RedisDB Documentation

Release 0.2.1

django-redisdb authors

2015-07-06

Contents

Django-redisdb is Redis [1] backend for Django [2] that allows using Redis as a cache and as a database at the same time. Django-redisdb provides backends for master/master and sharded configuration.

[1] Redis http://redis.io/
[2] Django https://djangoproject.com/

CHAPTER 1

Contents

1.1 Installation

In order to install django-redisdb, make sure Python is installed. django-redisdb was tested with:

- Python 2.7
- Python 3.3
- Python 3.4

1.1.1 Using pip

To install run:

```
pip install django-redisdb
```

To update an already installed verision:

```
pip install -U django-redisdb
```

You can use *easy_install* in place of *pip* as well:

```
easy_install install -U django-redisdb
```

Note: If you haven't pip read pip installation informations.

1.1.2 Using a Source Distribution

Download and extract source distribution from pypi and run:

```
python setup.py install
```

1.1.3 Using the Github Repository

To install the newest version from the Github repository run:

```
pip install git+https://github.com/kidosoft/django-redisdb
```

1.1.4 Using the Bitbucket Repository

To install the newest version from the Bitbucket repository run:

```
pip install hg+https://github.com/kidosoft/django-redisdb
```

1.2 Configuration

django-redisdb is configured using standard Django's cache framework:

```
CACHES = {
    'redis_ring': {
        'BACKEND': 'redisdb.backends.RedisRing',  # sharding backend
        'DB': 0,
        'LOCATION': [
            'localhost:6379',
            'localhost:6380',
        ],
        'OPTIONS': {
            'socket_timeout': 5,
            'socket_connect_timeout': 5,
        },
    },
    'redis_copy': {
        'BACKEND': 'redisdb.backends.RedisCopy',  # copying backend
        'DB': 0,
        'LOCATION': [
            'localhost:6379',
            'localhost:6380',
        ],
        'OPTIONS': {
            'socket_timeout': 5,
            'socket_connect_timeout': 5,
        },
    }
}
```

Required key's are:

- BACKEND - *redisdb.backends.RedisRing* or *redisdb.backends.RedisCopy*. It determines how data would be stored across nodes.
- DB - database identifier number
- LOCATION - list of node addresses

You can pass additional options for connection creation :

- password - password for authentication
- socket_timeout - timeout for blocking operations
- socket_connect_timeout - timeout on connection
- retry_on_timeout - if True, retry when timeout occured

1.3 Quick usage quide

If you configured connection *Configuration* you can use Redis like any other Django's cache:

```
>>> from django.core.cache import caches
>>> caches['redis_ring'].set('one_key', 123)  # set key1 only on on server
[True]
>>> caches['redis_copy'].set('other_key', 234)  # set key2 on all servers
[True, True]
```

With RedisRing value is set only on one node. With RedisCopy it's set on all nodes (two nodes in examle above).

Redis is much more powerfull then simple cache. It should be seen as a specialized database. With django-redisdb you can use all its power. For example you can use redis' sorted sets [1]:

```
>>> caches['redis_copy'].zadd('myzset', 1, 'one')
[0, 1]
>>> caches['redis_copy'].zadd('myzset', 2, 'two')
[0, 1]
>>> caches['redis_copy'].zadd('myzset', 3, 'three')
[0, 1]
>>> caches['redis_copy'].zrange('myzset', 0, -1)
['one', 'two', 'three']
>>> caches['redis_copy'].zrange('myzset', 0, -1, withscores=True)
[('one', 1.0), ('two', 2.0), ('three', 3.0)]
```

1.3.1 Return values

RedisCopy can save data to many nodes. Each of this nodes can return different result on save. For that reason commands that save data to nodes returns list of results from each node. E.g. with two nodes set for redis_copy:

```
>>> print caches['redis_copy'].set('key1', 2)
[True, True]
```

1.4 Backends

class `redisdb.backends.`**`RedisRing`**(*location*, *params*)

RedisRing backend which writes only to one Redis server based on key.

RedisRing is only a proxy to underlying *`ConstRedis`* objects, which represents each Redis instance. It has the same methods set as *`ConstRedis`*. Individual nodes can be accessed through *nodes* attribute.

RedisRing is better suited for using Redis as memcached replacement. With one server failure data kept on that server have to be recreated.

class `redisdb.backends.`**`RedisCopy`**(*location*, *params*)

Redis backend which writes to all Redis servers.

RedisCopy is only a proxy to underlying *`ConstRedis`* objects, which represents each Redis instance. It has the same methods set as *`ConstRedis`*. Individual nodes can be accessed through *nodes* attribute.

Fetching is done only from one server based on key just like in RedisRing. RedisCopy can be seen like master/master configuration where synchronization isn't done by Redis itself but by clients. It's well suited for storing data that should be available even if one server goes down.

[1] Sorted sets http://redis.io/commands/zrange

class redisdb.backends.**ConstRedis**(**args*, ***kwargs*)
Redis client that adds consistent hashing to StrictRedis.

append(*key*, *value*)
Appends the string value to the value at key. If key doesn't already exist, create it with a value of value. Returns the new length of the value at key.

bgrewriteaof()
Tell the Redis server to rewrite the AOF file from data in memory.

bgsave()
Tell the Redis server to save its data to disk. Unlike save(), this method is asynchronous and returns immediately.

bitcount(*key*, *start=None*, *end=None*)
Returns the count of set bits in the value of key. Optional start and end paramaters indicate which bytes to consider

bitop(*operation*, *dest*, **keys*)
Perform a bitwise operation using operation between keys and store the result in dest.

bitpos(*key*, *bit*, *start=None*, *end=None*)
Return the position of the first bit set to 1 or 0 in a string. start and end difines search range. The range is interpreted as a range of bytes and not a range of bits, so start=0 and end=2 means to look at the first three bytes.

blpop(*keys*, *timeout=0*)
LPOP a value off of the first non-empty list named in the keys list.

If none of the lists in keys has a value to LPOP, then block for timeout seconds, or until a value gets pushed on to one of the lists.

If timeout is 0, then block indefinitely.

brpop(*keys*, *timeout=0*)
RPOP a value off of the first non-empty list named in the keys list.

If none of the lists in keys has a value to LPOP, then block for timeout seconds, or until a value gets pushed on to one of the lists.

If timeout is 0, then block indefinitely.

brpoplpush(*src*, *dst*, *timeout=0*)
Pop a value off the tail of src, push it on the head of dst and then return it.

This command blocks until a value is in src or until timeout seconds elapse, whichever is first. A timeout value of 0 blocks forever.

client_getname()
Returns the current connection name

client_kill(*address*)
Disconnects the client at address (ip:port)

client_list()
Returns a list of currently connected clients

client_setname(*name*)
Sets the current connection name

config_get(*pattern='*'*)
Return a dictionary of configuration based on the pattern

config_resetstat()
Reset runtime statistics

config_rewrite()
Rewrite config file with the minimal change to reflect running config

config_set(*name*, *value*)
Set config item `name` with `value`

dbsize()
Returns the number of keys in the current database

debug_object(*key*)
Returns version specific meta information about a given key

decr(*name*, *amount=1*)
Decrements the value of `key` by `amount`. If no key exists, the value will be initialized as 0 - `amount`

delete(**names*)
Delete one or more keys specified by `names`

dump(*name*)
Return a serialized version of the value stored at the specified key. If key does not exist a nil bulk reply is returned.

echo(*value*)
Echo the string back from the server

eval(*script*, *numkeys*, **keys_and_args*)
Execute the Lua `script`, specifying the `numkeys` the script will touch and the key names and argument values in `keys_and_args`. Returns the result of the script.

In practice, use the object returned by `register_script`. This function exists purely for Redis API completion.

evalsha(*sha*, *numkeys*, **keys_and_args*)
Use the `sha` to execute a Lua script already registered via EVAL or SCRIPT LOAD. Specify the `numkeys` the script will touch and the key names and argument values in `keys_and_args`. Returns the result of the script.

In practice, use the object returned by `register_script`. This function exists purely for Redis API completion.

execute_command(**args*, ***options*)
Execute a command and return a parsed response

exists(*name*)
Returns a boolean indicating whether key `name` exists

expire(*name*, *time*)
Set an expire flag on key `name` for `time` seconds. `time` can be represented by an integer or a Python timedelta object.

expireat(*name*, *when*)
Set an expire flag on key `name`. `when` can be represented as an integer indicating unix time or a Python datetime object.

flushall()
Delete all keys in all databases on the current host

flushdb()
Delete all keys in the current database

from_url(*url*, *db=None*, ***kwargs*)
Return a Redis client object configured from the given URL.

For example:

```
redis://[:password]@localhost:6379/0
unix://[:password]@/path/to/socket.sock?db=0
```

There are several ways to specify a database number. The parse function will return the first specified option:

1. A `db` querystring option, e.g. redis://localhost?db=0
2. If using the redis:// scheme, the path argument of the url, e.g. redis://localhost/0
3. The `db` argument to this function.

If none of these options are specified, db=0 is used.

Any additional querystring arguments and keyword arguments will be passed along to the ConnectionPool class's initializer. In the case of conflicting arguments, querystring arguments always win.

get(*name*)
Return the value at key `name`, or None if the key doesn't exist

getbit(*name*, *offset*)
Returns a boolean indicating the value of `offset` in `name`

getrange(*key*, *start*, *end*)
Returns the substring of the string value stored at `key`, determined by the offsets `start` and `end` (both are inclusive)

getset(*name*, *value*)
Sets the value at key `name` to `value` and returns the old value at key `name` atomically.

hdel(*name*, **keys*)
Delete `keys` from hash `name`

hexists(*name*, *key*)
Returns a boolean indicating if `key` exists within hash `name`

hget(*name*, *key*)
Return the value of `key` within the hash `name`

hgetall(*name*)
Return a Python dict of the hash's name/value pairs

hincrby(*name*, *key*, *amount=1*)
Increment the value of `key` in hash `name` by `amount`

hincrbyfloat(*name*, *key*, *amount=1.0*)
Increment the value of `key` in hash `name` by floating `amount`

hkeys(*name*)
Return the list of keys within hash `name`

hlen(*name*)
Return the number of elements in hash `name`

hmget(*name*, *keys*, **args*)
Returns a list of values ordered identically to `keys`

hmset(*name*, *mapping*)
Set key to value within hash `name` for each corresponding key and value from the `mapping` dict.

hscan(*name*, *cursor=0*, *match=None*, *count=None*)
Incrementally return key/value slices in a hash. Also return a cursor indicating the scan position.

`match` allows for filtering the keys by pattern

`count` allows for hint the minimum number of returns

hscan_iter(*name*, *match=None*, *count=None*)
Make an iterator using the HSCAN command so that the client doesn't need to remember the cursor position.

`match` allows for filtering the keys by pattern

`count` allows for hint the minimum number of returns

hset(*name*, *key*, *value*)
Set `key` to `value` within hash `name` Returns 1 if HSET created a new field, otherwise 0

hsetnx(*name*, *key*, *value*)
Set `key` to `value` within hash `name` if `key` does not exist. Returns 1 if HSETNX created a field, otherwise 0.

hvals(*name*)
Return the list of values within hash `name`

incr(*name*, *amount=1*)
Increments the value of `key` by `amount`. If no key exists, the value will be initialized as `amount`

incrby(*name*, *amount=1*)
Increments the value of `key` by `amount`. If no key exists, the value will be initialized as `amount`

incrbyfloat(*name*, *amount=1.0*)
Increments the value at key `name` by floating `amount`. If no key exists, the value will be initialized as `amount`

info(*section=None*)
Returns a dictionary containing information about the Redis server

The `section` option can be used to select a specific section of information

The section option is not supported by older versions of Redis Server, and will generate ResponseError

keys(*pattern='*'*)
Returns a list of keys matching `pattern`

lastsave()
Return a Python datetime object representing the last time the Redis database was saved to disk

lindex(*name*, *index*)
Return the item from list `name` at position `index`

Negative indexes are supported and will return an item at the end of the list

linsert(*name*, *where*, *refvalue*, *value*)
Insert `value` in list `name` either immediately before or after [`where`] `refvalue`

Returns the new length of the list on success or -1 if `refvalue` is not in the list.

llen(*name*)
Return the length of the list `name`

lock(*name*, *timeout=None*, *sleep=0.1*, *blocking_timeout=None*, *lock_class=None*, *thread_local=True*)
Return a new Lock object using key `name` that mimics the behavior of threading.Lock.

If specified, `timeout` indicates a maximum life for the lock. By default, it will remain locked until release() is called.

`sleep` indicates the amount of time to sleep per loop iteration when the lock is in blocking mode and another client is currently holding the lock.

`blocking_timeout` indicates the maximum amount of time in seconds to spend trying to acquire the lock. A value of `None` indicates continue trying forever. `blocking_timeout` can be specified as a float or integer, both representing the number of seconds to wait.

`lock_class` forces the specified lock implementation.

`thread_local` indicates whether the lock token is placed in thread-local storage. By default, the token is placed in thread local storage so that a thread only sees its token, not a token set by another thread. Consider the following timeline:

time: 0, thread-1 acquires *my-lock*, with a timeout of 5 seconds. thread-1 sets the token to "abc"

time: 1, thread-2 blocks trying to acquire *my-lock* using the Lock instance.

time: 5, thread-1 has not yet completed. redis expires the lock key.

time: 5, thread-2 acquired *my-lock* now that it's available. thread-2 sets the token to "xyz"

time: 6, thread-1 finishes its work and calls release(). if the token is *not* stored in thread local storage, then thread-1 would see the token value as "xyz" and would be able to successfully release the thread-2's lock.

In some use cases it's necessary to disable thread local storage. For example, if you have code where one thread acquires a lock and passes that lock instance to a worker thread to release later. If thread local storage isn't disabled in this case, the worker thread won't see the token set by the thread that acquired the lock. Our assumption is that these cases aren't common and as such default to using thread local storage.

lpop(*name*)
Remove and return the first item of the list `name`

lpush(*name*, **values*)
Push `values` onto the head of the list `name`

lpushx(*name*, *value*)
Push `value` onto the head of the list `name` if `name` exists

lrange(*name*, *start*, *end*)
Return a slice of the list `name` between position `start` and `end`

`start` and `end` can be negative numbers just like Python slicing notation

lrem(*name*, *count*, *value*)
Remove the first `count` occurrences of elements equal to `value` from the list stored at `name`.

The count argument influences the operation in the following ways: count > 0: Remove elements equal to value moving from head to tail. count < 0: Remove elements equal to value moving from tail to head. count = 0: Remove all elements equal to value.

lset(*name*, *index*, *value*)
Set `position` of list `name` to `value`

ltrim(*name*, *start*, *end*)
Trim the list `name`, removing all values not within the slice between `start` and `end`

`start` and `end` can be negative numbers just like Python slicing notation

mget(*keys*, **args*)
Returns a list of values ordered identically to `keys`

move(*name*, *db*)
Moves the key `name` to a different Redis database `db`

mset(**args*, ***kwargs*)
Sets key/values based on a mapping. Mapping can be supplied as a single dictionary argument or as kwargs.

msetnx(**args*, ***kwargs*)
Sets key/values based on a mapping if none of the keys are already set. Mapping can be supplied as a single dictionary argument or as kwargs. Returns a boolean indicating if the operation was successful.

object(*infotype*, *key*)
Return the encoding, idletime, or refcount about the key

parse_response(*connection*, *command_name*, ***options*)
Parses a response from the Redis server

persist(*name*)
Removes an expiration on `name`

pexpire(*name*, *time*)
Set an expire flag on key `name` for `time` milliseconds. `time` can be represented by an integer or a Python timedelta object.

pexpireat(*name*, *when*)
Set an expire flag on key `name`. `when` can be represented as an integer representing unix time in milliseconds (unix time * 1000) or a Python datetime object.

pfadd(*name*, **values*)
Adds the specified elements to the specified HyperLogLog.

pfcount(*name*)
Return the approximated cardinality of the set observed by the HyperLogLog at key.

pfmerge(*dest*, **sources*)
Merge N different HyperLogLogs into a single one.

ping()
Ping the Redis server

pipeline(*transaction=True*, *shard_hint=None*)
Return a new pipeline object that can queue multiple commands for later execution. `transaction` indicates whether all commands should be executed atomically. Apart from making a group of operations atomic, pipelines are useful for reducing the back-and-forth overhead between the client and server.

psetex(*name*, *time_ms*, *value*)
Set the value of key `name` to `value` that expires in `time_ms` milliseconds. `time_ms` can be represented by an integer or a Python timedelta object

pttl(*name*)
Returns the number of milliseconds until the key `name` will expire

publish(*channel*, *message*)
Publish `message` on `channel`. Returns the number of subscribers the message was delivered to.

pubsub(***kwargs*)
Return a Publish/Subscribe object. With this object, you can subscribe to channels and listen for messages that get published to them.

randomkey()
Returns the name of a random key

register_script(*script*)
Register a Lua `script` specifying the `keys` it will touch. Returns a Script object that is callable and hides the complexity of deal with scripts, keys, and shas. This is the preferred way to work with Lua scripts.

rename(*src*, *dst*)
Rename key `src` to `dst`

renamenx(*src*, *dst*)
Rename key `src` to `dst` if `dst` doesn't already exist

restore(*name*, *ttl*, *value*)
Create a key using the provided serialized value, previously obtained using DUMP.

rpop(*name*)
Remove and return the last item of the list `name`

rpoplpush(*src*, *dst*)
RPOP a value off of the `src` list and atomically LPUSH it on to the `dst` list. Returns the value.

rpush(*name*, **values*)
Push `values` onto the tail of the list `name`

rpushx(*name*, *value*)
Push `value` onto the tail of the list `name` if `name` exists

sadd(*name*, **values*)
Add `value(s)` to set `name`

save()
Tell the Redis server to save its data to disk, blocking until the save is complete

scan(*cursor=0*, *match=None*, *count=None*)
Incrementally return lists of key names. Also return a cursor indicating the scan position.

`match` allows for filtering the keys by pattern

`count` allows for hint the minimum number of returns

scan_iter(*match=None*, *count=None*)
Make an iterator using the SCAN command so that the client doesn't need to remember the cursor position.

`match` allows for filtering the keys by pattern

`count` allows for hint the minimum number of returns

scard(*name*)
Return the number of elements in set `name`

script_exists(**args*)
Check if a script exists in the script cache by specifying the SHAs of each script as `args`. Returns a list of boolean values indicating if if each already script exists in the cache.

script_flush()
Flush all scripts from the script cache

script_kill()
Kill the currently executing Lua script

script_load(*script*)
Load a Lua `script` into the script cache. Returns the SHA.

sdiff(*keys*, **args*)
Return the difference of sets specified by `keys`

sdiffstore(*dest*, *keys*, **args*)
Store the difference of sets specified by `keys` into a new set named `dest`. Returns the number of keys in the new set.

sentinel(**args*)
Redis Sentinel's SENTINEL command.

sentinel_get_master_addr_by_name(*service_name*)
Returns a (host, port) pair for the given `service_name`

sentinel_master(*service_name*)
Returns a dictionary containing the specified masters state.

sentinel_masters()
Returns a list of dictionaries containing each master's state.

sentinel_monitor(*name*, *ip*, *port*, *quorum*)
Add a new master to Sentinel to be monitored

sentinel_remove(*name*)
Remove a master from Sentinel's monitoring

sentinel_sentinels(*service_name*)
Returns a list of sentinels for `service_name`

sentinel_set(*name*, *option*, *value*)
Set Sentinel monitoring parameters for a given master

sentinel_slaves(*service_name*)
Returns a list of slaves for `service_name`

set(*name*, *value*, *ex=None*, *px=None*, *nx=False*, *xx=False*)
Set the value at key `name` to `value`

`ex` sets an expire flag on key `name` for `ex` seconds.

`px` sets an expire flag on key `name` for `px` milliseconds.

nx if set to True, set the value at key `name` to `value` if it does not already exist.

xx if set to True, set the value at key `name` to `value` if it already exists.

set_response_callback(*command*, *callback*)
Set a custom Response Callback

setbit(*name*, *offset*, *value*)
Flag the `offset` in `name` as `value`. Returns a boolean indicating the previous value of `offset`.

setex(*name*, *time*, *value*)
Set the value of key `name` to `value` that expires in `time` seconds. `time` can be represented by an integer or a Python timedelta object.

setnx(*name*, *value*)
Set the value of key `name` to `value` if key doesn't exist

setrange(*name*, *offset*, *value*)
Overwrite bytes in the value of `name` starting at `offset` with `value`. If `offset` plus the length of `value` exceeds the length of the original value, the new value will be larger than before. If `offset` exceeds the length of the original value, null bytes will be used to pad between the end of the previous value and the start of what's being injected.

Returns the length of the new string.

shutdown()
Shutdown the server

sinter(*keys*, **args*)
Return the intersection of sets specified by `keys`

sinterstore(*dest*, *keys*, **args*)
Store the intersection of sets specified by `keys` into a new set named `dest`. Returns the number of keys in the new set.

sismember(*name*, *value*)
Return a boolean indicating if `value` is a member of set `name`

slaveof(*host=None*, *port=None*)
Set the server to be a replicated slave of the instance identified by the `host` and `port`. If called without arguments, the instance is promoted to a master instead.

slowlog_get(*num=None*)
Get the entries from the slowlog. If `num` is specified, get the most recent `num` items.

slowlog_len()
Get the number of items in the slowlog

slowlog_reset()
Remove all items in the slowlog

smembers(*name*)
Return all members of the set `name`

smove(*src*, *dst*, *value*)
Move `value` from set `src` to set `dst` atomically

sort(*name*, *start=None*, *num=None*, *by=None*, *get=None*, *desc=False*, *alpha=False*, *store=None*, *groups=False*)
Sort and return the list, set or sorted set at `name`.

`start` and `num` allow for paging through the sorted data

by allows using an external key to weight and sort the items. Use an "*" to indicate where in the key the item value is located

get allows for returning items from external keys rather than the sorted data itself. Use an "*" to indicate where int he key the item value is located

`desc` allows for reversing the sort

`alpha` allows for sorting lexicographically rather than numerically

store allows for storing the result of the sort into the key `store`

groups if set to True and if get contains at least two elements, sort will return a list of tuples, each containing the values fetched from the arguments to `get`.

spop(*name*)
Remove and return a random member of set `name`

srandmember(*name*, *number=None*)
If `number` is None, returns a random member of set `name`.

If `number` is supplied, returns a list of `number` random memebers of set `name`. Note this is only available when running Redis 2.6+.

srem(*name*, **values*)
Remove `values` from set `name`

sscan(*name*, *cursor=0*, *match=None*, *count=None*)
Incrementally return lists of elements in a set. Also return a cursor indicating the scan position.

`match` allows for filtering the keys by pattern

`count` allows for hint the minimum number of returns

sscan_iter(*name*, *match=None*, *count=None*)
Make an iterator using the SSCAN command so that the client doesn't need to remember the cursor position.

`match` allows for filtering the keys by pattern

`count` allows for hint the minimum number of returns

strlen(*name*)
Return the number of bytes stored in the value of `name`

substr(*name*, *start*, *end=-1*)
Return a substring of the string at key `name`. `start` and `end` are 0-based integers specifying the portion of the string to return.

sunion(*keys*, **args*)
Return the union of sets specified by `keys`

sunionstore(*dest*, *keys*, **args*)
Store the union of sets specified by `keys` into a new set named `dest`. Returns the number of keys in the new set.

time()
Returns the server time as a 2-item tuple of ints: (seconds since epoch, microseconds into this second).

transaction(*func*, **watches*, ***kwargs*)
Convenience method for executing the callable *func* as a transaction while watching all keys specified in *watches*. The 'func' callable should expect a single argument which is a Pipeline object.

ttl(*name*)
Returns the number of seconds until the key `name` will expire

type(*name*)
Returns the type of key `name`

unwatch()
Unwatches the value at key `name`, or None of the key doesn't exist

watch(**names*)
Watches the values at keys `names`, or None if the key doesn't exist

zadd(*name*, **args*, ***kwargs*)
Set any number of score, element-name pairs to the key `name`. Pairs can be specified in two ways:

As *args, in the form of: score1, name1, score2, name2, ... or as **kwargs, in the form of: name1=score1, name2=score2, ...

The following example would add four values to the 'my-key' key: redis.zadd('my-key', 1.1, 'name1', 2.2, 'name2', name3=3.3, name4=4.4)

zcard(*name*)
Return the number of elements in the sorted set `name`

zcount(*name*, *min*, *max*)
Returns the number of elements in the sorted set at key `name` with a score between `min` and `max`.

zincrby(*name*, *value*, *amount=1*)
Increment the score of `value` in sorted set `name` by `amount`

zinterstore(*dest*, *keys*, *aggregate=None*)
Intersect multiple sorted sets specified by `keys` into a new sorted set, `dest`. Scores in the destination will be aggregated based on the `aggregate`, or SUM if none is provided.

zlexcount(*name*, *min*, *max*)
Return the number of items in the sorted set `name` between the lexicographical range `min` and `max`.

zrange(*name*, *start*, *end*, *desc=False*, *withscores=False*, *score_cast_func=<type 'float'>*)
Return a range of values from sorted set `name` between `start` and `end` sorted in ascending order.

`start` and `end` can be negative, indicating the end of the range.

`desc` a boolean indicating whether to sort the results descendingly

`withscores` indicates to return the scores along with the values. The return type is a list of (value, score) pairs

`score_cast_func` a callable used to cast the score return value

zrangebylex(*name*, *min*, *max*, *start=None*, *num=None*)
Return the lexicographical range of values from sorted set `name` between `min` and `max`.

If `start` and `num` are specified, then return a slice of the range.

zrangebyscore(*name*, *min*, *max*, *start=None*, *num=None*, *withscores=False*, *score_cast_func=<type 'float'>*)
Return a range of values from the sorted set `name` with scores between `min` and `max`.

If `start` and `num` are specified, then return a slice of the range.

`withscores` indicates to return the scores along with the values. The return type is a list of (value, score) pairs

score_cast_func‘ a callable used to cast the score return value

zrank(*name*, *value*)
Returns a 0-based value indicating the rank of `value` in sorted set `name`

zrem(*name*, **values*)
Remove member `values` from sorted set `name`

zremrangebylex(*name*, *min*, *max*)
Remove all elements in the sorted set `name` between the lexicographical range specified by `min` and `max`.

Returns the number of elements removed.

zremrangebyrank(*name*, *min*, *max*)
Remove all elements in the sorted set `name` with ranks between `min` and `max`. Values are 0-based, ordered from smallest score to largest. Values can be negative indicating the highest scores. Returns the number of elements removed

zremrangebyscore(*name*, *min*, *max*)
Remove all elements in the sorted set `name` with scores between `min` and `max`. Returns the number of elements removed.

zrevrange(*name*, *start*, *end*, *withscores=False*, *score_cast_func=<type 'float'>*)
Return a range of values from sorted set `name` between `start` and `end` sorted in descending order.

`start` and `end` can be negative, indicating the end of the range.

`withscores` indicates to return the scores along with the values The return type is a list of (value, score) pairs

`score_cast_func` a callable used to cast the score return value

zrevrangebyscore(*name*, *max*, *min*, *start=None*, *num=None*, *withscores=False*, *score_cast_func=<type 'float'>*)
Return a range of values from the sorted set `name` with scores between `min` and `max` in descending order.

If `start` and `num` are specified, then return a slice of the range.

`withscores` indicates to return the scores along with the values. The return type is a list of (value, score) pairs

`score_cast_func` a callable used to cast the score return value

zrevrank(*name*, *value*)
Returns a 0-based value indicating the descending rank of `value` in sorted set `name`

zscan(*name*, *cursor=0*, *match=None*, *count=None*, *score_cast_func=<type 'float'>*)
Incrementally return lists of elements in a sorted set. Also return a cursor indicating the scan position.

`match` allows for filtering the keys by pattern

`count` allows for hint the minimum number of returns

`score_cast_func` a callable used to cast the score return value

zscan_iter(*name*, *match=None*, *count=None*, *score_cast_func=<type 'float'>*)
Make an iterator using the ZSCAN command so that the client doesn't need to remember the cursor position.

`match` allows for filtering the keys by pattern

`count` allows for hint the minimum number of returns

`score_cast_func` a callable used to cast the score return value

zscore(*name*, *value*)
Return the score of element `value` in sorted set `name`

zunionstore(*dest*, *keys*, *aggregate=None*)
Union multiple sorted sets specified by `keys` into a new sorted set, `dest`. Scores in the destination will be aggregated based on the `aggregate`, or SUM if none is provided.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

r

K

L

M

O

P

R

S

T

U

W

Z