integer*) – how long to keep around search results
- **conn_kwargs** – any named parameters that should be used when connecting to Redis, e.g. `host='localhost', port=6379`

RedisEngine is responsible for storing and searching data suitable for autocompletion. There are many different options you can use to configure how autocomplete behaves, but the defaults are intended to provide good general performance for searching things like blog post titles and the like.

The underlying data structure used to provide autocompletion is a sorted set, details are described in this post.

Usage:

```
from redis_completion import RedisEngine
engine = RedisEngine()
```

store(*obj_id*[, *title=None*[, *data=None*[, *obj_type=None*]]])

Parameters

- **obj_id** – a unique identifier for the object
- **title** – a string to store in the index and allow autocompletion on, which, if not provided defaults to the given `obj_id`
- **data** – any data you wish to store and return when a given title is searched for. If not provided, defaults to the given `title` (or `obj_id`)
- **obj_type** – an additional identifier for this object if multiple types are stored

Store an object in the index and allow it to be searched for.

Examples:

```
engine.store('some phrase')
engine.store('some other phrase')
```

In the following example a list of blog entries is being stored in the index. Note that arbitrary data can be stored in the index. When a search is performed this data will be returned.

```
for entry in Entry.select():
    engine.store(entry.id, entry.title, json.dumps({
        'id': entry.id,
        'published': entry.published,
        'title': entry.title,
        'url': entry.url,
    })
```

store_json(*obj_id*, *title*, *data*[, *obj_type=None*])

Like *store()* except `data` is automatically serialized as JSON before being stored in the index. Best when used in conjunction with *search_json()*.

exists(*obj_id*[, *obj_type=None*])

Parameters

- **obj_id** – a unique identifier for the object
- **obj_type** – an additional identifier for this object if multiple types are stored

Checks if the given object exists in the index

remove(*obj_id*[, *obj_type=None*])

Parameters

- **obj_id** – a unique identifier for the object
- **obj_type** – an additional identifier for this object if multiple types are stored

Removes the given object from the index.

boost(*obj_id*[, *multiplier=1.1*[, *negative=False*]])

Parameters

- **obj_id** – a unique identifier for the object

- **multiplier** (*float*) – a float indicating how much to boost obj_id by, where greater than one will push to the front, less than will push to the back
- **negative** (*boolean*) – whether to push forward or backward (if negative, will simply inverse the multiplier)

Boost the score of the given `obj_id` by the multiplier, then store the result in a special hash. These stored "boosts" can later be recalled when searching by specifying `autoboost=True`.

search(*phrase*[, *limit=None*[, *filters=None*[, *mappers=None*[, *boosts=None*[, *autoboost=False*]]]]])

Parameters

- **phrase** – search the index for the given phrase
- **limit** – an integer indicating the number of results to limit the search to.
- **filters** – a list of callables which will be used to filter the search results before returning to the user. Filters should take a single parameter, the `data` associated with a given object and should return either `True` or `False`. A `False` value returned by any of the filters will prevent a result from being returned.
- **mappers** – a list of callables which will be used to transform the raw data returned from the index.
- **boosts** (*dict*) – a mapping of type identifier -> float value – if provided, results of a given id/type will have their scores multiplied by the corresponding float value, e.g. `{'id1': 1.1, 'id2': 1.3, 'id3': .9}`
- **autoboost** (*boolean*) – automatically prepopulate boosts by looking at the stored boost information created using the *boost()* api

Return type A list containing data returned by the index

Note: Mappers act upon data before it is passed to the filters

---

Assume we have stored some interesting blog posts, encoding some metadata using JSON:

```
>>> engine.search('python', mappers=[json.loads])
[{'published': True, 'title': 'an entry about python', 'url': '/blog/1/'},
 {'published': False, 'title': 'using redis with python', 'url': '/blog/3/'}]
```

search_json(*phrase*[, *limit=None*[, *filters=None*[, *mappers=None*[, *boosts=None*[, *autoboost=False*]]]]])

Like *search()* except `json.loads` is inserted as the very first mapper. Best when used in conjunction with *store_json()*.

flush([*everything=False*[, *batch_size=1000*]])

Clears all data from the database. By default will only delete keys that are associated with the given engine (based on its prefix), but if `everything` is `True`, then the entire database will be flushed. The latter is faster.

Parameters

- **everything** (*boolean*) – whether to delete the entire current redis db
- **batch_size** (*int*) – number of keys to delete at-a-time

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

B

E

F

R

S