aerospike Documentation

Release

Ronen Botzer

Jan 10, 2018

Contents

`aerospike` is a package which provides a Python client for Aerospike database clusters. The Python client is a CPython module, built on the Aerospike C client.

- *`aerospike`* - the module containing the Client, Query, and Scan Classes.
- *`aerospike.predicates`* is a submodule containing predicate helpers for use with the Query class.
- *`aerospike.exception`* is a submodule containing the exception hierarchy for AerospikeError and its sub-classes.
- *Data_Mapping — Python Data Mappings* How Python types map to Aerospike Server types

See also:

The Python Client Manual for a quick guide.

CHAPTER 1

Content

1.1 `aerospike` — Aerospike Client for Python

The Aerospike client enables you to build an application in Python with an Aerospike cluster as its database. The client manages the connections to the cluster and handles the transactions performed against it.

Data Model

At the top is the `namespace`, a container that has one set of policy rules for all its data, and is similar to the *database* concept in an RDBMS, only distributed across the cluster. A namespace is subdivided into `sets`, similar to *tables*.

Pairs of key-value data called `bins` make up `records`, similar to *columns* of a *row* in a standard RDBMS. Aerospike is schema-less, meaning that you do not need to define your bins in advance.

Records are uniquely identified by their key, and record metadata is contained in an in-memory primary index.

See also:

Architecture Overview and Aerospike Data Model for more information about Aerospike.

`aerospike.`**`client`**(*config*)

Creates a new instance of the Client class. This client can *`connect()`* to the cluster and perform operations against it, such as *`put()`* and *`get()`* records.

Parameters **`config`** (*`dict`*) – the client's configuration.

- **hosts** a required `list` of (address, port, [tls-name]) tuples identifying a node (or multiple nodes) in the cluster. The client will connect to the first available node in the list, the *seed node*, and will learn about the cluster and partition map from it. If tls-name is specified, it must match the tls-name specified in the node's server configuration file and match the server's CA certificate. **Note: use of TLS requires Aerospike Enterprise Edition**
- **lua an optional `dict` containing the paths to two types of Lua modules**

  system_path the location of the system modules such as `aerospike.lua` (default: `/usr/local/aerospike/lua`)

user_path the location of the user's record and stream UDFs . (default: `./`)

- **policies a `dict` of policies**

  read A dictionary containing read policies. See *Read Policies* for available policy fields and values.

  write A dictionary containing write policies. See *Write Policies* for available policy fields and values.

  apply A dictionary containing apply policies. See *Apply Policies* for available policy fields and values.

  operate A dictionary containing operate policies. See *Operate Policies* for available policy fields and values.

  remove A dictionary containing remove policies. See *Remove Policies* for available policy fields and values.

  query A dictionary containing query policies. See *Query Policies* for available policy fields and values.

  scan A dictionary containing scan policies. See *Scan Policies* for available policy fields and values.

  batch A dictionary containing batch policies. See *Batch Policies* for available policy fields and values.

  total_timeout default connection timeout in milliseconds (**Deprecated**: set this the individual policy dictionaries)

  key default key policy, with values such as *`aerospike.POLICY_KEY_DIGEST`* (**Deprecated**: set this individually in the 'read', 'write', 'apply', 'operate', 'remove' policy dictionaries)

  exists default exists policy, with values such as *`aerospike.POLICY_EXISTS_CREATE`* (**Deprecated**: set in the 'write' policies dictionary)

  max_retries a `int` representing the number of times to retry a transaction (**Deprecated**: set this the individual policy dictionaries)

  consistency_level default consistency level policy, with values such as *`aerospike.POLICY_CONSISTENCY_ONE`* (**Deprecated**: set this individually as needed in the 'read','operate', 'batch' policy dictionaries)

  replica default replica policy, with values such as *`aerospike.POLICY_REPLICA_MASTER`* (**Deprecated**: set this in one or all of the 'read', 'write', 'apply', 'operate', 'remove' policy dictionaries)

  commit_level default commit level policy, with values such as *`aerospike.POLICY_COMMIT_LEVEL_ALL`* (**Deprecated**: set this as needed individually in the 'write', 'apply', 'operate', 'remove' policy dictionaries)

- **shm a `dict` with optional shared-memory cluster tending parameters. Shared-memory cluster tending is o**

  max_nodes maximum number of nodes allowed. Pad so new nodes can be added without configuration changes (default: 16)

  max_namespaces similarly pad (default: 8)

  takeover_threshold_sec take over tending if the cluster hasn't been checked for this many seconds (default: 30)

shm_key explicitly set the shm key for this client. If **use_shared_connection** is not set, or set to *False*, the user must provide a value for this field in order for shared memory to work correctly. If , and only if, **use_shared_connection** is set to *True*, the key will be implicitly evaluated per unique hostname, and can be inspected with *shm_key()* . It is still possible to specify a key when using **use_shared_connection** = *True*. (default: 0xA7000000)

- **use_shared_connection** `bool` indicating whether this instance should share its connection to the Aerospike cluster with other client instances in the same process. (default: `False`)
- **tls a `dict` of optional TLS configuration parameters. TLS usage requires Aerospike Enterprise Edition**

  enable a `bool` indicating whether tls should be enabled or not. Default: `False`

  cafile `str` Path to a trusted CA certificate file. By default TLS will use system standard trusted CA certificates

  capath `str` Path to a directory of trusted certificates. See the OpenSSL SSL_CTX_load_verify_locations manual page for more information about the format of the directory.

  protocols Specifies enabled protocols. This format is the same as Apache's SSLProtocol documented at https://httpd.apache.org/docs/current/mod/mod_ssl.html#sslprotocol . If not specified the client will use "-all +TLSv1.2".

  cipher_suite `str` Specifies enabled cipher suites. The format is the same as OpenSSL's Cipher List Format documented at https://www.openssl.org/docs/manmaster/apps/ciphers.html .If not specified the OpenSSL default cipher suite described in the ciphers documentation will be used. If you are not sure what cipher suite to select this option is best left unspecified

  keyfile `str` Path to the client's key for mutual authentication. By default mutual authentication is disabled.

  cert_blacklist `str` Path to a certificate blacklist file. The file should contain one line for each blacklisted certificate. Each line starts with the certificate serial number expressed in hex. Each entry may optionally specify the issuer name of the certificate (serial numbers are only required to be unique per issuer). Example records: 867EC87482B2 /C=US/ST=CA/O=Acme/OU=Engineering/CN=Test Chain CA E2D4B0E570F9EF8E885C065899886461

  certfile `str` Path to the client's certificate chain file for mutual authentication. By default mutual authentication is disabled.

  crl_check `bool` Enable CRL checking for the certificate chain leaf certificate. An error occurs if a suitable CRL cannot be found. By default CRL checking is disabled.

  crl_check_all `bool` Enable CRL checking for the entire certificate chain. An error occurs if a suitable CRL cannot be found. By default CRL checking is disabled.

  log_session_info `bool` Log session information for each connection.
- **serialization** an optional instance-level `tuple()` of (serializer, deserializer). Takes precedence over a class serializer registered with *set_serializer()*.
- **thread_pool_size** number of threads in the pool that is used in batch/scan/query commands (default: 16)
- **max_socket_idle Maximum socket idle time in seconds. Connection pools will discard sockets that have** been idle longer than the maximum. The value is limited to 24 hours (86400). It's important to set this value to a few seconds less than the server's proto-fd-idle-ms (default 60000 milliseconds or 1 minute), so the client does not attempt to use a socket

that has already been reaped by the server. Default: 0 seconds (disabled) for non-TLS connections, 55 seconds for TLS connections.

- **max_conns_per_node** maximum number of pipeline connections allowed for each node
- **batch_direct** whether to use the batch-direct protocol (default: `False`, so will use batch-index if available) (**Deprecated**: set 'use_batch_direct' in the batch policy dictionary)
- **tend_interval** polling interval in milliseconds for tending the cluster (default: 1000)
- **compression_threshold** compress data for transmission if the object size is greater than a given number of bytes (default: 0, meaning 'never compress') (**Deprecated**, set this in the 'write' policy dictionary)
- **cluster_name** only server nodes matching this name will be used when determining the cluster

Returns an instance of the *aerospike.Client* class.

See also:

Shared Memory and Per-Transaction Consistency Guarantees.

```
import aerospike

# configure the client to first connect to a cluster node at 127.0.0.1
# the client will learn about the other nodes in the cluster from the
# seed node.
# in this configuration shared-memory cluster tending is turned on,
# which is appropriate for a multi-process context, such as a webserver
config = {
    'hosts':    [ ('127.0.0.1', 3000) ],
    'policies': {'read': {total_timeout': 1000}},
    'shm':      { }}
client = aerospike.client(config)
```

Changed in version 2.0.0.

```
import aerospike
import sys

# NOTE: Use of TLS Requires Aerospike Enterprise Server Version >= 3.11 and␣
↪Python Client version 2.1.0 or greater
# To view Instructions for server configuration for TLS see https://www.aerospike.
↪com/docs/guide/security/tls.html
tls_name = "some-server-tls-name"
tls_ip = "127.0.0.1"
tls_port = 4333

# If tls-name is specified, it must match the tls-name specified in the node's␣
↪server configuration file
# and match the server's CA certificate.
tls_host_tuple = (tls_ip, tls_port, tls_name)
hosts = [tls_host_tuple]

# Example configuration which will use TLS with the specifed cafile
tls_config = {
    "cafile": "/path/to/cacert.pem",
    "enable": True
}

client = aerospike.client({
    "hosts": hosts,
    "tls": tls_config
```

```
})
try:
    client.connect()
except Exception as e:
    print(e)
    print("Failed to connect")
    sys.exit()

key = ('test', 'demo', 1)
client.put(key, {'aerospike': 'aerospike'})
print(client.get(key))
```

aerospike.**null**()
A type for distinguishing a server-side null from a Python None. Replaces the constant aerospike.null.

Returns a type representing the server-side type as_null.

New in version 2.0.1.

aerospike.**calc_digest**(*ns*, *set*, *key*) → bytearray
Calculate the digest of a particular key. See: *Key Tuple*.

Parameters

- **ns** (*str*) – the namespace in the aerospike cluster.
- **set** (*str*) – the set name.
- **key** (str, int or bytearray) – the primary key identifier of the record within the set.

Returns a RIPEMD-160 digest of the input tuple.

Return type bytearray

```
import aerospike
import pprint

digest = aerospike.calc_digest("test", "demo", 1 )
pp.pprint(digest)
```

Serialization

Note: By default, the *aerospike.Client* maps the supported types int, str, float, bytearray, list, dict to matching aerospike server types (int, string, double, bytes, list, map). When an unsupported type is encountered, the module uses cPickle to serialize and deserialize the data, storing it into *as_bytes* of type 'Python' (AS_BYTES_PYTHON).

The functions *set_serializer()* and *set_deserializer()* allow for user-defined functions to handle serialization, instead. The serialized data is stored as 'Generic' *as_bytes* of type (AS_BYTES_BLOB). The *serialization* config param of *aerospike.client()* registers an instance-level pair of functions that handle serialization.

aerospike.**set_serializer**(*callback*)
Register a user-defined serializer available to all *aerospike.Client* instances.

Parameters **callback** (*callable*) – the function to invoke for serialization.

See also:

To use this function with *put()* the argument to *serializer* should be *aerospike.SERIALIZER_USER*.

```
import aerospike
import json

def my_serializer(val):
    return json.dumps(val)

aerospike.set_serializer(my_serializer)
```

New in version 1.0.39.

aerospike.**set_deserializer**(*callback*)

Register a user-defined deserializer available to all *aerospike.Client* instances. Once registered, all read methods (such as *get()*) will run bins containing 'Generic' *as_bytes* of type (AS_BYTES_BLOB) through this deserializer.

Parameters **callback** (*callable*) – the function to invoke for deserialization.

aerospike.**unset_serializers**()

Deregister the user-defined de/serializer available from *aerospike.Client* instances.

New in version 1.0.53.

Note: Serialization Examples

The following example shows the three modes of serialization - built-in, class-level user functions, instance-level user functions:

```
from __future__ import print_function
import aerospike
import marshal
import json

def go_marshal(val):
    return marshal.dumps(val)

def demarshal(val):
    return marshal.loads(val)

def jsonize(val):
    return json.dumps(val)

def dejsonize(val):
    return json.loads(val)

aerospike.set_serializer(go_marshal)
aerospike.set_deserializer(demarshal)
config = {'hosts':[('127.0.0.1', 3000)]}
client = aerospike.client(config).connect()
config['serialization'] = (jsonize,dejsonize)
client2 = aerospike.client(config).connect()

for i in xrange(1, 4):
    try:
        client.remove(('test', 'demo', 'foo' + i))
    except:
        pass
```

```
bin_ = {'t': (1, 2, 3)} # tuple is an unsupported type
print("Use the built-in serialization (cPickle)")
client.put(('test','demo','foo1'), bin_)
(key, meta, bins) = client.get(('test','demo','foo1'))
print(bins)

print("Use the class-level user-defined serialization (marshal)")
client.put(('test','demo','foo2'), bin_, serializer=aerospike.SERIALIZER_USER)
(key, meta, bins) = client.get(('test','demo','foo2'))
print(bins)

print("Use the instance-level user-defined serialization (json)")
client2.put(('test','demo','foo3'), bin_, serializer=aerospike.SERIALIZER_USER)
# notice that json-encoding a tuple produces a list
(key, meta, bins) = client2.get(('test','demo','foo3'))
print(bins)
client.close()
```

The expected output is:

```
Use the built-in serialization (cPickle)
{'i': 321, 't': (1, 2, 3)}
Use the class-level user-defined serialization (marshal)
{'i': 321, 't': (1, 2, 3)}
Use the instance-level user-defined serialization (json)
{'i': 321, 't': [1, 2, 3]}
```

While AQL shows the records as having the following structure:

```
aql> select i,t from test.demo where PK='foo1'
+-----+-------------------------------------------------+
| i   | t                                               |
+-----+-------------------------------------------------+
| 321 | 28 49 31 0A 49 32 0A 49 33 0A 74 70 31 0A 2E |
+-----+-------------------------------------------------+
1 row in set (0.000 secs)

aql> select i,t from test.demo where PK='foo2'
+-----+-------------------------------------------------------------------+
| i   | t                                                                 |
+-----+-------------------------------------------------------------------+
| 321 | 28 03 00 00 00 69 01 00 00 00 69 02 00 00 00 69 03 00 00 00 |
+-----+-------------------------------------------------------------------+
1 row in set (0.000 secs)

aql> select i,t from test.demo where PK='foo3'
+-----+----------------------------+
| i   | t                          |
+-----+----------------------------+
| 321 | 5B 31 2C 20 32 2C 20 33 5D |
+-----+----------------------------+
1 row in set (0.000 secs)
```

Logging

aerospike.**set_log_handler**(*callback*)

Set a user-defined function as the log handler for all aerospike objects. The *callback* is invoked whenever a log event passing the logging level threshold is encountered.

Parameters **callback** (*callable*) – the function used as the logging handler.

Note: The callback function must have the five parameters (level, func, path, line, msg)

```
from __future__ import print_function
import aerospike

def as_logger(level, func, path, line, msg):
def as_logger(level, func, myfile, line, msg):
    print("**", myfile, line, func, ':: ', msg, "**")

aerospike.set_log_level(aerospike.LOG_LEVEL_DEBUG)
aerospike.set_log_handler(as_logger)
```

---

aerospike.**set_log_level**(*log_level*)

Declare the logging level threshold for the log handler.

Parameters **log_level** (*int*) – one of the *Log Level* constant values.

Geospatial

aerospike.**geodata**([*geo_data*])

Helper for creating an instance of the *GeoJSON* class. Used to wrap a geospatial object, such as a point, polygon or circle.

Parameters **geo_data** (*dict*) – a `dict` representing the geospatial data.

Returns an instance of the *aerospike.GeoJSON* class.

```
import aerospike

# Create GeoJSON point using WGS84 coordinates.
latitude = 45.920278
longitude = 63.342222
loc = aerospike.geodata({'type': 'Point',
                         'coordinates': [longitude, latitude]})
```

New in version 1.0.54.

aerospike.**geojson**([*geojson_str*])

Helper for creating an instance of the *GeoJSON* class from a raw GeoJSON `str`.

Parameters **geojson_str** (*dict*) – a `str` of raw GeoJSON.

Returns an instance of the *aerospike.GeoJSON* class.

```
import aerospike

# Create GeoJSON point using WGS84 coordinates.
loc = aerospike.geojson('{"type": "Point", "coordinates": [-80.604333, 28.608389]}
↪')
```

New in version 1.0.54.

1.1.1 Operators

Operators for the multi-ops method *operate()*.

aerospike.**OPERATOR_WRITE**
Write a value into a bin

```
{
    "op" : aerospike.OPERATOR_WRITE,
    "bin": "name",
    "val": "Peanut"
}
```

aerospike.**OPERATOR_APPEND**
Append to a bin with `str` type data

```
{
    "op" : aerospike.OPERATOR_APPEND,
    "bin": "name",
    "val": "Mr. "
}
```

aerospike.**OPERATOR_PREPEND**
Prepend to a bin with `str` type data

```
{
    "op" : aerospike.OPERATOR_PREPEND,
    "bin": "name",
    "val": " Esq."
}
```

aerospike.**OPERATOR_INCR**
Increment a bin with `int` or `float` type data

```
{
    "op" : aerospike.OPERATOR_INCR,
    "bin": "age",
    "val": 1
}
```

aerospike.**OPERATOR_READ**
Read a specific bin

```
{
    "op" : aerospike.OPERATOR_READ,
    "bin": "name"
}
```

aerospike.**OPERATOR_TOUCH**
Touch a record, setting its TTL. May be combined with *OPERATOR_READ*

```
{
    "op" : aerospike.OPERATOR_TOUCH
}
```

aerospike.**OP_LIST_APPEND**
Append an element to a bin with `list` type data

```
{
    "op" : aerospike.OP_LIST_APPEND,
    "bin": "events",
    "val": 1234
}
```

aerospike.**OP_LIST_APPEND_ITEMS**
Extend a bin with `list` type data with a list of items

```
{
    "op" : aerospike.OP_LIST_APPEND_ITEMS,
    "bin": "events",
    "val": [ 123, 456 ]
}
```

aerospike.**OP_LIST_INSERT**
Insert an element at a specified index of a bin with `list` type data

```
{
    "op" : aerospike.OP_LIST_INSERT,
    "bin": "events",
    "index": 2,
    "val": 1234
}
```

aerospike.**OP_LIST_INSERT_ITEMS**
Insert the items at a specified index of a bin with `list` type data

```
{
    "op" : aerospike.OP_LIST_INSERT_ITEMS,
    "bin": "events",
    "index": 2,
    "val": [ 123, 456 ]
}
```

aerospike.**OP_LIST_INCREMENT**
Increment the value of an item at the given index in a list stored in the specified bin

```
{
    "op": aerospike.OP_LIST_INCREMENT,
    "bin": "bin_name",
    "index": 2,
    "val": 5
}
```

aerospike.**OP_LIST_POP**
Remove and return the element at a specified index of a bin with `list` type data

```
{
    "op" : aerospike.OP_LIST_POP, # removes and returns a value
```

```
    "bin": "events",
    "index": 2
}
```

aerospike.**OP_LIST_POP_RANGE**
Remove and return a list of elements at a specified index range of a bin with `list` type data

```
{
    "op" : aerospike.OP_LIST_POP_RANGE,
    "bin": "events",
    "index": 2,
    "val": 3 # remove and return 3 elements starting at index 2
}
```

aerospike.**OP_LIST_REMOVE**
Remove the element at a specified index of a bin with `list` type data

```
{
    "op" : aerospike.OP_LIST_REMOVE, # remove a value
    "bin": "events",
    "index": 2
}
```

aerospike.**OP_LIST_REMOVE_RANGE**
Remove a list of elements at a specified index range of a bin with `list` type data

```
{
    "op" : aerospike.OP_LIST_REMOVE_RANGE,
    "bin": "events",
    "index": 2,
    "val": 3 # remove 3 elements starting at index 2
}
```

aerospike.**OP_LIST_CLEAR**
Remove all the elements in a bin with `list` type data

```
 {
    "op" : aerospike.OP_LIST_CLEAR,
    "bin": "events"
}
```

aerospike.**OP_LIST_SET**
Set the element *val* in a specified index of a bin with `list` type data

```
{
    "op" : aerospike.OP_LIST_SET,
    "bin": "events",
    "index": 2,
    "val": "latest event at index 2" # set this value at index 2
}
```

aerospike.**OP_LIST_GET**
Get the element at a specified index of a bin with `list` type data

```
{
    "op" : aerospike.OP_LIST_GET,
    "bin": "events",
```

```
    "index": 2
}
```

aerospike.**OP_LIST_GET_RANGE**
Get the list of elements starting at a specified index of a bin with `list` type data

```
{
    "op" : aerospike.OP_LIST_GET_RANGE,
    "bin": "events",
    "index": 2,
    "val": 3 # get 3 elements starting at index 2
}
```

aerospike.**OP_LIST_TRIM**
Remove elements from a bin with `list` type data which are not within the range starting at a given *index* plus *val*

```
{
    "op" : aerospike.OP_LIST_TRIM,
    "bin": "events",
    "index": 2,
    "val": 3 # remove all elements not in the range between index 2 and index 2 +␣
↪3
}
```

aerospike.**OP_LIST_SIZE**
Count the number of elements in a bin with `list` type data

```
{
    "op" : aerospike.OP_LIST_SIZE,
    "bin": "events" # gets the size of a list contained in the bin
}
```

aerospike.**OP_MAP_SET_POLICY**
Set the policy for a map bin. The policy controls the write mode and the ordering of the map entries.

```
{
    "op" : aerospike.OP_MAP_SET_POLICY,
    "bin": "scores",
    "map_policy": {"map_write_mode": Aeorspike.MAP_UPDATE, "map_order": Aerospike.
↪MAP_KEY_VALUE_ORDERED}
}
```

aerospike.**OP_MAP_PUT**
Put a key/value pair into a map. Operator accepts an optional map_policy dictionary (see OP_MAP_SET_POLICY for an example)

```
{
    "op" : aerospike.OP_MAP_PUT,
    "bin": "my_map",
    "key": "age",
    "val": 97
}
```

OP_MAP_PUT_ITEMS. Operator accepts an optional map_policy dictionary (see OP_MAP_SET_POLIC
Put a dictionary of key/value pairs into a map.

```
{
    "op" : aerospike.OP_MAP_PUT_ITEMS,
    "bin": "my_map",
    "val": {"name": "bubba", "occupation": "dancer"}
}
```

OP_MAP_INCREMENT. Operator accepts an optional map_policy dictionary (see OP_MAP_SET_POLICY
Increment the value of map entry by the given "val" argument.

```
{
    "op" : aerospike.OP_MAP_INCREMENT,
    "bin": "my_map",
    "key": "age",
    "val": 1
}
```

OP_MAP_DECREMENT. Operator accepts an optional map_policy dictionary (see OP_MAP_SET_POLICY
Decrement the value of map entry by the given "val" argument.

```
{
    "op" : aerospike.OP_MAP_DECREMENT,
    "bin": "my_map",
    "key": "age",
    "val": 1
}
```

aerospike.**OP_MAP_SIZE**
Return the number of entries in the given map bin.

```
{
    "op" : aerospike.OP_MAP_SIZE,
    "bin": "my_map"
}
```

aerospike.**OP_MAP_CLEAR**
Remove all entries from the given map bin.

```
{
    "op" : aerospike.OP_MAP_CLEAR,
    "bin": "my_map"
}
```

Note that if "return_type" is not specified in the parameters for a map operation, the default is aerospike.MAP_RETURN_NONE

aerospike.**OP_MAP_REMOVE_BY_KEY**
Remove the first entry from the map bin that matches the given key.

```
{
    "op" : aerospike.OP_MAP_REMOVE_BY_KEY,
    "bin": "my_map",
    "key": "age",
    "return_type": aerospike.MAP_RETURN_VALUE
}
```

aerospike.**OP_MAP_REMOVE_BY_KEY_LIST**
Remove the entries from the map bin that match the list of given keys.

```
{
    "op" : aerospike.OP_MAP_REMOVE_BY_KEY_LIST,
    "bin": "my_map",
    "val": ["name", "rank", "serial"]
}
```

aerospike.**OP_MAP_REMOVE_BY_KEY_RANGE**
Remove the entries from the map bin that have keys which fall between the given "key" (inclusive) and "val" (exclusive).

```
{
    "op" : aerospike.OP_MAP_REMOVE_BY_KEY_RANGE,
    "bin": "my_map",
    "key": "i",
    "val": "j",
    "return_type": aerospike.MAP_RETURN_KEY_VALUE
}
```

aerospike.**OP_MAP_REMOVE_BY_VALUE**
Remove the entry or entries from the map bin that have values which match the given "val" parameter.

```
{
    "op" : aerospike.OP_MAP_REMOVE_BY_VALUE,
    "bin": "my_map",
    "val": 97,
    "return_type": aerospike.MAP_RETURN_KEY
}
```

aerospike.**OP_MAP_REMOVE_BY_VALUE_LIST**
Remove the entries from the map bin that have values which match the list of values given in the "val" parameter.

```
{
    "op" : aerospike.OP_MAP_REMOVE_BY_VALUE_LIST,
    "bin": "my_map",
    "val": [97, 98, 99],
    "return_type": aerospike.MAP_RETURN_KEY
}
```

aerospike.**OP_MAP_REMOVE_BY_VALUE_RANGE**
Remove the entries from the map bin that have values starting with the given "val" parameter (inclusive) up to the given "range" parameter (exclusive).

```
{
    "op" : aerospike.OP_MAP_REMOVE_BY_VALUE_RANGE,
    "bin": "my_map",
    "val": 97,
    "range": 100,
    "return_type": aerospike.MAP_RETURN_KEY
}
```

aerospike.**OP_MAP_REMOVE_BY_INDEX**
Remove the entry from the map bin at the given "index" location.

```
{
    "op" : aerospike.OP_MAP_REMOVE_BY_INDEX,
    "bin": "my_map",
    "index": 0,
```

```
    "return_type": aerospike.MAP_RETURN_KEY_VALUE
}
```

aerospike.**OP_MAP_REMOVE_BY_INDEX_RANGE**
Remove the entries from the map bin starting at the given "index" location and removing "range" items.

```
{
    "op" : aerospike.OP_MAP_REMOVE_BY_INDEX_RANGE,
    "bin": "my_map",
    "index": 0,
    "val": 2,
    "return_type": aerospike.MAP_RETURN_KEY_VALUE
}
```

aerospike.**OP_MAP_REMOVE_BY_RANK**
Remove the first entry from the map bin that has a value with a rank matching the given "index".

```
{
    "op" : aerospike.OP_MAP_REMOVE_BY_RANK,
    "bin": "my_map",
    "index": 10
}
```

aerospike.**OP_MAP_REMOVE_BY_RANK_RANGE**
Remove the entries from the map bin that have values with a rank starting at the given "index" and removing "range" items.

```
{
    "op" : aerospike.OP_MAP_REMOVE_BY_RANK_RANGE,
    "bin": "my_map",
    "index": 10,
    "val": 2,
    "return_type": aerospike.MAP_RETURN_KEY_VALUE
}
```

aerospike.**OP_MAP_GET_BY_KEY**
Return the entry from the map bin that which has a key that matches the given "key" parameter.

```
{
    "op" : aerospike.OP_MAP_GET_BY_KEY,
    "bin": "my_map",
    "key": "age",
    "return_type": aerospike.MAP_RETURN_KEY_VALUE
}
```

aerospike.**OP_MAP_GET_BY_KEY_RANGE**
Return the entries from the map bin that have keys which fall between the given "key" (inclusive) and "val" (exclusive).

```
{
    "op" : aerospike.OP_MAP_GET_BY_KEY_RANGE,
    "bin": "my_map",
    "key": "i",
    "range": "j",
    "return_type": aerospike.MAP_RETURN_KEY_VALUE
}
```

aerospike.**OP_MAP_GET_BY_VALUE**
Return the entry or entries from the map bin that have values which match the given "val" parameter.

```
{
    "op" : aerospike.OP_MAP_GET_BY_VALUE,
    "bin": "my_map",
    "val": 97,
    "return_type": aerospike.MAP_RETURN_KEY
}
```

aerospike.**OP_MAP_GET_BY_VALUE_RANGE**
Return the entries from the map bin that have values starting with the given "val" parameter (inclusive) up to the given "range" parameter (exclusive).

```
{
    "op" : aerospike.OP_MAP_GET_BY_VALUE_RANGE,
    "bin": "my_map",
    "val": 97,
    "range": 100,
    "return_type": aerospike.MAP_RETURN_KEY
}
```

aerospike.**OP_MAP_GET_BY_INDEX**
Return the entry from the map bin at the given "index" location.

```
{
    "op" : aerospike.OP_MAP_GET_BY_INDEX,
    "bin": "my_map",
    "index": 0,
    "return_type": aerospike.MAP_RETURN_KEY_VALUE
}
```

aerospike.**OP_MAP_GET_BY_INDEX_RANGE**
Return the entries from the map bin starting at the given "index" location and removing "range" items.

```
{
    "op" : aerospike.OP_MAP_GET_BY_INDEX_RANGE,
    "bin": "my_map",
    "index": 0,
    "val": 2,
    "return_type": aerospike.MAP_RETURN_KEY_VALUE
}
```

aerospike.**OP_MAP_GET_BY_RANK**
Return the first entry from the map bin that has a value with a rank matching the given "index".

```
{
    "op" : aerospike.OP_MAP_GET_BY_RANK,
    "bin": "my_map",
    "index": 10
}
```

aerospike.**OP_MAP_GET_BY_RANK_RANGE**
Return the entries from the map bin that have values with a rank starting at the given "index" and removing "range" items.

```
{
    "op" : aerospike.OP_MAP_GET_BY_RANK_RANGE,
```

```
    "bin": "my_map",
    "index": 10,
    "val": 2,
    "return_type": aerospike.MAP_RETURN_KEY_VALUE
}
```

Changed in version 2.0.4.

1.1.2 Policies

Commit Level Policy Options

Specifies the number of replicas required to be successfully committed before returning success in a write operation to provide the desired consistency guarantee.

`aerospike.`**`POLICY_COMMIT_LEVEL_ALL`**
Return succcess only after successfully committing all replicas

`aerospike.`**`POLICY_COMMIT_LEVEL_MASTER`**
Return succcess after successfully committing the master replica

Consistency Level Policy Options

Specifies the number of replicas to be consulted in a read operation to provide the desired consistency guarantee.

`aerospike.`**`POLICY_CONSISTENCY_ONE`**
Involve a single replica in the operation

`aerospike.`**`POLICY_CONSISTENCY_ALL`**
Involve all replicas in the operation

Existence Policy Options

Specifies the behavior for writing the record depending whether or not it exists.

`aerospike.`**`POLICY_EXISTS_CREATE`**
Create a record, ONLY if it doesn't exist

`aerospike.`**`POLICY_EXISTS_CREATE_OR_REPLACE`**
Completely replace a record if it exists, otherwise create it

`aerospike.`**`POLICY_EXISTS_IGNORE`**
Write the record, regardless of existence. (i.e. create or update)

`aerospike.`**`POLICY_EXISTS_REPLACE`**
Completely replace a record, ONLY if it exists

`aerospike.`**`POLICY_EXISTS_UPDATE`**
Update a record, ONLY if it exists

Generation Policy Options

Specifies the behavior of record modifications with regard to the generation value.

aerospike.**POLICY_GEN_IGNORE**
Write a record, regardless of generation

aerospike.**POLICY_GEN_EQ**
Write a record, ONLY if generations are equal

aerospike.**POLICY_GEN_GT**
Write a record, ONLY if local generation is greater-than remote generation

Key Policy Options

Specifies the behavior for whether keys or digests should be sent to the cluster.

aerospike.**POLICY_KEY_DIGEST**
Calculate the digest on the client-side and send it to the server

aerospike.**POLICY_KEY_SEND**
Send the key in addition to the digest. This policy causes a write operation to store the key on the server

Replica Options

Specifies which partition replica to read from.

aerospike.**POLICY_REPLICA_MASTER**
Read from the partition master replica node

aerospike.**POLICY_REPLICA_ANY**
Distribute reads across nodes containing key's master and replicated partition in round-robin fashion. Currently restricted to master and one prole.

Retry Policy Options

Specifies the behavior of failed operations.

aerospike.**POLICY_RETRY_NONE**
Only attempt an operation once

aerospike.**POLICY_RETRY_ONCE**
If an operation fails, attempt the operation one more time

1.1.3 Scan Constants

aerospike.**SCAN_PRIORITY_AUTO**

aerospike.**SCAN_PRIORITY_HIGH**

aerospike.**SCAN_PRIORITY_LOW**

aerospike.**SCAN_PRIORITY_MEDIUM**

aerospike.**SCAN_STATUS_ABORTED**
Deprecated since version 1.0.50: used by *scan_info()*

aerospike.**SCAN_STATUS_COMPLETED**
Deprecated since version 1.0.50: used by *scan_info()*

aerospike.**SCAN_STATUS_INPROGRESS**
Deprecated since version 1.0.50: used by *scan_info()*

aerospike.**SCAN_STATUS_UNDEF**
Deprecated since version 1.0.50: used by *scan_info()*

New in version 1.0.39.

1.1.4 Job Constants

aerospike.**JOB_SCAN**
Scan job type argument for the module parameter of *job_info()*

aerospike.**JOB_QUERY**
Query job type argument for the module parameter of *job_info()*

aerospike.**JOB_STATUS_UNDEF**

aerospike.**JOB_STATUS_INPROGRESS**

aerospike.**JOB_STATUS_COMPLETED**

New in version 1.0.50.

1.1.5 Serialization Constants

aerospike.**SERIALIZER_PYTHON**
Use the cPickle serializer to handle unsupported types (default)

aerospike.**SERIALIZER_USER**
Use a user-defined serializer to handle unsupported types. Must have been registered for the aerospike class or configured for the Client object

aerospike.**SERIALIZER_NONE**
Do not serialize bins whose data type is unsupported

New in version 1.0.47.

1.1.6 Miscellaneous

aerospike.**__version__**
A `str` containing the module's version.

New in version 1.0.54.

aerospike.**null**
A value for distinguishing a server-side null from a Python `None` .

Deprecated since version 2.0.1: use the function *aerospike.null()* instead.

aerospike.**UDF_TYPE_LUA**

aerospike.**INDEX_STRING**
An index whose values are of the aerospike string data type

aerospike.**INDEX_NUMERIC**
An index whose values are of the aerospike integer data type

aerospike.**INDEX_GEO2DSPHERE**
An index whose values are of the aerospike GetJSON data type

See also:

Data Types.

aerospike.**INDEX_TYPE_LIST**
Index a bin whose contents is an aerospike list

aerospike.**INDEX_TYPE_MAPKEYS**
Index the keys of a bin whose contents is an aerospike map

aerospike.**INDEX_TYPE_MAPVALUES**
Index the values of a bin whose contents is an aerospike map

1.1.7 Log Level

aerospike.**LOG_LEVEL_TRACE**

aerospike.**LOG_LEVEL_DEBUG**

aerospike.**LOG_LEVEL_INFO**

aerospike.**LOG_LEVEL_WARN**

aerospike.**LOG_LEVEL_ERROR**

aerospike.**LOG_LEVEL_OFF**

1.1.8 Privileges

Permission codes define the type of permission granted for a user's role.

aerospike.**PRIV_READ**
The user is granted read access.

aerospike.**PRIV_READ_WRITE**
The user is granted read and write access.

aerospike.**PRIV_READ_WRITE_UDF**
The user is granted read and write access, and the ability to invoke UDFs.

aerospike.**PRIV_SYS_ADMIN**
The user is granted the ability to perform system administration operations. Global scope only.

aerospike.**PRIV_USER_ADMIN**
The user is granted the ability to perform user administration operations. Global scope only.

aerospike.**PRIV_DATA_ADMIN**
User can perform systems administration functions on a database that do not involve user administration. Examples include setting dynamic server configuration. Global scope only.

1.1.9 Map Return Types

Return types used by various map operations

aerospike.**MAP_RETURN_NONE**
Do not return any value.

aerospike.**MAP_RETURN_INDEX**
Return key index order.

aerospike.**MAP_RETURN_REVERSE_INDEX**
Return reverse key order.

aerospike.**MAP_RETURN_RANK**
Return value order.

aerospike.**MAP_RETURN_REVERSE_RANK**
Return reserve value order.

aerospike.**MAP_RETURN_COUNT**
Return count of items selected.

aerospike.**MAP_RETURN_KEY**
Return key for single key read and key list for range read.

aerospike.**MAP_RETURN_VALUE**
Return value for single key read and value list for range read.

aerospike.**MAP_RETURN_KEY_VALUE**
Return key/value items. Note that key/value pairs will be returned as a list of tuples (i.e. [(key1, value1), (key2, value2)])

1.2 Data_Mapping — Python Data Mappings

How Python types map to server types

Note: By default, the *aerospike.Client* maps the supported types `int`, `str`, `float`, `bytearray`, `list`, `dict` to matching aerospike server types (int, string, double, blob, list, map). When an unsupported type is encountered, the module uses cPickle to serialize and deserialize the data, storing it into a blob of type 'Python' (AS_BYTES_PYTHON).

The functions *set_serializer()* and *set_deserializer()* allow for user-defined functions to handle serialization, instead. The user provided function will be run instead of cPickle. The serialized data is stored as type (AS_BYTES_BLOB). This type allows the storage of binary data readable by Aerospike Clients in other languages. The *serialization* config param of *aerospike.client()* registers an instance-level pair of functions that handle serialization.

Unless a user specified serializer has been provided, all other types will be stored as Python specific bytes. Python specific bytes may not be readable by Aerospike Clients for other languages.

The following table shows which Python types map directly to Aerospike server types.

Python Type	server type
int	integer
long	integer
str	string
unicode	string
float	double
dict	map
list	list
bytearray	blob
aerospike.GeoJSON	GeoJSON

It is possible to nest these datatypes. For example a list may contain a dictionary, or a dictionary may contain a list as a value.

Note: Unless a user specified serializer has been provided, all other types will be stored as Python specific bytes. Python specific bytes may not be readable by Aerospike Clients for other languages.

1.3 Client Class — `Client`

1.3.1 `Client`

The client connects through a seed node (the address of a single node) to an Aerospike database cluster. From the seed node, the client learns of the other nodes and establishes connections to them. It also gets the partition map of the cluster, which is how it knows where every record actually lives.

The client handles the connections, including re-establishing them ahead of executing an operation. It keeps track of changes to the cluster through a cluster-tending thread.

See also:

Client Architecture and Data Distribution.

class aerospike.**Client**

Example:

```
from __future__ import print_function
# import the module
import aerospike
from aerospike import exception as ex
import sys

# Configure the client
config = {
    'hosts': [ ('127.0.0.1', 3000) ]
}

# Optionally set policies for various method types
write_policies = {'total_timeout': 2000, 'max_retries': 0}
read_policies = {'total_timeout': 1500, 'max_retries': 1}
policies = {'write': write_policies, 'read': read_policies}
config['policies'] = policies

# Create a client and connect it to the cluster
try:
    client = aerospike.client(config).connect()
except ex.ClientError as e:
    print("Error: {0} [{1}]".format(e.msg, e.code))
    sys.exit(1)

# Records are addressable via a tuple of (namespace, set, primary key)
key = ('test', 'demo', 'foo')

try:
    # Write a record
    client.put(key, {
        'name': 'John Doe',
```

```
        'age': 32
    })
except ex.RecordError as e:
    print("Error: {0} [{1}]".format(e.msg, e.code))

# Read a record
(key, meta, record) = client.get(key)

# Close the connection to the Aerospike cluster
client.close()
```

connect (*[username, password]*)
Connect to the cluster. The optional *username* and *password* only apply when connecting to the Enterprise Edition of Aerospike.

Parameters

- **username** (*str*) – a defined user with roles in the cluster. See *admin_create_user()*.
- **password** (*str*) – the password will be hashed by the client using bcrypt.

Raises *ClientError*, for example when a connection cannot be established to a seed node (any single node in the cluster from which the client learns of the other nodes).

See also:

Security features article.

is_connected()
Tests the connections between the client and the nodes of the cluster. If the result is `False`, the client will require another call to *connect()*.

Return type `bool`

Changed in version 2.0.0.

close()
Close all connections to the cluster. It is recommended to explicitly call this method when the program is done communicating with the cluster.

get (*key[, policy]*) -> *(key, meta, bins)*
Read a record with a given *key*, and return the record as a `tuple()` consisting of *key*, *meta* and *bins*.

Parameters

- **key** (*tuple*) – a *Key Tuple* associated with the record.
- **policy** (*dict*) – optional *Read Policies*.

Returns a *Record Tuple*. See *Unicode Handling*.

Raises *RecordNotFound*.

```
from __future__ import print_function
import aerospike
from aerospike import exception as ex
import sys

config = {'hosts': [('127.0.0.1', 3000)]}
client = aerospike.client(config).connect()

try:
```

```
    # assuming a record with such a key exists in the cluster
    key = ('test', 'demo', 1)
    (key, meta, bins) = client.get(key)
    print(key)
    print('---------------------------')
    print(meta)
    print('---------------------------')
    print(bins)
except ex.RecordNotFound:
    print("Record not found:", key)
except ex.AerospikeError as e:
    print("Error: {0} [{1}]".format(e.msg, e.code))
    sys.exit(1)
finally:
    client.close()
```

Warning: The client has been changed to raise a *RecordNotFound* exception when *get()* does not find the record. Code that used to check for `meta != None` should be modified.

Changed in version 2.0.0.

select (*key, bins[, policy]) -> (key, meta, bins*)

Read a record with a given *key*, and return the record as a `tuple()` consisting of *key*, *meta* and *bins*, with the specified bins projected. Prior to Aerospike server 3.6.0, if a selected bin does not exist its value will be `None`. Starting with 3.6.0, if a bin does not exist it will not be present in the returned *Record Tuple*.

Parameters

- **key** (*tuple*) – a *Key Tuple* associated with the record.
- **bins** (*list*) – a list of bin names to select from the record.
- **policy** (*dict*) – optional *Read Policies*.

Returns a *Record Tuple*. See *Unicode Handling*.

Raises *RecordNotFound*.

```
from __future__ import print_function
import aerospike
from aerospike import exception as ex
import sys

config = { 'hosts': [('127.0.0.1', 3000)] }
client = aerospike.client(config).connect()

try:
    # assuming a record with such a key exists in the cluster
    key = ('test', 'demo', 1)
    (key, meta, bins) = client.select(key, ['name'])
    print("name: ", bins.get('name'))
except ex.RecordNotFound:
    print("Record not found:", key)
except ex.AerospikeError as e:
    print("Error: {0} [{1}]".format(e.msg, e.code))
    sys.exit(1)
finally:
    client.close()
```

> **Warning:** The client has been changed to raise a *RecordNotFound* exception when *select()* does not find the record. Code that used to check for `meta != None` should be modified.

Changed in version 2.0.0.

exists (*key[, policy]) -> (key, meta*)
Check if a record with a given *key* exists in the cluster and return the record as a `tuple()` consisting of *key* and *meta*. If the record does not exist the *meta* data will be `None`.

Parameters

- **key** (*tuple*) – a *Key Tuple* associated with the record.
- **policy** (*dict*) – optional *Read Policies*.

Return type `tuple()` (key, meta)

Raises a subclass of *AerospikeError*.

```
from __future__ import print_function
import aerospike
from aerospike import exception as ex
import sys

config = { 'hosts': [('127.0.0.1', 3000)] }
client = aerospike.client(config).connect()

try:
    # assuming a record with such a key exists in the cluster
    key = ('test', 'demo', 1)
    (key, meta) = client.exists(key)
    print(key)
    print('---------------------------')
    print(meta)
except ex.RecordNotFound:
    print("Record not found:", key)
except ex.AerospikeError as e:
    print("Error: {0} [{1}]".format(e.msg, e.code))
    sys.exit(1)
finally:
    client.close()
```

Changed in version 2.0.3.

put (*key*, *bins*[, *meta*[, *policy*[, *serializer*]]])
Write a record with a given *key* to the cluster.

Parameters

- **key** (*tuple*) – a *Key Tuple* tuple associated with the record.
- **bins** (*dict*) – a `dict` of bin-name / bin-value pairs.
- **meta** (*dict*) – optional record metadata to be set, with field `'ttl'` set to `int` number of seconds or one of `aerospike.TTL_NAMESPACE_DEFAULT`, `aerospike.TTL_NEVER_EXPIRE`, `aerospike.TTL_DONT_UPDATE`
- **policy** (*dict*) – optional *Write Policies*.
- **serializer** – optionally override the serialization mode of the client with one of the *Serialization Constants*. To use a class-level user-defined serialization function registered

with *aerospike.set_serializer()* use *aerospike.SERIALIZER_USER*.

Raises a subclass of *AerospikeError*.

```
from __future__ import print_function
import aerospike
from aerospike import exception as ex

config = {
    'hosts': [ ('127.0.0.1', 3000) ],
    'total_timeout': 1500
}
client = aerospike.client(config).connect()
try:
    key = ('test', 'demo', 1)
    bins = {
        'l': [ "qwertyuiop", 1, bytearray("asd;as[d'as;d", "utf-8") ],
        'm': { "key": "asd';q;'1';" },
        'i': 1234,
        'f': 3.14159265359,
        's': '!@#@#$QSDAsd;as'
    }
    client.put(key, bins,
               policy={'key': aerospike.POLICY_KEY_SEND},
               meta={'ttl':180})
    # adding a bin
    client.put(key, {'smiley': u"\ud83d\ude04"})
    # removing a bin
    client.put(key, {'i': aerospike.null()})
except ex.AerospikeError as e:
    print("Error: {0} [{1}]".format(e.msg, e.code))
    sys.exit(1)
finally:
    client.close()
```

Note: Using Generation Policy

The generation policy allows a record to be written only when the generation is a specific value. In the following example, we only want to write the record if no change has occurred since *exists()* was called.

```
from __future__ import print_function
import aerospike
from aerospike import exception as ex
import sys

config = { 'hosts': [ ('127.0.0.1',3000)]}
client = aerospike.client(config).connect()

try:
    (key, meta) = client.exists(('test','test','key1'))
    print(meta)
    print('============')
    client.put(('test','test','key1'), {'id':1,'a':2},
        meta={'gen': 33},
        policy={'gen':aerospike.POLICY_GEN_EQ})
    print('Record written.')
except ex.RecordGenerationError:
```

```
    print("put() failed due to generation policy mismatch")
except ex.AerospikeError as e:
    print("Error: {0} [{1}]".format(e.msg, e.code))
client.close()
```

---

touch (*key*[, *val=0*[, *meta*[, *policy*]]])
Touch the given record, setting its time-to-live and incrementing its generation.

Parameters

- **key** (*tuple*) – a *Key Tuple* tuple associated with the record.
- **val** (*int*) – the optional ttl in seconds, with `0` resolving to the default value in the server config.
- **meta** (*dict*) – optional record metadata to be set.
- **policy** (*dict*) – optional *Operate Policies*.

Raises a subclass of *AerospikeError*.

See also:

Record TTL and Evictions and FAQ.

```
import aerospike

config = { 'hosts': [('127.0.0.1', 3000)] }
client = aerospike.client(config).connect()

key = ('test', 'demo', 1)
client.touch(key, 120, policy={'total_timeout': 100})
client.close()
```

remove (*key*[, *policy*])
Remove a record matching the *key* from the cluster.

Parameters

- **key** (*tuple*) – a *Key Tuple* associated with the record.
- **policy** (*dict*) – optional *Remove Policies*.

Raises a subclass of *AerospikeError*.

```
import aerospike

config = { 'hosts': [('127.0.0.1', 3000)] }
client = aerospike.client(config).connect()

key = ('test', 'demo', 1)
client.remove(key, {'retry': aerospike.POLICY_RETRY_ONCE
client.close()
```

get_key_digest (*ns*, *set*, *key*) → bytearray
Calculate the digest of a particular key. See: *Key Tuple*.

Parameters

- **ns** (*str*) – the namespace in the aerospike cluster.
- **set** (*str*) – the set name.

- **key** (str or int) – the primary key identifier of the record within the set.

Returns a RIPEMD-160 digest of the input tuple.

Return type bytearray

```
import aerospike
import pprint

pp = pprint.PrettyPrinter(indent=2)
config = { 'hosts': [('127.0.0.1', 3000)] }
client = aerospike.client(config).connect()

digest = client.get_key_digest("test", "demo", 1 )
pp.pprint(digest)
key = ('test', 'demo', None, digest)
(key, meta, bins) = client.get(key)
pp.pprint(bins)
client.close()
```

Deprecated since version 2.0.1: use the function *aerospike.calc_digest()* instead.

Bin Operations

remove_bin (*key*, *list*[, *meta*[, *policy*]])

Remove a list of bins from a record with a given *key*. Equivalent to setting those bins to *aerospike.null()* with a *put()*.

Parameters

- **key** (*tuple*) – a *Key Tuple* associated with the record.
- **list** (*list*) – the bins names to be removed from the record.
- **meta** (*dict*) – optional record metadata to be set, with field 'ttl' set to int number of seconds or one of aerospike.TTL_NAMESPACE_DEFAULT, aerospike.TTL_NEVER_EXPIRE, aerospike.TTL_DONT_UPDATE
- **policy** (*dict*) – optional *Write Policies*.

Raises a subclass of *AerospikeError*.

```
import aerospike

config = { 'hosts': [('127.0.0.1', 3000)] }
client = aerospike.client(config).connect()

key = ('test', 'demo', 1)
meta = { 'ttl': 3600 }
client.remove_bin(key, ['name', 'age'], meta, {'retry': aerospike.POLICY_
↪RETRY_ONCE})
client.close()
```

append (*key*, *bin*, *val*[, *meta*[, *policy*]])

Append the string *val* to the string value in *bin*.

Parameters

- **key** (*tuple*) – a *Key Tuple* tuple associated with the record.
- **bin** (*str*) – the name of the bin.

- **val** (*str*) – the string to append to the value of *bin*.
- **meta** (*dict*) – optional record metadata to be set, with field `'ttl'` set to `int` number of seconds or one of `aerospike.TTL_NAMESPACE_DEFAULT`, `aerospike.TTL_NEVER_EXPIRE`, `aerospike.TTL_DONT_UPDATE`
- **policy** (*dict*) – optional *Operate Policies*.

Raises a subclass of *AerospikeError*.

```
from __future__ import print_function
import aerospike
from aerospike import exception as ex
import sys

config = { 'hosts': [('127.0.0.1', 3000)] }
client = aerospike.client(config).connect()

try:
    key = ('test', 'demo', 1)
    client.append(key, 'name', ' jr.', policy={'total_timeout': 1200})
except ex.AerospikeError as e:
    print("Error: {0} [{1}]".format(e.msg, e.code))
    sys.exit(1)
finally:
    client.close()
```

prepend (*key*, *bin*, *val*[, *meta*[, *policy*]])

Prepend the string value in *bin* with the string *val*.

Parameters

- **key** (*tuple*) – a *Key Tuple* tuple associated with the record.
- **bin** (*str*) – the name of the bin.
- **val** (*str*) – the string to prepend to the value of *bin*.
- **meta** (*dict*) – optional record metadata to be set, with field `'ttl'` set to `int` number of seconds or one of `aerospike.TTL_NAMESPACE_DEFAULT`, `aerospike.TTL_NEVER_EXPIRE`, `aerospike.TTL_DONT_UPDATE`
- **policy** (*dict*) – optional *Operate Policies*.

Raises a subclass of *AerospikeError*.

```
from __future__ import print_function
import aerospike
from aerospike import exception as ex
import sys

config = { 'hosts': [('127.0.0.1', 3000)] }
client = aerospike.client(config).connect()

try:
    key = ('test', 'demo', 1)
    client.prepend(key, 'name', 'Dr. ', policy={'total_timeout': 1200})
except ex.AerospikeError as e:
    print("Error: {0} [{1}]".format(e.msg, e.code))
    sys.exit(1)
finally:
    client.close()
```

increment(*key*, *bin*, *offset*[, *meta*[, *policy*]])
Increment the integer value in *bin* by the integer *val*.

Parameters

- **key** (*tuple*) – a *Key Tuple* tuple associated with the record.
- **bin** (*str*) – the name of the bin.
- **offset** (int or float) – the value by which to increment the value in *bin*.
- **meta** (*dict*) – optional record metadata to be set, with field 'ttl' set to int number of seconds or one of aerospike.TTL_NAMESPACE_DEFAULT, aerospike.TTL_NEVER_EXPIRE, aerospike.TTL_DONT_UPDATE
- **policy** (*dict*) – optional *Operate Policies*.

Raises a subclass of *AerospikeError*.

```
from __future__ import print_function
import aerospike
from aerospike import exception as ex
import sys

config = { 'hosts': [('127.0.0.1', 3000)] }
client = aerospike.client(config).connect()

try:
    client.put(('test', 'cats', 'mr. peppy'), {'breed':'persian'}, policy={
↪'exists': aerospike.POLICY_EXISTS_CREATE_OR_REPLACE})
    (key, meta, bins) = client.get(('test', 'cats', 'mr. peppy'))
    print("Before:", bins, "\n")
    client.increment(key, 'lives', -1)
    (key, meta, bins) = client.get(key)
    print("After:", bins, "\n")
    client.increment(key, 'lives', -1)
    (key, meta, bins) = client.get(key)
    print("Poor Kitty:", bins, "\n")
    print(bins)
except ex.AerospikeError as e:
    print("Error: {0} [{1}]".format(e.msg, e.code))
    sys.exit(1)
finally:
    client.close()
```

list_append(*key*, *bin*, *val*[, *meta*[, *policy*]])
Append a single element to a list value in *bin*.

Parameters

- **key** (*tuple*) – a *Key Tuple* tuple associated with the record.
- **bin** (*str*) – the name of the bin.
- **val** – int, str, float, bytearray, list, dict. An unsupported type will be serialized.
- **meta** (*dict*) – optional record metadata to be set, with field 'ttl' set to int number of seconds or one of aerospike.TTL_NAMESPACE_DEFAULT, aerospike.TTL_NEVER_EXPIRE, aerospike.TTL_DONT_UPDATE
- **policy** (*dict*) – optional *Operate Policies*.

Raises a subclass of *AerospikeError*.

Note: Requires server version >= 3.7.0

New in version 1.0.59.

list_extend(*key*, *bin*, *items*[, *meta*[, *policy*]])
Extend the list value in *bin* with the given *items*.

Parameters

- **key** (*tuple*) – a *Key Tuple* tuple associated with the record.
- **bin** (*str*) – the name of the bin.
- **items** (*list*) – the items to append the list in *bin*.
- **meta** (*dict*) – optional record metadata to be set, with field `'ttl'` set to `int` number of seconds or one of `aerospike.TTL_NAMESPACE_DEFAULT`, `aerospike.TTL_NEVER_EXPIRE`, `aerospike.TTL_DONT_UPDATE`
- **policy** (*dict*) – optional *Operate Policies*.

Raises a subclass of *AerospikeError*.

Note: Requires server version >= 3.7.0

New in version 1.0.59.

list_insert(*key*, *bin*, *index*, *val*[, *meta*[, *policy*]])
Insert an element at the specified *index* of a list value in *bin*.

Parameters

- **key** (*tuple*) – a *Key Tuple* tuple associated with the record.
- **bin** (*str*) – the name of the bin.
- **index** (*int*) – the position in the index where the value should be inserted.
- **val** – `int`, `str`, `float`, `bytearray`, `list`, `dict`. An unsupported type will be serialized.
- **meta** (*dict*) – optional record metadata to be set, with field `'ttl'` set to `int` number of seconds or one of `aerospike.TTL_NAMESPACE_DEFAULT`, `aerospike.TTL_NEVER_EXPIRE`, `aerospike.TTL_DONT_UPDATE`
- **policy** (*dict*) – optional *Operate Policies*.

Raises a subclass of *AerospikeError*.

Note: Requires server version >= 3.7.0

New in version 1.0.59.

list_insert_items(*key*, *bin*, *index*, *items*[, *meta*[, *policy*]])
Insert the *items* at the specified *index* of a list value in *bin*.

Parameters

- **key** (*tuple*) – a *Key Tuple* tuple associated with the record.

- **bin** (*str*) – the name of the bin.
- **index** (*int*) – the position in the index where the items should be inserted.
- **items** (*list*) – the items to insert into the list in *bin*.
- **meta** (*dict*) – optional record metadata to be set, with field `'ttl'` set to `int` number of seconds or one of `aerospike.TTL_NAMESPACE_DEFAULT`, `aerospike.TTL_NEVER_EXPIRE`, `aerospike.TTL_DONT_UPDATE`
- **policy** (*dict*) – optional *Operate Policies*.

Raises a subclass of *AerospikeError*.

Note: Requires server version >= 3.7.0

New in version 1.0.59.

list_pop(*key*, *bin*, *index*[, *meta*[, *policy*]]) → val

Remove and get back a list element at a given *index* of a list value in *bin*.

Parameters

- **key** (*tuple*) – a *Key Tuple* tuple associated with the record.
- **bin** (*str*) – the name of the bin.
- **index** (*int*) – the index position in the list element which should be removed and returned.
- **meta** (*dict*) – optional record metadata to be set, with field `'ttl'` set to `int` number of seconds or one of `aerospike.TTL_NAMESPACE_DEFAULT`, `aerospike.TTL_NEVER_EXPIRE`, `aerospike.TTL_DONT_UPDATE`
- **policy** (*dict*) – optional *Operate Policies*.

Returns a single list element.

Raises a subclass of *AerospikeError*.

Note: Requires server version >= 3.7.0

New in version 1.0.59.

list_pop_range(*key*, *bin*, *index*, *count*[, *meta*[, *policy*]]) → val

Remove and get back list elements at a given *index* of a list value in *bin*.

Parameters

- **key** (*tuple*) – a *Key Tuple* tuple associated with the record.
- **bin** (*str*) – the name of the bin.
- **index** (*int*) – the index of first element in a range which should be removed and returned.
- **count** (*int*) – the number of elements in the range.
- **meta** (*dict*) – optional record metadata to be set, with field `'ttl'` set to `int` number of seconds or one of `aerospike.TTL_NAMESPACE_DEFAULT`, `aerospike.TTL_NEVER_EXPIRE`, `aerospike.TTL_DONT_UPDATE`
- **policy** (*dict*) – optional *Operate Policies*.

Returns a `list` of elements.

Raises a subclass of *AerospikeError*.

Note: Requires server version >= 3.7.0

New in version 1.0.59.

list_remove(*key*, *bin*, *index*[, *meta*[, *policy*]])
Remove a list element at a given *index* of a list value in *bin*.

Parameters

- **key** (*tuple*) – a *Key Tuple* tuple associated with the record.
- **bin** (*str*) – the name of the bin.
- **index** (*int*) – the index position in the list element which should be removed.
- **meta** (*dict*) – optional record metadata to be set, with field `'ttl'` set to `int` number of seconds or one of `aerospike.TTL_NAMESPACE_DEFAULT`, `aerospike.TTL_NEVER_EXPIRE`, `aerospike.TTL_DONT_UPDATE`
- **policy** (*dict*) – optional *Operate Policies*.

Raises a subclass of *AerospikeError*.

Note: Requires server version >= 3.7.0

New in version 1.0.59.

list_remove_range(*key*, *bin*, *index*, *count*[, *meta*[, *policy*]])
Remove list elements at a given *index* of a list value in *bin*.

Parameters

- **key** (*tuple*) – a *Key Tuple* tuple associated with the record.
- **bin** (*str*) – the name of the bin.
- **index** (*int*) – the index of first element in a range which should be removed.
- **count** (*int*) – the number of elements in the range.
- **meta** (*dict*) – optional record metadata to be set, with field `'ttl'` set to `int` number of seconds or one of `aerospike.TTL_NAMESPACE_DEFAULT`, `aerospike.TTL_NEVER_EXPIRE`, `aerospike.TTL_DONT_UPDATE`
- **policy** (*dict*) – optional *Operate Policies*.

Raises a subclass of *AerospikeError*.

Note: Requires server version >= 3.7.0

New in version 1.0.59.

list_clear(*key*, *bin*[, *meta*[, *policy*]])
Remove all the elements from a list value in *bin*.

Parameters

- **key** (*tuple*) – a *Key Tuple* tuple associated with the record.
- **bin** (*str*) – the name of the bin.
- **meta** (*dict*) – optional record metadata to be set, with field `'ttl'` set to `int` number of seconds or one of `aerospike.TTL_NAMESPACE_DEFAULT`, `aerospike.TTL_NEVER_EXPIRE`, `aerospike.TTL_DONT_UPDATE`
- **policy** (*dict*) – optional *Operate Policies*.

Raises a subclass of *AerospikeError*.

Note: Requires server version >= 3.7.0

New in version 1.0.59.

list_set (*key*, *bin*, *index*, *val*[, *meta*[, *policy*]])
Set list element *val* at the specified *index* of a list value in *bin*.

Parameters

- **key** (*tuple*) – a *Key Tuple* tuple associated with the record.
- **bin** (*str*) – the name of the bin.
- **index** (*int*) – the position in the index where the value should be set.
- **val** – `int`, `str`, `float`, `bytearray`, `list`, `dict`. An unsupported type will be serialized.
- **meta** (*dict*) – optional record metadata to be set, with field `'ttl'` set to `int` number of seconds or one of `aerospike.TTL_NAMESPACE_DEFAULT`, `aerospike.TTL_NEVER_EXPIRE`, `aerospike.TTL_DONT_UPDATE`
- **policy** (*dict*) – optional *Operate Policies*.

Raises a subclass of *AerospikeError*.

Note: Requires server version >= 3.7.0

New in version 1.0.59.

list_get (*key*, *bin*, *index*[, *meta*[, *policy*]]) → val
Get the list element at the specified *index* of a list value in *bin*.

Parameters

- **key** (*tuple*) – a *Key Tuple* tuple associated with the record.
- **bin** (*str*) – the name of the bin.
- **index** (*int*) – the position in the index where the value should be set.
- **meta** (*dict*) – unused for this operation
- **policy** (*dict*) – optional *Operate Policies*.

Raises a subclass of *AerospikeError*.

Returns the list elements at the given index.

Note: Requires server version >= 3.7.0

New in version 1.0.59.

list_get_range(*key*, *bin*, *index*, *count*[, *meta*[, *policy*]]) → val
Get the list of *count* elements starting at a specified *index* of a list value in *bin*.

Parameters

- **key** (*tuple*) – a *Key Tuple* tuple associated with the record.
- **bin** (*str*) – the name of the bin.
- **index** (*int*) – the position in the index where the value should be set.
- **meta** (*dict*) – unused for this operation
- **policy** (*dict*) – optional *Operate Policies*.

Raises a subclass of *AerospikeError*.

Returns a `list` of elements.

Note: Requires server version >= 3.7.0

New in version 1.0.59.

list_trim(*key*, *bin*, *index*, *count*[, *meta*[, *policy*]]) → val
Remove elements from the list which are not within the range starting at the given *index* plus *count*.

Parameters

- **key** (*tuple*) – a *Key Tuple* tuple associated with the record.
- **bin** (*str*) – the name of the bin.
- **index** (*int*) – the position in the index marking the start of the range.
- **index** – the index position of the first element in a range which should not be removed.
- **count** (*int*) – the number of elements in the range.
- **meta** (*dict*) – optional record metadata to be set, with field `'ttl'` set to `int` number of seconds or one of `aerospike.TTL_NAMESPACE_DEFAULT`, `aerospike.TTL_NEVER_EXPIRE`, `aerospike.TTL_DONT_UPDATE`
- **policy** (*dict*) – optional *Operate Policies*.

Raises a subclass of *AerospikeError*.

Returns a `list` of elements.

Note: Requires server version >= 3.7.0

New in version 1.0.59.

list_size(*key*, *bin*[, *meta*[, *policy*]]) → count
Count the number of elements in the list value in *bin*.

Parameters

- **key** (*tuple*) – a *Key Tuple* tuple associated with the record.

- **bin** (*str*) – the name of the bin.
- **meta** (*dict*) – unused for this operation
- **policy** (*dict*) – optional *Operate Policies*.

Raises a subclass of *AerospikeError*.

Returns a int.

Note: Requires server version >= 3.7.0

New in version 1.0.59.

map_set_policy(*key*, *bin*, *map_policy*)
Set the map policy for the given *bin*.

Parameters

- **key** (*tuple*) – a *Key Tuple* tuple associated with the record.
- **bin** (*str*) – the name of the bin.
- **map_policy** (*dict*) – *Map Policies*.

Raises a subclass of *AerospikeError*.

Note: Requires server version >= 3.8.4

New in version 2.0.4.

map_put(*key*, *bin*, *map_key*, *val*[, *map_policy*[, *meta*[, *policy*]]])
Add the given *map_key*/*value* pair to the map record specified by *key* and *bin*.

Parameters

- **key** (*tuple*) – a *Key Tuple* tuple associated with the record.
- **bin** (*str*) – the name of the bin.
- **map_key** – int, str, float, bytearray. An unsupported type will be serialized.
- **val** – int, str, float, bytearray, list, dict. An unsupported type will be serialized.
- **map_policy** (*dict*) – optional *Map Policies*.
- **meta** (*dict*) – optional record metadata to be set, with field 'ttl' set to int number of seconds or one of aerospike.TTL_NAMESPACE_DEFAULT, aerospike.TTL_NEVER_EXPIRE, aerospike.TTL_DONT_UPDATE
- **policy** (*dict*) – optional *Operate Policies*.

Raises a subclass of *AerospikeError*.

Note: Requires server version >= 3.8.4

New in version 2.0.4.

map_put_items(*key*, *bin*, *items*[, *map_policy*[, *meta*[, *policy*]]])
Add the given *items* dict of key/value pairs to the map record specified by *key* and *bin*.

Parameters

- **key** (*tuple*) – a *Key Tuple* tuple associated with the record.
- **bin** (*str*) – the name of the bin.
- **items** (*dict*) – key/value pairs.
- **map_policy** (*dict*) – optional *Map Policies*.
- **meta** (*dict*) – optional record metadata to be set, with field `'ttl'` set to `int` number of seconds or one of `aerospike.TTL_NAMESPACE_DEFAULT`, `aerospike.TTL_NEVER_EXPIRE`, `aerospike.TTL_DONT_UPDATE`
- **policy** (*dict*) – optional *Operate Policies*.

Raises a subclass of *AerospikeError*.

Note: Requires server version >= 3.8.4

New in version 2.0.4.

map_increment(*key*, *bin*, *map_key*, *incr*[, *map_policy*[, *meta*[, *policy*]]])
Increment the value of the map entry by given *incr*. Map entry is specified by *key*, *bin* and *map_key*.

Parameters

- **key** (*tuple*) – a *Key Tuple* tuple associated with the record.
- **bin** (*str*) – the name of the bin.
- **map_key** – `int`, `str`, `float`, `bytearray`. An unsupported type will be serialized.
- **incr** – `int` or `float`
- **map_policy** (*dict*) – optional *Map Policies*.
- **meta** (*dict*) – optional record metadata to be set, with field `'ttl'` set to `int` number of seconds or one of `aerospike.TTL_NAMESPACE_DEFAULT`, `aerospike.TTL_NEVER_EXPIRE`, `aerospike.TTL_DONT_UPDATE`
- **policy** (*dict*) – optional *Operate Policies*.

Raises a subclass of *AerospikeError*.

Note: Requires server version >= 3.8.4

New in version 2.0.4.

map_decrement(*key*, *bin*, *map_key*, *decr*[, *map_policy*[, *meta*[, *policy*]]])
Decrement the value of the map entry by given *decr*. Map entry is specified by *key*, *bin* and *map_key*.

Parameters

- **key** (*tuple*) – a *Key Tuple* tuple associated with the record.
- **bin** (*str*) – the name of the bin.
- **map_key** – `int`, `str`, `float`, `bytearray`. An unsupported type will be serialized.
- **decr** – `int` or `float`
- **map_policy** (*dict*) – optional *Map Policies*.

- **meta** (*dict*) – optional record metadata to be set, with field `'ttl'` set to `int` number of seconds or one of `aerospike.TTL_NAMESPACE_DEFAULT`, `aerospike.TTL_NEVER_EXPIRE`, `aerospike.TTL_DONT_UPDATE`
- **policy** (*dict*) – optional *Operate Policies*.

Raises a subclass of *AerospikeError*.

Note: Requires server version >= 3.8.4

New in version 2.0.4.

map_size(*key*, *bin*[, *meta*[, *policy*]]) → count
Return the size of the map specified by *key* and *bin*.

Parameters

- **key** (*tuple*) – a *Key Tuple* tuple associated with the record.
- **bin** (*str*) – the name of the bin.
- **meta** (*dict*) – unused for this operation
- **policy** (*dict*) – optional *Operate Policies*.

Raises a subclass of *AerospikeError*.

Returns a `int`.

Note: Requires server version >= 3.8.4

New in version 2.0.4.

map_clear(*key*, *bin*[, *meta*[, *policy*]])
Remove all entries from the map specified by *key* and *bin*.

Parameters

- **key** (*tuple*) – a *Key Tuple* tuple associated with the record.
- **bin** (*str*) – the name of the bin.
- **meta** (*dict*) – optional record metadata to be set, with field `'ttl'` set to `int` number of seconds or one of `aerospike.TTL_NAMESPACE_DEFAULT`, `aerospike.TTL_NEVER_EXPIRE`, `aerospike.TTL_DONT_UPDATE`
- **policy** (*dict*) – optional *Operate Policies*.

Raises a subclass of *AerospikeError*.

Note: Requires server version >= 3.8.4

New in version 2.0.4.

map_remove_by_key(*key*, *bin*, *map_key*, *return_type*[, *meta*[, *policy*]])
Remove and optionally return first map entry from the map specified by *key* and *bin* which matches given *map_key*.

Parameters

- **key** (*tuple*) – a *Key Tuple* tuple associated with the record.

- **bin** (*str*) – the name of the bin.
- **map_key** – int, str, float, bytearray. An unsupported type will be serialized.
- **return_type** – int *Map Return Types*
- **meta** (*dict*) – optional record metadata to be set, with field 'ttl' set to int number of seconds or one of aerospike.TTL_NAMESPACE_DEFAULT, aerospike.TTL_NEVER_EXPIRE, aerospike.TTL_DONT_UPDATE
- **policy** (*dict*) – optional *Operate Policies*.

Raises a subclass of *AerospikeError*.

Returns depends on return_type parameter

Note: Requires server version >= 3.8.4

New in version 2.0.4.

map_remove_by_key_list(*key, bin, list, return_type[, meta[, policy]][, meta[, policy]]*)
Remove and optionally return map entries from the map specified by *key* and *bin* which have keys that match the given *list* of keys.

Parameters

- **key** (*tuple*) – a *Key Tuple* tuple associated with the record.
- **bin** (*str*) – the name of the bin.
- **list** – list the list of keys to match
- **return_type** – int *Map Return Types*
- **meta** (*dict*) – optional record metadata to be set, with field 'ttl' set to int number of seconds or one of aerospike.TTL_NAMESPACE_DEFAULT, aerospike.TTL_NEVER_EXPIRE, aerospike.TTL_DONT_UPDATE
- **policy** (*dict*) – optional *Operate Policies*.

Raises a subclass of *AerospikeError*.

Returns depends on return_type parameter

Note: Requires server version >= 3.8.4

New in version 2.0.4.

map_remove_by_key_range(*key*, *bin*, *map_key*, *range*, *return_type*[, *meta*[, *policy*]])
Remove and optionally return map entries from the map specified by *key* and *bin* identified by the key range (*map_key* inclusive, *range* exclusive).

Parameters

- **key** (*tuple*) – a *Key Tuple* tuple associated with the record.
- **bin** (*str*) – the name of the bin.
- **map_key** – int, str, float, bytearray. An unsupported type will be serialized.
- **range** – int, str, float, bytearray. An unsupported type will be serialized.
- **return_type** – int *Map Return Types*

- **meta** (*dict*) – optional record metadata to be set, with field `'ttl'` set to `int` number of seconds or one of `aerospike.TTL_NAMESPACE_DEFAULT`, `aerospike.TTL_NEVER_EXPIRE`, `aerospike.TTL_DONT_UPDATE`
- **policy** (*dict*) – optional *Operate Policies*.

Raises a subclass of *AerospikeError*.

Returns depends on return_type parameter

Note: Requires server version >= 3.8.4

New in version 2.0.4.

map_remove_by_value (*key*, *bin*, *val*, *return_type*[, *meta*[, *policy*]])
Remove and optionally return map entries from the map specified by *key* and *bin* which have a value matching *val* parameter.

Parameters

- **key** (*tuple*) – a *Key Tuple* tuple associated with the record.
- **bin** (*str*) – the name of the bin.
- **val** – `int`, `str`, `float`, `bytearray`. An unsupported type will be serialized.
- **return_type** – `int` *Map Return Types*
- **meta** (*dict*) – optional record metadata to be set, with field `'ttl'` set to `int` number of seconds or one of `aerospike.TTL_NAMESPACE_DEFAULT`, `aerospike.TTL_NEVER_EXPIRE`, `aerospike.TTL_DONT_UPDATE`
- **policy** (*dict*) – optional *Operate Policies*.

Raises a subclass of *AerospikeError*.

Returns depends on return_type parameter

Note: Requires server version >= 3.8.4

New in version 2.0.4.

map_remove_by_value_list (*key*, *bin*, *list*, *return_type*[, *meta*[, *policy*]])
Remove and optionally return map entries from the map specified by *key* and *bin* which have a value matching the *list* of values.

Parameters

- **key** (*tuple*) – a *Key Tuple* tuple associated with the record.
- **bin** (*str*) – the name of the bin.
- **list** – `list` the list of values to match
- **return_type** – `int` *Map Return Types*
- **meta** (*dict*) – optional record metadata to be set, with field `'ttl'` set to `int` number of seconds or one of `aerospike.TTL_NAMESPACE_DEFAULT`, `aerospike.TTL_NEVER_EXPIRE`, `aerospike.TTL_DONT_UPDATE`
- **policy** (*dict*) – optional *Operate Policies*.

Raises a subclass of *AerospikeError*.

Returns depends on return_type parameter

Note: Requires server version >= 3.8.4

New in version 2.0.4.

map_remove_by_value_range(*key*, *bin*, *val*, *range*, *return_type*[, *meta*[, *policy*]])
Remove and optionally return map entries from the map specified by *key* and *bin* identified by the value range (*val* inclusive, *range* exclusive).

Parameters

- **key** (*tuple*) – a *Key Tuple* tuple associated with the record.
- **bin** (*str*) – the name of the bin.
- **val** – int, str, float, bytearray. An unsupported type will be serialized.
- **range** – int, str, float, bytearray. An unsupported type will be serialized.
- **return_type** – int *Map Return Types*
- **meta** (*dict*) – optional record metadata to be set, with field 'ttl' set to int number of seconds or one of aerospike.TTL_NAMESPACE_DEFAULT, aerospike.TTL_NEVER_EXPIRE, aerospike.TTL_DONT_UPDATE
- **policy** (*dict*) – optional *Operate Policies*.

Raises a subclass of *AerospikeError*.

Returns depends on return_type parameter

Note: Requires server version >= 3.8.4

New in version 2.0.4.

map_remove_by_index(*key*, *bin*, *index*, *return_type*[, *meta*[, *policy*]])
Remove and optionally return the map entry from the map specified by *key* and *bin* at the given *index* location.

Parameters

- **key** (*tuple*) – a *Key Tuple* tuple associated with the record.
- **bin** (*str*) – the name of the bin.
- **index** – int the index location of the map entry
- **return_type** – int *Map Return Types*
- **meta** (*dict*) – optional record metadata to be set, with field 'ttl' set to int number of seconds or one of aerospike.TTL_NAMESPACE_DEFAULT, aerospike.TTL_NEVER_EXPIRE, aerospike.TTL_DONT_UPDATE
- **policy** (*dict*) – optional *Operate Policies*.

Raises a subclass of *AerospikeError*.

Returns depends on return_type parameter

Note: Requires server version >= 3.8.4

New in version 2.0.4.

map_remove_by_index_range(*key*, *bin*, *index*, *range*, *return_type*[, *meta*[, *policy*]])
Remove and optionally return the map entries from the map specified by *key* and *bin* starting at the given *index* location and removing *range* number of items.

Parameters

- **key** (*tuple*) – a *Key Tuple* tuple associated with the record.
- **bin** (*str*) – the name of the bin.
- **index** – int the index location of the first map entry to remove
- **range** – int the number of items to remove from the map
- **return_type** – int *Map Return Types*
- **meta** (*dict*) – optional record metadata to be set, with field 'ttl' set to int number of seconds or one of aerospike.TTL_NAMESPACE_DEFAULT, aerospike.TTL_NEVER_EXPIRE, aerospike.TTL_DONT_UPDATE
- **policy** (*dict*) – optional *Operate Policies*.

Raises a subclass of *AerospikeError*.

Returns depends on return_type parameter

Note: Requires server version >= 3.8.4

New in version 2.0.4.

map_remove_by_rank(*key*, *bin*, *rank*, *return_type*[, *meta*[, *policy*]])
Remove and optionally return the map entry from the map specified by *key* and *bin* with a value that has the given *rank*.

Parameters

- **key** (*tuple*) – a *Key Tuple* tuple associated with the record.
- **bin** (*str*) – the name of the bin.
- **rank** – int the rank of the value of the entry in the map
- **return_type** – int *Map Return Types*
- **meta** (*dict*) – optional record metadata to be set, with field 'ttl' set to int number of seconds or one of aerospike.TTL_NAMESPACE_DEFAULT, aerospike.TTL_NEVER_EXPIRE, aerospike.TTL_DONT_UPDATE
- **policy** (*dict*) – optional *Operate Policies*.

Raises a subclass of *AerospikeError*.

Returns depends on return_type parameter

Note: Requires server version >= 3.8.4

New in version 2.0.4.

map_remove_by_rank_range(*key*, *bin*, *rank*, *range*, *return_type*[, *meta*[, *policy*]])
Remove and optionally return the map entries from the map specified by *key* and *bin* which have a value rank starting at *rank* and removing *range* number of items.

Parameters

- **key** (*tuple*) – a *Key Tuple* tuple associated with the record.
- **bin** (*str*) – the name of the bin.
- **rank** – int the rank of the value of the first map entry to remove
- **range** – int the number of items to remove from the map
- **return_type** – int *Map Return Types*
- **meta** (*dict*) – optional record metadata to be set, with field 'ttl' set to int number of seconds or one of aerospike.TTL_NAMESPACE_DEFAULT, aerospike.TTL_NEVER_EXPIRE, aerospike.TTL_DONT_UPDATE
- **policy** (*dict*) – optional *Operate Policies*.

Raises a subclass of *AerospikeError*.

Returns depends on return_type parameter

Note: Requires server version >= 3.8.4

New in version 2.0.4.

map_get_by_key(*key*, *bin*, *map_key*, *return_type*[, *meta*[, *policy*]])
Return map entry from the map specified by *key* and *bin* which has a key that matches the given *map_key*.

Parameters

- **key** (*tuple*) – a *Key Tuple* tuple associated with the record.
- **bin** (*str*) – the name of the bin.
- **map_key** – int, str, float, bytearray. An unsupported type will be serialized.
- **return_type** – int *Map Return Types*
- **meta** (*dict*) – unused for this operation
- **policy** (*dict*) – optional *Operate Policies*.

Raises a subclass of *AerospikeError*.

Returns depends on return_type parameter

Note: Requires server version >= 3.8.4

New in version 2.0.4.

map_get_by_key_range(*key*, *bin*, *map_key*, *range*, *return_type*[, *meta*[, *policy*]])
Return map entries from the map specified by *key* and *bin* identified by the key range (*map_key* inclusive, *range* exclusive).

Parameters

- **key** (*tuple*) – a *Key Tuple* tuple associated with the record.
- **bin** (*str*) – the name of the bin.
- **map_key** – int, str, float, bytearray. An unsupported type will be serialized.
- **range** – int, str, float, bytearray. An unsupported type will be serialized.

- **return_type** – int *Map Return Types*
- **meta** (*dict*) – unused for this operation
- **policy** (*dict*) – optional *Operate Policies*.

Raises a subclass of *AerospikeError*.

Returns depends on return_type parameter

Note: Requires server version >= 3.8.4

New in version 2.0.4.

map_get_by_value(*key*, *bin*, *val*, *return_type*[, *meta*[, *policy*]])
Return map entries from the map specified by *key* and *bin* which have a value matching *val* parameter.

Parameters

- **key** (*tuple*) – a *Key Tuple* tuple associated with the record.
- **bin** (*str*) – the name of the bin.
- **val** – int, str, float, bytearray. An unsupported type will be serialized.
- **return_type** – int *Map Return Types*
- **meta** (*dict*) – unused for this operation
- **policy** (*dict*) – optional *Operate Policies*.

Raises a subclass of *AerospikeError*.

Returns depends on return_type parameter

Note: Requires server version >= 3.8.4

New in version 2.0.4.

map_get_by_value_range(*key*, *bin*, *val*, *range*, *return_type*[, *meta*[, *policy*]])
Return map entries from the map specified by *key* and *bin* identified by the value range (*val* inclusive, *range* exclusive).

Parameters

- **key** (*tuple*) – a *Key Tuple* tuple associated with the record.
- **bin** (*str*) – the name of the bin.
- **val** – int, str, float, bytearray. An unsupported type will be serialized.
- **range** – int, str, float, bytearray. An unsupported type will be serialized.
- **return_type** – int *Map Return Types*
- **meta** (*dict*) – unused for this operation
- **policy** (*dict*) – optional *Operate Policies*.

Raises a subclass of *AerospikeError*.

Returns depends on return_type parameter

Note: Requires server version >= 3.8.4

New in version 2.0.4.

map_get_by_index (*key*, *bin*, *index*, *return_type*[, *meta*[, *policy*]])
Return the map entry from the map specified by *key* and *bin* at the given *index* location.

Parameters

- **key** (*tuple*) – a *Key Tuple* tuple associated with the record.
- **bin** (*str*) – the name of the bin.
- **index** – int the index location of the map entry
- **return_type** – int *Map Return Types*
- **meta** (*dict*) – unused for this operation
- **policy** (*dict*) – optional *Operate Policies*.

Raises a subclass of *AerospikeError*.

Returns depends on return_type parameter

Note: Requires server version >= 3.8.4

New in version 2.0.4.

map_get_by_index_range (*key*, *bin*, *index*, *range*, *return_type*[, *meta*[, *policy*]])
Return the map entries from the map specified by *key* and *bin* starting at the given *index* location and removing *range* number of items.

Parameters

- **key** (*tuple*) – a *Key Tuple* tuple associated with the record.
- **bin** (*str*) – the name of the bin.
- **index** – int the index location of the first map entry to remove
- **range** – int the number of items to remove from the map
- **return_type** – int *Map Return Types*
- **meta** (*dict*) – unused for this operation
- **policy** (*dict*) – optional *Operate Policies*.

Raises a subclass of *AerospikeError*.

Returns depends on return_type parameter

Note: Requires server version >= 3.8.4

New in version 2.0.4.

map_get_by_rank (*key*, *bin*, *rank*, *return_type*[, *meta*[, *policy*]])
Return the map entry from the map specified by *key* and *bin* with a value that has the given *rank*.

Parameters

- **key** (*tuple*) – a *Key Tuple* tuple associated with the record.
- **bin** (*str*) – the name of the bin.
- **rank** – int the rank of the value of the entry in the map
- **return_type** – int *Map Return Types*
- **meta** (*dict*) – unused for this operation
- **policy** (*dict*) – optional *Operate Policies*.

Raises a subclass of *AerospikeError*.

Returns depends on return_type parameter

Note: Requires server version >= 3.8.4

New in version 2.0.4.

map_get_by_rank_range(*key*, *bin*, *rank*, *range*, *return_type*[, *meta*[, *policy*]])
Return the map entries from the map specified by *key* and *bin* which have a value rank starting at *rank* and removing *range* number of items.

Parameters

- **key** (*tuple*) – a *Key Tuple* tuple associated with the record.
- **bin** (*str*) – the name of the bin.
- **rank** – int the rank of the value of the first map entry to remove
- **range** – int the number of items to remove from the map
- **return_type** – int *Map Return Types*
- **meta** (*dict*) – unused for this operation
- **policy** (*dict*) – optional *Operate Policies*.

Raises a subclass of *AerospikeError*.

Returns depends on return_type parameter

Note: Requires server version >= 3.8.4

New in version 2.0.4.

operate(*key, list[, meta[, policy]]*) -> *(key, meta, bins)*
Perform multiple bin operations on a record with a given *key*, In Aerospike server versions prior to 3.6.0, non-existent bins being read will have a None value. Starting with 3.6.0 non-existent bins will not be present in the returned *Record Tuple*. The returned record tuple will only contain one entry per bin, even if multiple operations were performed on the bin.

Parameters

- **key** (*tuple*) – a *Key Tuple* associated with the record.
- **list** (*list*) – a list of one or more bin operations, each structured as the dict `{'bin': bin name, 'op': aerospike.OPERATOR_* [, 'val': value]}`. See *Operators*.

- **meta** (*dict*) – optional record metadata to be set, with field `'ttl'` set to `int` number of seconds or one of `aerospike.TTL_NAMESPACE_DEFAULT`, `aerospike.TTL_NEVER_EXPIRE`, `aerospike.TTL_DONT_UPDATE`
- **policy** (*dict*) – optional *Operate Policies*.

Returns a *Record Tuple*. See *Unicode Handling*.

Raises a subclass of *AerospikeError*.

Note: In version *2.1.3* the return format of certain bin entries for this method, **only in cases when a map operation specifying a 'return_type' is used**, has changed. Bin entries for map operations using "return_type" of aerospike.MAP_RETURN_KEY_VALUE will now return a bin value of a list of keys and corresponding values, rather than a list of key/value tuples. See the following code block for details.

```
# pre 2.1.3 formatting of key/value bin value
[('key1', 'val1'), ('key2', 'val2'), ('key3', 'val3')]

# >= 2.1.3 formatting
['key1', 'val1', 'key2', 'val2', 'key3', 'val3']
```

Note: *operate()* can now have multiple write operations on a single bin.

```
from __future__ import print_function
import aerospike
from aerospike import exception as ex
import sys

config = { 'hosts': [('127.0.0.1', 3000)] }
client = aerospike.client(config).connect()

try:
    key = ('test', 'demo', 1)
    client.put(key, {'age': 25, 'career': 'delivery boy'})
    ops = [
        {
          "op" : aerospike.OPERATOR_INCR,
          "bin": "age",
          "val": 1000
        },
        {
          "op" : aerospike.OPERATOR_WRITE,
          "bin": "name",
          "val": "J."
        },
        {
          "op" : aerospike.OPERATOR_PREPEND,
          "bin": "name",
          "val": "Phillip "
        },
        {
          "op" : aerospike.OPERATOR_APPEND,
          "bin": "name",
          "val": " Fry"
        },
```

```
        {
          "op" : aerospike.OPERATOR_READ,
          "bin": "name"
        },
        {
          "op" : aerospike.OPERATOR_READ,
          "bin": "career"
        }
    ]
    (key, meta, bins) = client.operate(key, ops, {'ttl':360}, {'total_timeout
↪':500})

    print(key)
    print('---------------------------')
    print(meta)
    print('---------------------------')
    print(bins) # will display all bins selected by OPERATOR_READ operations
except ex.AerospikeError as e:
    print("Error: {0} [{1}]".format(e.msg, e.code))
    sys.exit(1)
finally:
    client.close()
```

Note: *OPERATOR_TOUCH* should only ever combine with *OPERATOR_READ*, for example to implement LRU expiry on the records of a set.

Warning: Having *val* associated with *OPERATOR_TOUCH* is deprecated. Use the meta *ttl* field instead.

```
from __future__ import print_function
import aerospike
from aerospike import exception as ex
import sys

config = { 'hosts': [('127.0.0.1', 3000)] }
client = aerospike.client(config).connect()

try:
    key = ('test', 'demo', 1)
    ops = [
        {
          "op" : aerospike.OPERATOR_TOUCH,
        },
        {
          "op" : aerospike.OPERATOR_READ,
          "bin": "name"
        }
    ]
    (key, meta, bins) = client.operate(key, ops, {'ttl':1800})
    print("Touched the record for {0}, extending its ttl by 30m".format(bins))
except ex.AerospikeError as e:
    print("Error: {0} [{1}]".format(e.msg, e.code))
    sys.exit(1)
finally:
```

```
client.close()
```

Changed in version 2.1.3.

operate_ordered (*key, list[, meta[, policy]]) -> (key, meta, bins*)

Perform multiple bin operations on a record with the results being returned as a list of (bin-name, result) tuples. The order of the elements in the list will correspond to the order of the operations from the input parameters.

param tuple key a *Key Tuple* associated with the record.

param list list a `list` of one or more bin operations, each structured as the `dict` `{'bin': bin name, 'op': aerospike.OPERATOR_* [, 'val': value]}`. See *Operators*.

param dict meta optional record metadata to be set, with field `'ttl'` set to `int` number of seconds or one of `aerospike.TTL_NAMESPACE_DEFAULT`, `aerospike.TTL_NEVER_EXPIRE`, `aerospike.TTL_DONT_UPDATE`

param dict policy optional *Operate Policies*.

return a *Record Tuple*. See *Unicode Handling*.

raises a subclass of *AerospikeError*.

Note: In version *2.1.3* the return format of bin entries for this method, **only in cases when a map operation specifying a 'return_type' is used**, has changed. Map operations using "return_type" of aerospike.MAP_RETURN_KEY_VALUE will now return a bin value of a list of keys and corresponding values, rather than a list of key/value tuples. See the following code block for details. In addition, some operations which did not return a value in versions <= 2.1.2 will now return a response.

```
# pre 2.1.3 formatting of key/value bin value
[('key1', 'val1'), ('key2', 'val2'), ('key3', 'val3')]

# >= 2.1.3 formatting
['key1', 'val1', 'key2', 'val2', 'key3', 'val3']
```

```
from __future__ import print_function
import aerospike
from aerospike import exception as ex
import sys

config = { 'hosts': [('127.0.0.1', 3000)] }
client = aerospike.client(config).connect()

try:
    key = ('test', 'demo', 1)
    policy = {
        'total_timeout': 1000,
        'key': aerospike.POLICY_KEY_SEND,
        'commit_level': aerospike.POLICY_COMMIT_LEVEL_MASTER
    }

    llist = [{"op": aerospike.OPERATOR_APPEND,
              "bin": "name",
              "val": "aa"},
```

```
            {"op": aerospike.OPERATOR_READ,
             "bin": "name"},
            {"op": aerospike.OPERATOR_INCR,
             "bin": "age",
             "val": 3}]

    client.operate_ordered(key, llist, {}, policy)
except ex.AerospikeError as e:
    print("Error: {0} [{1}]".format(e.msg, e.code))
    sys.exit(1)
finally:
    client.close()
```

New in version 2.0.2.

Changed in version 2.1.3.

Batch Operations

get_many (*keys*[, *policy*]) → [(key, meta, bins)]

Batch-read multiple records, and return them as a `list`. Any record that does not exist will have a `None` value for metadata and bins in the record tuple.

Parameters

- **keys** (*list*) – a list of *Key Tuple*.
- **policy** (*dict*) – optional *Batch Policies*.

Returns a `list` of *Record Tuple*.

Raises a *ClientError* if the batch is too big.

See also:

More information about the Batch Index interface new to Aerospike server >= 3.6.0.

```
from __future__ import print_function
import aerospike
from aerospike import exception as ex
import sys

config = { 'hosts': [('127.0.0.1', 3000)] }
client = aerospike.client(config).connect()

try:
    # assume the fourth key has no matching record
    keys = [
      ('test', 'demo', '1'),
      ('test', 'demo', '2'),
      ('test', 'demo', '3'),
      ('test', 'demo', '4')
    ]
    records = client.get_many(keys)
    print records
except ex.AerospikeError as e:
    print("Error: {0} [{1}]".format(e.msg, e.code))
    sys.exit(1)
```

```
finally:
    client.close()
```

Note: We expect to see something like:

```
[
  (('test', 'demo', '1', bytearray(b'ev\xb4\x88\x8c\xcf\x92\x9c␣
↪\x0bo\xbd\x90\xd0\x9d\xf3\xf6\xd1\x0c\xf3')), {'gen': 1, 'ttl': 2592000}, {
↪'age': 1, 'name': u'Name1'}),
  (('test', 'demo', '2', bytearray(b
↪'n\xcd7p\x88\xdcF\xe1\xd6\x0e\x05\xfb\xcbs\xa68I\xf0T\xfd')), {'gen': 1,
↪'ttl': 2592000}, {'age': 2, 'name': u'Name2'}),
  (('test', 'demo', '3', bytearray(b'\x9f\xf2\xe3\xf3\xc0\xc1\xc3q\xb5
↪$n\xf8\xccV\xa9\xed\xd91a\x86')), {'gen': 1, 'ttl': 2592000}, {'age': 3,
↪'name': u'Name3'}),
  (('test', 'demo', '4', bytearray(b'\x8eu\x19\xbe\xe0(\xda ^
↪\xfa\x8ca\x93s\xe8\xb3%\xa8]\x8b')), None, None)
]
```

Warning: The return type changed to `list` starting with version 1.0.50.

Changed in version 1.0.50.

exists_many (*keys*[, *policy*]) → [(key, meta)]

Batch-read metadata for multiple keys, and return it as a `list`. Any record that does not exist will have a `None` value for metadata in the result tuple.

Parameters

- **keys** (*list*) – a list of *Key Tuple*.
- **policy** (*dict*) – optional *Batch Policies*.

Returns a `list` of (key, metadata) `tuple()`.

See also:

More information about the Batch Index interface new to Aerospike server >= 3.6.0.

```
from __future__ import print_function
import aerospike
from aerospike import exception as ex
import sys

config = { 'hosts': [('127.0.0.1', 3000)] }
client = aerospike.client(config).connect()

try:
    # assume the fourth key has no matching record
    keys = [
      ('test', 'demo', '1'),
      ('test', 'demo', '2'),
      ('test', 'demo', '3'),
      ('test', 'demo', '4')
    ]
    records = client.exists_many(keys)
```

```
    print records
except ex.AerospikeError as e:
    print("Error: {0} [{1}]".format(e.msg, e.code))
    sys.exit(1)
finally:
    client.close()
```

Note: We expect to see something like:

```
[
   (('test', 'demo', '1', bytearray(b'ev\xb4\x88\x8c\xcf\x92\x9c␣
↪\x0bo\xbd\x90\xd0\x9d\xf3\xf6\xd1\x0c\xf3')), {'gen': 2, 'ttl': 2592000}),
   (('test', 'demo', '2', bytearray(b
↪'n\xcd7p\x88\xdcF\xe1\xd6\x0e\x05\xfb\xcbs\xa68I\xf0T\xfd')), {'gen': 7,
↪'ttl': 1337}),
   (('test', 'demo', '3', bytearray(b'\x9f\xf2\xe3\xf3\xc0\xc1\xc3q\xb5
↪$n\xf8\xccV\xa9\xed\xd91a\x86')), {'gen': 9, 'ttl': 543}),
   (('test', 'demo', '4', bytearray(b'\x8eu\x19\xbe\xe0(\xda ^
↪\xfa\x8ca\x93s\xe8\xb3%\xa8]\x8b')), None)
]
```

Warning: The return type changed to `list` starting with version 1.0.50.

Changed in version 1.0.50.

select_many (*keys*, *bins*[, *policy*]) → [(key, meta, bins), ...]}

Batch-read multiple records, and return them as a `list`. Any record that does not exist will have a `None` value for metadata and bins in the record tuple. The *bins* will be filtered as specified.

Parameters

- **keys** (*list*) – a list of *Key Tuple*.
- **bins** (*list*) – the bin names to select from the matching records.
- **policy** (*dict*) – optional *Batch Policies*.

Returns a `list` of *Record Tuple*.

See also:

More information about the Batch Index interface new to Aerospike server >= 3.6.0.

```
from __future__ import print_function
import aerospike
from aerospike import exception as ex
import sys

config = { 'hosts': [('127.0.0.1', 3000)] }
client = aerospike.client(config).connect()

try:
    # assume the fourth key has no matching record
    keys = [
      ('test', 'demo', None, bytearray(b'ev\xb4\x88\x8c\xcf\x92\x9c␣
↪\x0bo\xbd\x90\xd0\x9d\xf3\xf6\xd1\x0c\xf3'),
```

```
      ('test', 'demo', None, bytearray(b
↪'n\xcd7p\x88\xdcF\xe1\xd6\x0e\x05\xfb\xcbs\xa68I\xf0T\xfd'),
      ('test', 'demo', None, bytearray(b'\x9f\xf2\xe3\xf3\xc0\xc1\xc3q\xb5
↪$n\xf8\xccV\xa9\xed\xd91a\x86'),
      ('test', 'demo', None, bytearray(b'\x8eu\x19\xbe\xe0(\xda ^
↪\xfa\x8ca\x93s\xe8\xb3%\xa8]\x8b')
    ]
    records = client.select_many(keys, [u'name'])
    print records
except ex.AerospikeError as e:
    print("Error: {0} [{1}]".format(e.msg, e.code))
    sys.exit(1)
finally:
    client.close()
```

Note: We expect to see something like:

```
[
  (('test', 'demo', None, bytearray(b'ev\xb4\x88\x8c\xcf\x92\x9c␣
↪\x0bo\xbd\x90\xd0\x9d\xf3\xf6\xd1\x0c\xf3'), {'gen': 1, 'ttl': 2592000}, {
↪'name': u'Name1'}),
  (('test', 'demo', None, bytearray(b
↪'n\xcd7p\x88\xdcF\xe1\xd6\x0e\x05\xfb\xcbs\xa68I\xf0T\xfd'), {'gen': 1, 'ttl
↪': 2592000}, {'name': u'Name2'}),
  (('test', 'demo', None, bytearray(b'\x9f\xf2\xe3\xf3\xc0\xc1\xc3q\xb5
↪$n\xf8\xccV\xa9\xed\xd91a\x86'), {'gen': 1, 'ttl': 2592000}, {'name': u
↪'Name3'}),
  (('test', 'demo', None, bytearray(b'\x8eu\x19\xbe\xe0(\xda ^
↪\xfa\x8ca\x93s\xe8\xb3%\xa8]\x8b'), None, None)
]
```

Warning: The return type changed to `list` starting with version 1.0.50.

Changed in version 1.0.50.

Scans

scan(*namespace*[, *set*]) → Scan

Return a *aerospike.Scan* object to be used for executing scans over a specified *set* (which can be omitted or None) in a *namespace*. A scan with a None set returns all the records in the namespace.

Parameters

- **namespace** (*str*) – the namespace in the aerospike cluster.
- **set** (*str*) – optional specified set name, otherwise the entire *namespace* will be scanned.

Returns an *aerospike.Scan* class.

Queries

query (*namespace*[, *set*]) → Query
Return a *aerospike.Query* object to be used for executing queries over a specified *set* (which can be omitted or None) in a *namespace*. A query with a None set returns records which are **not in any named set**. This is different than the meaning of a None set in a scan.

Parameters

- **namespace** (*str*) – the namespace in the aerospike cluster.
- **set** (*str*) – optional specified set name, otherwise the records which are not part of any *set* will be queried (**Note**: this is different from not providing the *set* in *scan()*).

Returns an *aerospike.Query* class.

UDFs

udf_put (*filename*[, *udf_type=aerospike.UDF_TYPE_LUA*[, *policy*]])
Register a UDF module with the cluster.

Parameters

- **filename** (*str*) – the path to the UDF module to be registered with the cluster.
- **udf_type** (*int*) – one of aerospike.UDF_TYPE_*
- **policy** (*dict*) – currently **timeout** in milliseconds is the available policy.

Raises a subclass of *AerospikeError*.

Note: Register the UDF module and copy it to the Lua 'user_path', a directory that should contain a copy of the modules registered with the cluster.

```
config = {
    'hosts': [ ('127.0.0.1', 3000)],
    'lua': { 'user_path': '/path/to/lua/user_path'}}
client = aerospike.client(config).connect()
client.udf_put('/path/to/my_module.lua')
client.close()
```

Changed in version 1.0.45.

udf_remove (*module*[, *policy*])
Remove a previously registered UDF module from the cluster.

Parameters

- **module** (*str*) – the UDF module to be deregistered from the cluster.
- **policy** (*dict*) – currently **timeout** in milliseconds is the available policy.

Raises a subclass of *AerospikeError*.

```
client.udf_remove('my_module.lua')
```

Changed in version 1.0.39.

udf_list ([*policy*]) → []
Return the list of UDF modules registered with the cluster.

Parameters **policy** (*dict*) – currently **timeout** in milliseconds is the available policy.

Return type list

Raises a subclass of *AerospikeError*.

```
from __future__ import print_function
import aerospike

config = {'hosts': [('127.0.0.1', 3000)] }
client = aerospike.client(config).connect()
print(client.udf_list())
client.close()
```

Note: We expect to see something like:

```
[{'content': bytearray(b''),
  'hash': bytearray(b'195e39ceb51c110950bd'),
  'name': 'my_udf1.lua',
  'type': 0},
 {'content': bytearray(b''),
  'hash': bytearray(b'8a2528e8475271877b3b'),
  'name': 'stream_udf.lua',
  'type': 0},
 {'content': bytearray(b''),
  'hash': bytearray(b'362ea79c8b64857701c2'),
  'name': 'aggregate_udf.lua',
  'type': 0},
 {'content': bytearray(b''),
  'hash': bytearray(b'635f47081431379baa4b'),
  'name': 'module.lua',
  'type': 0}]
```

Changed in version 1.0.39.

udf_get (*module*[, *language=aerospike.UDF_TYPE_LUA*[, *policy*]]) → str
Return the content of a UDF module which is registered with the cluster.

Parameters

- **module** (*str*) – the UDF module to read from the cluster.
- **udf_type** (*int*) – one of aerospike.UDF_TYPE_*
- **policy** (*dict*) – currently **timeout** in milliseconds is the available policy.

Return type str

Raises a subclass of *AerospikeError*.

Changed in version 1.0.39.

apply (*key*, *module*, *function*, *args*[, *policy*])
Apply a registered (see *udf_put()*) record UDF to a particular record.

Parameters

- **key** (*tuple*) – a *Key Tuple* associated with the record.

- **module** (*str*) – the name of the UDF module.
- **function** (*str*) – the name of the UDF to apply to the record identified by *key*.
- **args** (*list*) – the arguments to the UDF.
- **policy** (*dict*) – optional *Apply Policies*.

Returns the value optionally returned by the UDF, one of `str`,`int`, `float`, `bytearray`, `list`, `dict`.

Raises a subclass of *AerospikeError*.

See also:

Record UDF and Developing Record UDFs.

scan_apply(*ns*, *set*, *module*, *function*[, *args*[, *policy*[, *options*]]]) → int

Initiate a background scan and apply a record UDF to each record matched by the scan.

Parameters

- **ns** (*str*) – the namespace in the aerospike cluster.
- **set** (*str*) – the set name. Should be `None` if the entire namespace is to be scanned.
- **module** (*str*) – the name of the UDF module.
- **function** (*str*) – the name of the UDF to apply to the records matched by the scan.
- **args** (*list*) – the arguments to the UDF.
- **policy** (*dict*) – optional *Scan Policies*.
- **options** (*dict*) – the *Scan Options* that will apply to the scan.

Return type `int`

Returns a job ID that can be used with *job_info()* to track the status of the `aerospike.JOB_SCAN`, as it runs in the background.

Raises a subclass of *AerospikeError*.

See also:

Record UDF and Developing Record UDFs.

query_apply(*ns*, *set*, *predicate*, *module*, *function*[, *args*[, *policy*]]) → int

Initiate a background query and apply a record UDF to each record matched by the query.

Parameters

- **ns** (*str*) – the namespace in the aerospike cluster.
- **set** (*str*) – the set name. Should be `None` if you want to query records in the *ns* which are in no set.
- **predicate** (*tuple*) – the `tuple()` produced by one of the *aerospike.predicates* methods.
- **module** (*str*) – the name of the UDF module.
- **function** (*str*) – the name of the UDF to apply to the records matched by the query.
- **args** (*list*) – the arguments to the UDF.
- **policy** (*dict*) – optional *Write Policies*.

Return type `int`

Returns a job ID that can be used with *job_info()* to track the status of the aerospike.JOB_QUERY , as it runs in the background.

Raises a subclass of *AerospikeError*.

See also:

Record UDF and Developing Record UDFs.

> **Warning:** This functionality will become available with a future release of the Aerospike server.

New in version 1.0.50.

job_info (*job_id*, *module*[, *policy*]) → dict

Return the status of a job running in the background.

Parameters

- **job_id** (*int*) – the job ID returned by *scan_apply()* and *query_apply()*.
- **module** – one of aerospike.JOB_SCAN or aerospike.JOB_QUERY.

Returns a dict with keys *status*, *records_read*, and *progress_pct*. The value of *status* is one of aerospike.JOB_STATUS_*. See: *Job Constants*.

Raises a subclass of *AerospikeError*.

```
from __future__ import print_function
import aerospike
from aerospike import exception as ex
import time

config = {'hosts': [ ('127.0.0.1', 3000)]}
client = aerospike.client(config).connect()
try:
    # run the UDF 'add_val' in Lua module 'simple' on the records of test.demo
    job_id = client.scan_apply('test', 'demo', 'simple', 'add_val', ['age',␣
↪1])
    while True:
        time.sleep(0.25)
        response = client.job_info(job_id, aerospike.JOB_SCAN)
        if response['status'] == aerospike.JOB_STATUS_COMPLETED:
            break
    print("Job ", str(job_id), " completed")
    print("Progress percentage : ", response['progress_pct'])
    print("Number of scanned records : ", response['records_read'])
except ex.AerospikeError as e:
    print("Error: {0} [{1}]".format(e.msg, e.code))
client.close()
```

New in version 1.0.50.

scan_info (*scan_id*) → dict

Return the status of a scan running in the background.

Parameters **scan_id** (*int*) – the scan ID returned by *scan_apply()*.

Returns a dict with keys *status*, *records_scanned*, and *progress_pct*. The value of *status* is one of aerospike.SCAN_STATUS_*. See: *Scan Constants*.

Raises a subclass of *AerospikeError*.

Deprecated since version 1.0.50: Use *job_info()* instead.

```
from __future__ import print_function
import aerospike
from aerospike import exception as ex

config = {'hosts': [ ('127.0.0.1', 3000)]}
client = aerospike.client(config).connect()
try:
    scan_id = client.scan_apply('test', 'demo', 'simple', 'add_val', ['age',␣
↪1])
    while True:
        response = client.scan_info(scan_id)
        if response['status'] == aerospike.SCAN_STATUS_COMPLETED or \
           response['status'] == aerospike.SCAN_STATUS_ABORTED:
            break
    if response['status'] == aerospike.SCAN_STATUS_COMPLETED:
        print("Background scan successful")
        print("Progress percentage : ", response['progress_pct'])
        print("Number of scanned records : ", response['records_scanned'])
        print("Background scan status : ", "SCAN_STATUS_COMPLETED")
    else:
        print("Scan_apply failed")
except ex.AerospikeError as e:
    print("Error: {0} [{1}]".format(e.msg, e.code))
client.close()
```

Info

index_string_create(*ns*, *set*, *bin*, *index_name*[, *policy*])

Create a string index with *index_name* on the *bin* in the specified *ns*, *set*.

Parameters

- **ns** (*str*) – the namespace in the aerospike cluster.
- **set** (*str*) – the set name.
- **bin** (*str*) – the name of bin the secondary index is built on.
- **index_name** (*str*) – the name of the index.
- **policy** (*dict*) – optional *Info Policies*.

Raises a subclass of *AerospikeError*.

Changed in version 1.0.39.

index_integer_create(*ns*, *set*, *bin*, *index_name*[, *policy*])

Create an integer index with *index_name* on the *bin* in the specified *ns*, *set*.

Parameters

- **ns** (*str*) – the namespace in the aerospike cluster.
- **set** (*str*) – the set name.
- **bin** (*str*) – the name of bin the secondary index is built on.
- **index_name** (*str*) – the name of the index.
- **policy** (*dict*) – optional *Info Policies*.

Raises a subclass of *AerospikeError*.

Changed in version 1.0.39.

index_list_create(*ns*, *set*, *bin*, *index_datatype*, *index_name*[, *policy*])
Create an index named *index_name* for numeric, string or GeoJSON values (as defined by *index_datatype*) on records of the specified *ns*, *set* whose *bin* is a list.

Parameters

- **ns** (*str*) – the namespace in the aerospike cluster.
- **set** (*str*) – the set name.
- **bin** (*str*) – the name of bin the secondary index is built on.
- **index_datatype** – Possible values are `aerospike.INDEX_STRING`, `aerospike.INDEX_NUMERIC` and `aerospike.INDEX_GEO2DSPHERE`.
- **index_name** (*str*) – the name of the index.
- **policy** (*dict*) – optional *Info Policies*.

Raises a subclass of *AerospikeError*.

Note: Requires server version >= 3.8.0

New in version 1.0.42.

index_map_keys_create(*ns*, *set*, *bin*, *index_datatype*, *index_name*[, *policy*])
Create an index named *index_name* for numeric, string or GeoJSON values (as defined by *index_datatype*) on records of the specified *ns*, *set* whose *bin* is a map. The index will include the keys of the map.

Parameters

- **ns** (*str*) – the namespace in the aerospike cluster.
- **set** (*str*) – the set name.
- **bin** (*str*) – the name of bin the secondary index is built on.
- **index_datatype** – Possible values are `aerospike.INDEX_STRING`, `aerospike.INDEX_NUMERIC` and `aerospike.INDEX_GEO2DSPHERE`.
- **index_name** (*str*) – the name of the index.
- **policy** (*dict*) – optional *Info Policies*.

Raises a subclass of *AerospikeError*.

Note: Requires server version >= 3.8.0

New in version 1.0.42.

index_map_values_create(*ns*, *set*, *bin*, *index_datatype*, *index_name*[, *policy*])
Create an index named *index_name* for numeric, string or GeoJSON values (as defined by *index_datatype*) on records of the specified *ns*, *set* whose *bin* is a map. The index will include the values of the map.

Parameters

- **ns** (*str*) – the namespace in the aerospike cluster.
- **set** (*str*) – the set name.

- **bin** (*str*) – the name of bin the secondary index is built on.
- **index_datatype** – Possible values are `aerospike.INDEX_STRING`, `aerospike.INDEX_NUMERIC` and `aerospike.INDEX_GEO2DSPHERE`.
- **index_name** (*str*) – the name of the index.
- **policy** (*dict*) – optional *Info Policies*.

Raises a subclass of *AerospikeError*.

Note: Requires server version >= 3.8.0

```
import aerospike

client = aerospike.client({ 'hosts': [ ('127.0.0.1', 3000)]}).connect()

# assume the bin fav_movies in the set test.demo bin should contain
# a dict { (str) _title_ : (int) _times_viewed_ }
# create a secondary index for string values of test.demo records whose 'fav_
↪movies' bin is a map
client.index_map_keys_create('test', 'demo', 'fav_movies', aerospike.INDEX_
↪STRING, 'demo_fav_movies_titles_idx')
# create a secondary index for integer values of test.demo records whose 'fav_
↪movies' bin is a map
client.index_map_values_create('test', 'demo', 'fav_movies', aerospike.INDEX_
↪NUMERIC, 'demo_fav_movies_views_idx')
client.close()
```

New in version 1.0.42.

index_geo2dsphere_create(*ns*, *set*, *bin*, *index_name*[, *policy*])

Create a geospatial 2D spherical index with *index_name* on the *bin* in the specified *ns*, *set*.

Parameters

- **ns** (*str*) – the namespace in the aerospike cluster.
- **set** (*str*) – the set name.
- **bin** (*str*) – the name of bin the secondary index is built on.
- **index_name** (*str*) – the name of the index.
- **policy** (*dict*) – optional *Info Policies*.

Raises a subclass of *AerospikeError*.

See also:

aerospike.GeoJSON, *aerospike.predicates*

Note: Requires server version >= 3.7.0

```
import aerospike

client = aerospike.client({ 'hosts': [ ('127.0.0.1', 3000)]}).connect()
client.index_geo2dsphere_create('test', 'pads', 'loc', 'pads_loc_geo')
client.close()
```

New in version 1.0.53.

index_remove(*ns*, *index_name*[, *policy*])
Remove the index with *index_name* from the namespace.

Parameters

- **ns** (*str*) – the namespace in the aerospike cluster.
- **index_name** (*str*) – the name of the index.
- **policy** (*dict*) – optional *Info Policies*.

Raises a subclass of *AerospikeError*.

Changed in version 1.0.39.

get_nodes() → []

Return the list of hosts present in a connected cluster.

return a `list` of node address tuples.

raises a subclass of *AerospikeError*.

```
import aerospike

config = {'hosts': [('127.0.0.1', 3000)] }
client = aerospike.client(config).connect()

nodes = client.get_nodes()
print(nodes)
client.close()
```

Note: We expect to see something like:

```
[('127.0.0.1', 3000), ('127.0.0.1', 3010)]
```

Changed in version 3.0.0.

Warning: In versions < 3.0.0 `get_nodes` will not work when using TLS

info(*command*[, *hosts*[, *policy*]]) → {}
Deprecated since version 3.0.0: Use *info_node()* to send a request to a single node, or *info_all()* to send a request to the entire cluster. Sending requests to specific nodes can be better handled with a simple Python function such as:

```
def info_to_host_list(client, request, hosts, policy=None):
    output = {}
    for host in hosts:
        try:
            response = client.info_node(request, host, policy)
            output[host] = response
        except Exception as e:
            #  Handle the error gracefully here
            output[host] = e
    return output
```

Send an info *command* to all nodes in the cluster and filter responses to only include nodes specified in a *hosts* list.

Parameters

- **command** (*str*) – the info command.
- **hosts** (*list*) – a `list` containing an *address*, *port* `tuple()`. Example: `[('127.0.0.1', 3000)]`
- **policy** (*dict*) – optional *Info Policies*.

Return type `dict`

Raises a subclass of *AerospikeError*.

See also:

Info Command Reference.

```
import aerospike

config = {'hosts': [('127.0.0.1', 3000)] }
client = aerospike.client(config).connect()

response = client.info(command)
client.close()
```

Note: We expect to see something like:

```
{'BB9581F41290C00': (None, '127.0.0.1:3000\n'), 'BC3581F41290C00': (None,
↪'127.0.0.1:3010\n')}
```

Changed in version 3.0.0.

info_all (*command[, policy]]*) → {}

Send an info *command* to all nodes in the cluster to which the client is connected. If any of the individual requests fail, this will raise an exception.

Parameters

- **command** (*str*) – the info command.
- **policy** (*dict*) – optional *Info Policies*.

Return type `dict`

Raises a subclass of *AerospikeError*.

See also:

Info Command Reference.

```
import aerospike

config = {'hosts': [('127.0.0.1', 3000)] }
client = aerospike.client(config).connect()

response = client.info_all(command)
client.close()
```

Note: We expect to see something like:

```
{'BB9581F41290C00': (None, '127.0.0.1:3000\n'), 'BC3581F41290C00': (None,
↪'127.0.0.1:3010\n')}
```

New in version 3.0.0.

info_node(*command*, *host*[, *policy*]) → str

Send an info *command* to a single node specified by *host*.

Parameters

- **command** (*str*) – the info command.
- **host** (*tuple*) – a `tuple()` containing an *address*, *port* , optional *tls-name* . Example: `('127.0.0.1', 3000)` or when using TLS `('127.0.0.1', 4333, 'server-tls-name')`. In order to send an info request when TLS is enabled, the *tls-name* must be present.
- **policy** (*dict*) – optional *Info Policies*.

Return type `str`

Raises a subclass of *AerospikeError*.

See also:

Info Command Reference.

Changed in version 3.0.0.

> **Warning:** for client versions < 3.0.0 `info_node` will not work when using TLS

has_geo() → bool

Check whether the connected cluster supports geospatial data and indexes.

Return type `bool`

Raises a subclass of *AerospikeError*.

New in version 1.0.53.

shm_key() → int

Expose the value of the shm_key for this client if shared-memory cluster tending is enabled,

Return type `int` or `None`

New in version 1.0.56.

truncate(*namespace*, *set*, *nanos*[, *policy*])

Remove records in specified namespace/set efficiently. This method is many orders of magnitude faster than deleting records one at a time. Works with Aerospike Server versions >= 3.12.

This asynchronous server call may return before the truncation is complete. The user can still write new records after the server returns because new records will have last update times greater than the truncate cutoff (set at the time of truncate call)

Parameters

- **namespace** (*str*) – The namespace on which the truncation operation should be performed.
- **set** (*str*) – The set to truncate. Pass in `None` to indicate that all records in the namespace should be truncated.
- **nanos** (*long*) – A cutoff threshold indicating that records last updated before the threshold will be removed.Units are in nanoseconds since unix epoch (1970-01-01). A value of `0` indicates that all records in the set should be truncated regardless of update time. The value must not be in the future.
- **policy** (*dict*) – Optional *Info Policies*

Return type Status indicating the success of the operation.

Raises a subclass of *AerospikeError*.

New in version 2.0.11.

```
import aerospike
import time

client = aerospike.client({'hosts': [('localhost', 3000)]}).connect()

# Store 10 items in the database
for i in range(10):
    key = ('test', 'truncate', i)
    record = {'item': i}
    client.put(key, record)

time.sleep(2)
current_time = time.time()
# Convert the current time to nanoseconds since epoch
threshold_ns = int(current_time * 10 ** 9)

time.sleep(2)  # Make sure some time passes before next round of additions

# Store another 10 items into the database
for i in range(10, 20):
    key = ('test', 'truncate', i)
    record = {'item': i}
    client.put(key, record)

# Store a record in the 'test' namespace without a set
key = ('test', None, 'no set')
record = ({'item': 'no set'})
client.put(key, record)

# Remove all items created before the threshold time
# The first 10 records we added will be removed by this call.
# The second 10 will remain.
client.truncate('test', 'truncate', threshold_ns)


# Remove all records from test/truncate.
# After this the record with key ('test', None, 'no set') still exists
client.truncate('test', 'truncate', 0)

# Remove all records from the test namespace
client.truncate('test', None, 0)

client.close()
```

Admin

Note: The admin methods implement the security features of the Enterprise Edition of Aerospike. These methods will raise a *SecurityNotSupported* when the client is connected to a Community Edition cluster (see *aerospike.exception*).

A user is validated by the client against the server whenever a connection is established through the use of a username and password (passwords hashed using bcrypt). When security is enabled, each operation is validated against the user's roles. Users are assigned roles, which are collections of *Privilege Objects*.

```
import aerospike
from aerospike import exception as ex
import time

config = {'hosts': [('127.0.0.1', 3000)] }
client = aerospike.client(config).connect('ipji', 'life is good')

try:
    dev_privileges = [{'code': aerospike.PRIV_READ}, {'code': aerospike.PRIV_READ_
↪WRITE}]
    client.admin_create_role('dev_role', dev_privileges)
    client.admin_grant_privileges('dev_role', [{'code': aerospike.PRIV_READ_WRITE_
↪UDF}])
    client.admin_create_user('dev', 'you young whatchacallit... idiot', ['dev_role
↪'])
    time.sleep(1)
    print(client.admin_query_user('dev'))
    print(admin_query_users())
except ex.AdminError as e:
    print("Error [{0}]: {1}".format(e.code, e.msg))
client.close()
```

See also:

Security features article.

---

admin_create_role(*role*, *privileges*[, *policy*])
Create a custom, named *role* containing a `list` of *privileges*.

Parameters

- **role** (*str*) – the name of the role.
- **privileges** (*list*) – a list of *Privilege Objects*.
- **policy** (*dict*) – optional *Admin Policies*.

Raises one of the *AdminError* subclasses.

Changed in version 1.0.44.

admin_drop_role(*role*[, *policy*])
Drop a custom *role*.

Parameters

- **role** (*str*) – the name of the role.
- **policy** (*dict*) – optional *Admin Policies*.

Raises one of the *AdminError* subclasses.

Changed in version 1.0.44.

admin_grant_privileges(*role*, *privileges*[, *policy*])
Add *privileges* to a *role*.

Parameters

- **role** (*str*) – the name of the role.
- **privileges** (*list*) – a list of *Privilege Objects*.
- **policy** (*dict*) – optional *Admin Policies*.

Raises one of the *AdminError* subclasses.

Changed in version 1.0.44.

admin_revoke_privileges(*role*, *privileges*[, *policy*])
Remove *privileges* from a *role*.

Parameters

- **role** (*str*) – the name of the role.
- **privileges** (*list*) – a list of *Privilege Objects*.
- **policy** (*dict*) – optional *Admin Policies*.

Raises one of the *AdminError* subclasses.

Changed in version 1.0.44.

admin_query_role(*role*[, *policy*]) → []
Get the `list` of privileges associated with a *role*.

Parameters

- **role** (*str*) – the name of the role.
- **policy** (*dict*) – optional *Admin Policies*.

Returns a `list` of *Privilege Objects*.

Raises one of the *AdminError* subclasses.

Changed in version 1.0.44.

admin_query_roles([*policy*]) → {}
Get all named roles and their privileges.

Parameters **policy** (*dict*) – optional *Admin Policies*.

Returns a `dict` of *Privilege Objects* keyed by role name.

Raises one of the *AdminError* subclasses.

Changed in version 1.0.44.

admin_create_user(*username*, *password*, *roles*[, *policy*])
Create a user with a specified *username* and grant it *roles*.

Parameters

- **username** (*str*) – the username to be added to the aerospike cluster.
- **password** (*str*) – the password associated with the given username.
- **roles** (*list*) – the list of role names assigned to the user.
- **policy** (*dict*) – optional *Admin Policies*.

Raises one of the *AdminError* subclasses.

Changed in version 1.0.44.

admin_drop_user(*username*[, *policy*])
Drop the user with a specified *username* from the cluster.

Parameters

- **username** (*str*) – the username to be dropped from the aerospike cluster.
- **policy** (*dict*) – optional *Admin Policies*.

Raises one of the *AdminError* subclasses.

Changed in version 1.0.44.

admin_change_password(*username*, *password*[, *policy*])
Change the *password* of the user *username*. This operation can only be performed by that same user.

Parameters

- **username** (*str*) – the username.
- **password** (*str*) – the password associated with the given username.
- **policy** (*dict*) – optional *Admin Policies*.

Raises one of the *AdminError* subclasses.

Changed in version 1.0.44.

admin_set_password(*username*, *password*[, *policy*])
Set the *password* of the user *username* by a user administrator.

Parameters

- **username** (*str*) – the username to be added to the aerospike cluster.
- **password** (*str*) – the password associated with the given username.
- **policy** (*dict*) – optional *Admin Policies*.

Raises one of the *AdminError* subclasses.

Changed in version 1.0.44.

admin_grant_roles(*username*, *roles*[, *policy*])
Add *roles* to the user *username*.

Parameters

- **username** (*str*) – the username to be granted the roles.
- **roles** (*list*) – a list of role names.
- **policy** (*dict*) – optional *Admin Policies*.

Raises one of the *AdminError* subclasses.

Changed in version 1.0.44.

admin_revoke_roles(*username*, *roles*[, *policy*])
Remove *roles* from the user *username*.

Parameters

- **username** (*str*) – the username to have the roles revoked.
- **roles** (*list*) – a list of role names.
- **policy** (*dict*) – optional *Admin Policies*.

Raises one of the *AdminError* subclasses.

Changed in version 1.0.44.

admin_query_user(*username*[, *policy*]) → []
Return the list of roles granted to the specified user *username*.

Parameters

- **username** (*str*) – the username to query for.

- **policy** (*dict*) – optional *Admin Policies*.

Returns a `list` of role names.

Raises one of the *AdminError* subclasses.

Changed in version 1.0.44.

admin_query_users([*policy*]) → {}

Return the `dict` of users, with their roles keyed by username.

Parameters **policy** (*dict*) – optional *Admin Policies*.

Returns a `dict` of roles keyed by username.

Raises one of the *AdminError* subclasses.

Changed in version 1.0.44.

Key Tuple

key

The key tuple which is sent and returned by various operations contains

```
(namespace, set, primary key[, the record's RIPEMD-160 digest])
```

- *namespace* the `str` name of the namespace, which must be preconfigured on the cluster.
- *set* the `str` name of the set. Will be created automatically if it does not exist.
- *primary key* the value by which the client-side application identifies the record, which can be of type `str`, `int` or `bytearray`.
- *digest* the first three parts of the tuple get hashed through RIPEMD-160, and the digest used by the clients and cluster nodes to locate the record. A key tuple is also valid if it has the digest part filled and the primary key part set to `None`.

```
>>> client = aerospike.client(config).connect()
>>> client.put(('test','demo','oof'), {'id':0, 'a':1})
>>> (key, meta, bins) = client.get(('test','demo','oof'))
>>> key
('test', 'demo', None, bytearray(b'\ti\xcb\xb9\xb6V#V\xecI
↪#\xealu\x05\x00H\x98\xe4='))
>>> (key2, meta2, bins2) = client.get(key)
>>> bins2
{'a': 1, 'id': 0}
>>> client.close()
```

See also:

Data Model: Keys and Digests.

Changed in version 1.0.47.

Record Tuple

record

The record tuple `(key, meta, bins)` which is returned by various read operations.

- *key* the *Key Tuple*.
- *meta* a `dict` containing `{'gen' : genration value, 'ttl': ttl value}`.
- *bins* a `dict` containing bin-name/bin-value pairs.

See also:

Data Model: Record.

Unicode Handling

Both `str` and `unicode()` values are converted by the client into UTF-8 encoded strings for storage on the aerospike server. Read methods such as *`get()`*, *`query()`*, *`scan()`* and *`operate()`* will return that data as UTF-8 encoded `str` values. To get a `unicode()` you will need to manually decode.

Warning: Prior to release 1.0.43 read operations always returned strings as `unicode()`.

```
>>> client.put(key, { 'name': 'Dr. Zeta Alphabeta', 'age': 47})
>>> (key, meta, record) = client.get(key)
>>> type(record['name'])
<type 'str'>
>>> record['name']
'Dr. Zeta Alphabeta'
>>> client.put(key, { 'name': unichr(0x2603), 'age': 21})
>>> (key, meta, record) = client.get(key)
>>> type(record['name'])
<type 'str'>
>>> record['name']
'\xe2\x98\x83'
>>> print(record['name'])

>>> name = record['name'].decode('utf-8')
>>> type(name)
<type 'unicode'>
>>> name
u'\u2603'
>>> print(name)
```

Changed in version 1.0.43.

Write Policies

`policy`
A `dict` of optional write policies which are applicable to *`put()`*, *`query_apply()`*. *`remove_bin()`*.

- **max_retries**
  An `int`. Maximum number of retries before aborting the current transaction. The initial attempt is not counted as a retry.

  If max_retries is exceeded, the transaction will return error `AEROSPIKE_ERR_TIMEOUT`.

  WARNING: Database writes that are not idempotent (such as "add") should not be retried because the write operation may be performed multiple times
  if the client timed out previous transaction attempts. It's important to use a distinct write policy for non-idempotent writes which sets max_retries = *0*;

  Default: `0`

- **sleep_between_retries**
  An `int`. Milliseconds to sleep between retries. Enter zero to skip sleep. Default: `0`
- **socket_timeout**
  An `int`. Socket idle timeout in milliseconds when processing a database command.

  If socket_timeout is not zero and the socket has been idle for at least socket_timeout, both max_retries and total_timeout are checked. If max_retries and total_timeout are not exceeded, the transaction is retried.

  If both `socket_timeout` and `total_timeout` are non-zero and `socket_timeout` > `total_timeout`, then `socket_timeout` will be set to `total_timeout`. If `socket_timeout` is zero, there will be no socket idle limit.

  Default: `0`.
- **total_timeout**
  An `int`. Total transaction timeout in milliseconds.

  The total_timeout is tracked on the client and sent to the server along with the transaction in the wire protocol. The client will most likely timeout first, but the server also has the capability to timeout the transaction.

  If `total_timeout` is not zero and `total_timeout` is reached before the transaction completes, the transaction will return error `AEROSPIKE_ERR_TIMEOUT`. If `total_timeout` is zero, there will be no total time limit.

  Default: `1000`
- **key** one of the `aerospike.POLICY_KEY_*` values such as *aerospike.POLICY_KEY_DIGEST*
- **exists** one of the `aerospike.POLICY_EXISTS_*` values such as *aerospike.POLICY_EXISTS_CREATE*
- **gen** one of the `aerospike.POLICY_GEN_*` values such as *aerospike.POLICY_GEN_IGNORE*
- **commit_level** one of the `aerospike.POLICY_COMMIT_LEVEL_*` values such as *aerospike.POLICY_COMMIT_LEVEL_ALL*
- **durable_delete** boolean value: True to perform durable delete (requires Enterprise server version >= 3.10)

Read Policies

policy
A `dict` of optional read policies which are applicable to *get()*, *exists()*, *select()*.

- **max_retries**
  An `int`. Maximum number of retries before aborting the current transaction. The initial attempt is not counted as a retry.

  If max_retries is exceeded, the transaction will return error `AEROSPIKE_ERR_TIMEOUT`.

  WARNING: Database writes that are not idempotent (such as "add") should not be retried because the write operation may be performed multiple times
  if the client timed out previous transaction attempts. It's important to use a distinct write policy for non-idempotent writes which sets max_retries = *0*;

  Default: `2`

- **sleep_between_retries**
  An `int`. Milliseconds to sleep between retries. Enter zero to skip sleep. Default: `0`
- **socket_timeout**
  An `int`. Socket idle timeout in milliseconds when processing a database command.

  If socket_timeout is not zero and the socket has been idle for at least socket_timeout, both max_retries and total_timeout are checked. If max_retries and total_timeout are not exceeded, the transaction is retried.

  If both `socket_timeout` and `total_timeout` are non-zero and `socket_timeout` > `total_timeout`, then `socket_timeout` will be set to `total_timeout`. If `socket_timeout` is zero, there will be no socket idle limit.

  Default: `0`.
- **total_timeout**
  An `int`. Total transaction timeout in milliseconds.

  The total_timeout is tracked on the client and sent to the server along with the transaction in the wire protocol. The client will most likely timeout first, but the server also has the capability to timeout the transaction.

  If `total_timeout` is not zero and `total_timeout` is reached before the transaction completes, the transaction will return error `AEROSPIKE_ERR_TIMEOUT`. If `total_timeout` is zero, there will be no total time limit.

  Default: `1000`
- **deserialize**
  `bool` Should raw bytes representing a list or map be deserialized to a list or dictionary.
  Set to *False* for backup programs that just need access to raw bytes.
  Default: `True`
- **linearize_read**
  `bool`
  Force reads to be linearized for server namespaces that support CP mode. Setting this policy to `True` requires an Enterprise server with version 4.0.0 or greater.
  Default: `False`
- **key** one of the `aerospike.POLICY_KEY_*` values such as *aerospike.POLICY_KEY_DIGEST*
- **consistency_level** one of the `aerospike.POLICY_CONSISTENCY_*` values such as *aerospike.POLICY_CONSISTENCY_ONE*
- **replica** one of the `aerospike.POLICY_REPLICA_*` values such as *aerospike.POLICY_REPLICA_MASTER*

Operate Policies

policy
A `dict` of optional operate policies which are applicable to *append()*, *prepend()*, *increment()*, *operate()*, and atomic list and map operations.

- **max_retries**
  An `int`. Maximum number of retries before aborting the current transaction. The initial attempt is not counted as a retry.

If max_retries is exceeded, the transaction will return error `AEROSPIKE_ERR_TIMEOUT`.

WARNING: Database writes that are not idempotent (such as "add") should not be retried because the write operation may be performed multiple times
if the client timed out previous transaction attempts. It's important to use a distinct write policy for non-idempotent writes which sets max_retries = *0*;

Default: `0`

- **sleep_between_retries**
  An `int`. Milliseconds to sleep between retries. Enter zero to skip sleep.

  Default: `0`
- **socket_timeout**
  An `int`. Socket idle timeout in milliseconds when processing a database command.

  If socket_timeout is not zero and the socket has been idle for at least socket_timeout, both max_retries and total_timeout are checked. If max_retries and total_timeout are not exceeded, the transaction is retried.

  If both `socket_timeout` and `total_timeout` are non-zero and `socket_timeout` > `total_timeout`, then `socket_timeout` will be set to `total_timeout`. If `socket_timeout` is zero, there will be no socket idle limit.

  Default: `0`.
- **total_timeout**
  An `int`. Total transaction timeout in milliseconds.

  The total_timeout is tracked on the client and sent to the server along with the transaction in the wire protocol. The client will most likely timeout first, but the server also has the capability to timeout the transaction.

  If `total_timeout` is not zero and `total_timeout` is reached before the transaction completes, the transaction will return error `AEROSPIKE_ERR_TIMEOUT`. If `total_timeout` is zero, there will be no total time limit.

  Default: `1000`
- **linearize_read**
  `bool`
  Force reads to be linearized for server namespaces that support CP mode. Setting this policy to `True` requires an Enterprise server with version 4.0.0 or greater.
  Default: `False`
- **key** one of the `aerospike.POLICY_KEY_*` values such as *aerospike.POLICY_KEY_DIGEST*
- **gen** one of the `aerospike.POLICY_GEN_*` values such as *aerospike.POLICY_GEN_IGNORE*
- **replica** one of the `aerospike.POLICY_REPLICA_*` values such as *aerospike.POLICY_REPLICA_MASTER*
- **commit_level** one of the `aerospike.POLICY_COMMIT_LEVEL_*` values such as *aerospike.POLICY_COMMIT_LEVEL_ALL*
- **consistency_level** one of the `aerospike.POLICY_CONSISTENCY_*` values such as *aerospike.POLICY_CONSISTENCY_ONE*
- **durable_delete** boolean value: True to perform durable delete (requires Enterprise server version >= 3.10)

Apply Policies

policy
A dict of optional apply policies which are applicable to *apply()*.

- **max_retries**
  An int. Maximum number of retries before aborting the current transaction. The initial attempt is not counted as a retry.

  If max_retries is exceeded, the transaction will return error AEROSPIKE_ERR_TIMEOUT.

  WARNING: Database writes that are not idempotent (such as "add") should not be retried because the write operation may be performed multiple times
  if the client timed out previous transaction attempts. It's important to use a distinct write policy for non-idempotent writes which sets max_retries = *0*;

  Default: 0
- **sleep_between_retries**
  An int. Milliseconds to sleep between retries. Enter zero to skip sleep. Default: 0
- **socket_timeout**
  An int. Socket idle timeout in milliseconds when processing a database command.

  If socket_timeout is not zero and the socket has been idle for at least socket_timeout, both max_retries and total_timeout are checked. If max_retries and total_timeout are not exceeded, the transaction is retried.

  If both socket_timeout and total_timeout are non-zero and socket_timeout > total_timeout, then socket_timeout will be set to total_timeout. If socket_timeout is zero, there will be no socket idle limit.

  Default: 0.
- **total_timeout**
  An int. Total transaction timeout in milliseconds.

  The total_timeout is tracked on the client and sent to the server along with the transaction in the wire protocol. The client will most likely timeout first, but the server also has the capability to timeout the transaction.

  If total_timeout is not zero and total_timeout is reached before the transaction completes, the transaction will return error AEROSPIKE_ERR_TIMEOUT. If total_timeout is zero, there will be no total time limit.

  Default: 1000
- **linearize_read**
  bool
  Force reads to be linearized for server namespaces that support CP mode. Setting this policy to True requires an Enterprise server with version 4.0.0 or greater.
  Default: False
- **key** one of the aerospike.POLICY_KEY_* values such as *aerospike.POLICY_KEY_DIGEST*
- **commit_level** one of the aerospike.POLICY_COMMIT_LEVEL_* values such as *aerospike.POLICY_COMMIT_LEVEL_ALL*

- **durable_delete** boolean value: True to perform durable delete (requires Enterprise server version >= 3.10)

Remove Policies

policy
A `dict` of optional remove policies which are applicable to *`remove()`*.

- **max_retries**
  An `int`. Maximum number of retries before aborting the current transaction. The initial attempt is not counted as a retry.

  If max_retries is exceeded, the transaction will return error `AEROSPIKE_ERR_TIMEOUT`.

  WARNING: Database writes that are not idempotent (such as "add") should not be retried because the write operation may be performed multiple times
  if the client timed out previous transaction attempts. It's important to use a distinct write policy for non-idempotent writes which sets max_retries = *0*;

  Default: `0`
- **sleep_between_retries**
  An `int`. Milliseconds to sleep between retries. Enter zero to skip sleep. Default: `0`
- **socket_timeout**
  An `int`. Socket idle timeout in milliseconds when processing a database command.

  If socket_timeout is not zero and the socket has been idle for at least socket_timeout, both max_retries and total_timeout are checked. If max_retries and total_timeout are not exceeded, the transaction is retried.

  If both `socket_timeout` and `total_timeout` are non-zero and `socket_timeout` > `total_timeout`, then `socket_timeout` will be set to `total_timeout`. If `socket_timeout` is zero, there will be no socket idle limit.

  Default: `0`.
- **total_timeout**
  An `int`. Total transaction timeout in milliseconds.

  The total_timeout is tracked on the client and sent to the server along with the transaction in the wire protocol. The client will most likely timeout first, but the server also has the capability to timeout the transaction.

  If `total_timeout` is not zero and `total_timeout` is reached before the transaction completes, the transaction will return error `AEROSPIKE_ERR_TIMEOUT`. If `total_timeout` is zero, there will be no total time limit.

  Default: `1000`
- **key** one of the `aerospike.POLICY_KEY_*` values such as *`aerospike.POLICY_KEY_DIGEST`*
- **commit_level** one of the `aerospike.POLICY_COMMIT_LEVEL_*` values such as *`aerospike.POLICY_COMMIT_LEVEL_ALL`*
- **gen** one of the `aerospike.POLICY_GEN_*` values such as *`aerospike.POLICY_GEN_IGNORE`*
- **max_retries** integer, number of times to retry the operation if it fails due to netowrk error. Default *2*
- **durable_delete** boolean value: True to perform durable delete (requires Enterprise server version >= 3.10)

Batch Policies

policy

A `dict` of optional batch policies which are applicable to *get_many()*, *exists_many()* and *select_many()*.

- **max_retries**
  An `int`. Maximum number of retries before aborting the current transaction. The initial attempt is not counted as a retry.

  If max_retries is exceeded, the transaction will return error `AEROSPIKE_ERR_TIMEOUT`.

  WARNING: Database writes that are not idempotent (such as "add") should not be retried because the write operation may be performed multiple times
  if the client timed out previous transaction attempts. It's important to use a distinct write policy for non-idempotent writes which sets max_retries = *0*;

  Default: `2`
- **sleep_between_retries**
  An `int`. Milliseconds to sleep between retries. Enter zero to skip sleep.

  Default: `0`
- **socket_timeout**
  An `int`. Socket idle timeout in milliseconds when processing a database command.

  If socket_timeout is not zero and the socket has been idle for at least socket_timeout, both max_retries and total_timeout are checked. If max_retries and total_timeout are not exceeded, the transaction is retried.

  If both `socket_timeout` and `total_timeout` are non-zero and `socket_timeout` > `total_timeout`, then `socket_timeout` will be set to `total_timeout`. If `socket_timeout` is zero, there will be no socket idle limit.

  Default: `0`.
- **total_timeout**
  An `int`. Total transaction timeout in milliseconds.

  The total_timeout is tracked on the client and sent to the server along with the transaction in the wire protocol. The client will most likely timeout first, but the server also has the capability to timeout the transaction.

  If `total_timeout` is not zero and `total_timeout` is reached before the transaction completes, the transaction will return error `AEROSPIKE_ERR_TIMEOUT`. If `total_timeout` is zero, there will be no total time limit.

  Default: `1000`
- **linearize_read**
  `bool`
  Force reads to be linearized for server namespaces that support CP mode. Setting this policy to `True` requires an Enterprise server with version 4.0.0 or greater.

Default: `False`

- **consistency_level** one of the `aerospike.POLICY_CONSISTENCY_*` values such as *aerospike.POLICY_CONSISTENCY_ONE*
- **concurrent** `bool` Determine if batch commands to each server are run in parallel threads. Default *False*
- **allow_inline** `bool` . Allow batch to be processed immediately in the server's receiving thread when the server deems it to be appropriate. If *False*, the batch will always be processed in separate transaction threads. This field is only relevant for the new batch index protocol. Default *True*.
- **send_set_name** `bool` Send set name field to server for every key in the batch for batch index protocol. This is only necessary when authentication is enabled and security roles are defined on a per set basis. Default: *False*
- **deserialize** `bool` Should raw bytes be deserialized to as_list or as_map. Set to *False* for backup programs that just need access to raw bytes. Default: *True*

Info Policies

policy

A `dict` of optional info policies which are applicable to *info()*, *info_node()* and index operations.

- **timeout** read timeout in milliseconds

Admin Policies

policy

A `dict` of optional admin policies which are applicable to admin (security) operations.

- **timeout** admin operation timeout in milliseconds

Map Policies

policy

A `dict` of optional map policies which are applicable to map operations.

- **map_write_mode** write mode for the map. Valid values: aerospike.MAP_UPDATE, aerospike.MAP_UPDATE_ONLY, aerospike.MAP_CREATE_ONLY
- **map_order** ordering to maintain for the map entries. Valid values: aerospike.MAP_UNORDERED, aerospike.MAP_KEY_ORDERED, aerospike.MAP_KEY_VALUE_ORDERED

Privilege Objects

privilege

A `dict` describing a privilege associated with a specific role.

- **code** one of the `aerospike.PRIV_*` values such as *aerospike.PRIV_READ*
- **ns** optional namespace to which the privilege applies, otherwise the privilege applies globally.
- **set** optional set within the *ns* to which the privilege applies, otherwise to the entire namespace.

Example:

```
{'code': aerospike.PRIV_READ, 'ns': 'test', 'set': 'demo'}
```

1.4 Scan Class — Scan

1.4.1 Scan

class aerospike.**Scan**

The Scan object is used to return all the records in a specified set (which can be ommitted or None). A Scan with a None set returns all the records in the namespace.

The scan is invoked using either *foreach()* or *results()*. The bins returned can be filtered using *select()*.

See also:

Scans and Managing Scans.

select(*bin1*[, *bin2*[, *bin3..*]])

Set a filter on the record bins resulting from *results()* or *foreach()*. If a selected bin does not exist in a record it will not appear in the *bins* portion of that record tuple.

results(*[policy[, nodename]]*) -> *list of (key, meta, bins)*

Buffer the records resulting from the scan, and return them as a list of records.

Parameters

- **policy** (*dict*) – optional *Scan Policies*.
- **nodename** (*str*) – optional name of node used to limit the scan to a single node.

Returns a list of *Record Tuple*.

```
import aerospike
import pprint

pp = pprint.PrettyPrinter(indent=2)
config = { 'hosts': [ ('127.0.0.1',3000)]}
client = aerospike.client(config).connect()

client.put(('test','test','key1'), {'id':1,'a':1},
    policy={'key':aerospike.POLICY_KEY_SEND})
client.put(('test','test','key2'), {'id':2,'b':2},
    policy={'key':aerospike.POLICY_KEY_SEND})

scan = client.scan('test', 'test')
scan.select('id','a','zzz')
res = scan.results()
pp.pprint(res)
client.close()
```

Note: We expect to see:

```
[ ( ( 'test',
      'test',
      u'key2',
      bytearray(b'\xb2\x18\n\xd4\xce\xd8\xba:\x96s\xf5\x9ba\xf1j\xa7t\xeem\x01
↪')),
    { 'gen': 52, 'ttl': 2592000},
    { 'id': 2}),
  ( ( 'test',
      'test',
```

```
    u'key1',
    bytearray(b'\x1cJ\xce\xa7\xd4Vj\xef+\xdf@W\xa5\xd8o\x8d:\xc9\xf4\xde')),
  { 'gen': 52, 'ttl': 2592000},
  { 'a': 1, 'id': 1})]
```

foreach (*callback*[, *policy*[, *options*[, *nodename*]]])

Invoke the *callback* function for each of the records streaming back from the scan.

Parameters

- **callback** (*callable*) – the function to invoke for each record.
- **policy** (*dict*) – optional *Scan Policies*.
- **options** (*dict*) – the *Scan Options* that will apply to the scan.
- **nodename** (*str*) – optional name of node used to limit the scan to a single node.

Note: A *Record Tuple* is passed as the argument to the callback function.

```
import aerospike
import pprint

pp = pprint.PrettyPrinter(indent=2)
config = { 'hosts': [ ('127.0.0.1',3000)]}
client = aerospike.client(config).connect()

client.put(('test','test','key1'), {'id':1,'a':1},
    policy={'key':aerospike.POLICY_KEY_SEND})
client.put(('test','test','key2'), {'id':2,'b':2},
    policy={'key':aerospike.POLICY_KEY_SEND})

def show_key((key, meta, bins)):
    print(key)

scan = client.scan('test', 'test')
scan_opts = {
  'concurrent': True,
  'nobins': True,
  'priority': aerospike.SCAN_PRIORITY_MEDIUM
}
scan.foreach(show_key, options=scan_opts)
client.close()
```

Note: We expect to see:

```
('test', 'test', u'key2', bytearray(b
↪'\xb2\x18\n\xd4\xce\xd8\xba:\x96s\xf5\x9ba\xf1j\xa7t\xeem\x01'))
('test', 'test', u'key1', bytearray(b
↪'\x1cJ\xce\xa7\xd4Vj\xef+\xdf@W\xa5\xd8o\x8d:\xc9\xf4\xde'))
```

Note: To stop the stream return `False` from the callback function.

```
from __future__ import print_function
import aerospike

config = { 'hosts': [ ('127.0.0.1',3000)]}
client = aerospike.client(config).connect()

def limit(lim, result):
    c = [0] # integers are immutable so a list (mutable) is used for the␣
↪counter
    def key_add((key, metadata, bins)):
        if c[0] < lim:
            result.append(key)
            c[0] = c[0] + 1
        else:
            return False
    return key_add

scan = client.scan('test','user')
keys = []
scan.foreach(limit(100, keys))
print(len(keys)) # this will be 100 if the number of matching records > 100
client.close()
```

Scan Policies

policy
A `dict` of optional scan policies which are applicable to *`Scan.results()`* and *`Scan.foreach()`*. See *Policies*.

- **max_retries**
  An `int`. Maximum number of retries before aborting the current transaction. The initial attempt is not counted as a retry.

  If max_retries is exceeded, the transaction will return error `AEROSPIKE_ERR_TIMEOUT`.

  WARNING: Database writes that are not idempotent (such as "add") should not be retried because the write operation may be performed multiple times
  if the client timed out previous transaction attempts. It's important to use a distinct write policy for non-idempotent writes which sets max_retries = *0*;

  Default: `0`
- **sleep_between_retries**
  An `int`. Milliseconds to sleep between retries. Enter zero to skip sleep. Default: `0`
- **socket_timeout**
  An `int`. Socket idle timeout in milliseconds when processing a database command.

  If socket_timeout is not zero and the socket has been idle for at least socket_timeout, both max_retries and total_timeout are checked. If max_retries and total_timeout are not exceeded, the transaction is retried.

  If both `socket_timeout` and `total_timeout` are non-zero and `socket_timeout` > `total_timeout`, then `socket_timeout` will be set to `total_timeout`. If

socket_timeout is zero, there will be no socket idle limit.

Default: 10000.

- **total_timeout**
  An int. Total transaction timeout in milliseconds.

  The total_timeout is tracked on the client and sent to the server along with the transaction in the wire protocol. The client will most likely timeout first, but the server also has the capability to timeout the transaction.

  If total_timeout is not zero and total_timeout is reached before the transaction completes, the transaction will return error AEROSPIKE_ERR_TIMEOUT. If total_timeout is zero, there will be no total time limit.

  Default: 0
- **fail_on_cluster_change** bool: Abort the scan if the cluster is not in a stable state. Default: False
- **durable_delete**
  A bool : If the transaction results in a record deletion, leave a tombstone for the record.

  This prevents deleted records from reappearing after node failures.

  Valid for Aerospike Server Enterprise Edition only.

  Default: False (do not tombstone deleted records).

Scan Options

options
A dict of optional scan options which are applicable to *Scan.foreach()*.

- **priority** See *Scan Constants* for values. Default aerospike.SCAN_PRIORITY_AUTO.
- **nobins** bool whether to return the *bins* portion of the *Record Tuple*. Default False.
- **concurrent** bool whether to run the scan concurrently on all nodes of the cluster. Default False.
- **percent** int percentage of records to return from the scan. Default 100.

New in version 1.0.39.

1.5 Query Class — Query

1.5.1 Query

class aerospike.**Query**
The Query object created by calling *aerospike.Client.query()* is used for executing queries over a secondary index of a specified set (which can be omitted or None). For queries, the None set contains those records which are not part of any named set.

The Query can (optionally) be assigned one of the *predicates* (*between()* or *equals()*) using *where()*. A query without a predicate will match all the records in the given set, similar to a *Scan*.

The query is invoked using either *foreach()* or *results()*. The bins returned can be filtered by using *select()*.

Finally, a stream UDF may be applied with *apply()*. It will aggregate results out of the records streaming back from the query.

See also:

Queries and Managing Queries.

select (*bin1*[, *bin2*[, *bin3*..]])
Set a filter on the record bins resulting from *results()* or *foreach()*. If a selected bin does not exist in a record it will not appear in the *bins* portion of that record tuple.

where (*predicate*)
Set a where *predicate* for the query, without which the query will behave similar to *aerospike.Scan*. The predicate is produced by one of the *aerospike.predicates* methods *equals()* and *between()*.

Parameters **predicate** (*tuple*) – the `tuple()` produced by one of the *aerospike.predicates* methods.

Note: Currently, you can assign at most one predicate to the query.

---

results (*[,policy [, options]]) -> list of (key, meta, bins*)
Buffer the records resulting from the query, and return them as a `list` of records.

Parameters

- **policy** (*dict*) – optional *Query Policies*.
- **options** (*dict*) – optional *Query Options*.

Returns a `list` of *Record Tuple*.

```
import aerospike
from aerospike import predicates as p
import pprint

config = { 'hosts': [ ('127.0.0.1', 3000)]}
client = aerospike.client(config).connect()

pp = pprint.PrettyPrinter(indent=2)
query = client.query('test', 'demo')
query.select('name', 'age') # matched records return with the values of these␣
↪bins
# assuming there is a secondary index on the 'age' bin of test.demo
query.where(p.equals('age', 40))
records = query.results( {'total_timeout':2000})
pp.pprint(records)
client.close()
```

Note: Queries require a secondary index to exist on the *bin* being queried.

---

foreach (*callback*[, *policy*[, *options*]])
Invoke the *callback* function for each of the records streaming back from the query.

Parameters

- **callback** (*callable*) – the function to invoke for each record.
- **policy** (*dict*) – optional *Query Policies*.

- **options** (*dict*) – optional *Query Options*.

Note: A *Record Tuple* is passed as the argument to the callback function.

```
import aerospike
from aerospike import predicates as p
import pprint

config = { 'hosts': [ ('127.0.0.1', 3000)]}
client = aerospike.client(config).connect()

pp = pprint.PrettyPrinter(indent=2)
query = client.query('test', 'demo')
query.select('name', 'age') # matched records return with the values of these␣
↪bins
# assuming there is a secondary index on the 'age' bin of test.demo
query.where(p.between('age', 20, 30))
names = []
def matched_names((key, metadata, bins)):
    pp.pprint(bins)
    names.append(bins['name'])

query.foreach(matched_names, {'total_timeout':2000})
pp.pprint(names)
client.close()
```

Note: To stop the stream return `False` from the callback function.

```
from __future__ import print_function
import aerospike
from aerospike import predicates as p

config = { 'hosts': [ ('127.0.0.1',3000)]}
client = aerospike.client(config).connect()

def limit(lim, result):
    c = [0] # integers are immutable so a list (mutable) is used for the␣
↪counter
    def key_add((key, metadata, bins)):
        if c[0] < lim:
            result.append(key)
            c[0] = c[0] + 1
        else:
            return False
    return key_add

query = client.query('test','user')
query.where(p.between('age', 20, 30))
keys = []
query.foreach(limit(100, keys))
print(len(keys)) # this will be 100 if the number of matching records > 100
client.close()
```

apply(*module*, *function*[, *arguments*])

Aggregate the *results()* using a stream UDF. If no predicate is attached to the *Query* the stream UDF

will aggregate over all the records in the specified set.

Parameters

- **module** (*str*) – the name of the Lua module.
- **function** (*str*) – the name of the Lua function within the *module*.
- **arguments** (*list*) – optional arguments to pass to the *function*.

Returns one of the supported types, `int`, `str`, `float` (double), `list`, `dict` (map), `bytearray` (bytes).

See also:

Developing Stream UDFs

Note: Assume we registered the following Lua module with the cluster as **stream_udf.lua** using *aerospike.Client.udf_put()*.

```
local function having_ge_threshold(bin_having, ge_threshold)
    return function(rec)
        debug("group_count::thresh_filter: %s >  %s ?", tostring(rec[bin_
↪having]), tostring(ge_threshold))
        if rec[bin_having] < ge_threshold then
            return false
        end
        return true
    end
end

local function count(group_by_bin)
  return function(group, rec)
    if rec[group_by_bin] then
      local bin_name = rec[group_by_bin]
      group[bin_name] = (group[bin_name] or 0) + 1
      debug("group_count::count: bin %s has value %s which has the count of %s
↪", tostring(bin_name), tostring(group[bin_name]))
    end
    return group
  end
end

local function add_values(val1, val2)
  return val1 + val2
end

local function reduce_groups(a, b)
  return map.merge(a, b, add_values)
end

function group_count(stream, group_by_bin, bin_having, ge_threshold)
  if bin_having and ge_threshold then
    local myfilter = having_ge_threshold(bin_having, ge_threshold)
    return stream : filter(myfilter) : aggregate(map{}, count(group_by_bin))␣
↪: reduce(reduce_groups)
  else
    return stream : aggregate(map{}, count(group_by_bin)) : reduce(reduce_
↪groups)
  end
```

```
end
```

Find the first name distribution of users in their twenties using a query aggregation:

```
import aerospike
from aerospike import predicates as p
import pprint

config = {'hosts': [('127.0.0.1', 3000)],
          'lua': {'system_path':'/usr/local/aerospike/lua/',
                  'user_path':'/usr/local/aerospike/usr-lua/'}}
client = aerospike.client(config).connect()

pp = pprint.PrettyPrinter(indent=2)
query = client.query('test', 'users')
query.where(p.between('age', 20, 29))
query.apply('stream_udf', 'group_count', [ 'first_name' ])
names = query.results()
# we expect a dict (map) whose keys are names, each with a count value
pp.pprint(names)
client.close()
```

With stream UDFs, the final reduce steps (which ties the results from the reducers of the cluster nodes) executes on the client-side. Explicitly setting the Lua `user_path` in the config helps the client find the local copy of the module containing the stream UDF. The `system_path` is constructed when the Python package is installed, and contains system modules such as `aerospike.lua`, `as.lua`, and `stream_ops.lua`. The default value for the Lua `system_path` is `/usr/local/aerospike/lua`.

Query Policies

policy
A `dict` of optional query policies which are applicable to *`Query.results()`* and *`Query.foreach()`*. See *Policies*.

- **max_retries**
  An `int`. Maximum number of retries before aborting the current transaction. The initial attempt is not counted as a retry.

  If max_retries is exceeded, the transaction will return error `AEROSPIKE_ERR_TIMEOUT`.

  WARNING: Database writes that are not idempotent (such as "add") should not be retried because the write operation may be performed multiple times
  if the client timed out previous transaction attempts. It's important to use a distinct write policy for non-idempotent writes which sets max_retries = *0*;

  Default: `0`
- **sleep_between_retries**
  An `int`. Milliseconds to sleep between retries. Enter zero to skip sleep. Default: `0`
- **socket_timeout**
  An `int`. Socket idle timeout in milliseconds when processing a database command.

If socket_timeout is not zero and the socket has been idle for at least socket_timeout, both max_retries and total_timeout are checked. If max_retries and total_timeout are not exceeded, the transaction is retried.

If both `socket_timeout` and `total_timeout` are non-zero and `socket_timeout` > `total_timeout`, then `socket_timeout` will be set to `total_timeout`. If `socket_timeout` is zero, there will be no socket idle limit.

Default: `10000`.

- **total_timeout**
  An `int`. Total transaction timeout in milliseconds.

  The total_timeout is tracked on the client and sent to the server along with the transaction in the wire protocol. The client will most likely timeout first, but the server also has the capability to timeout the transaction.

  If `total_timeout` is not zero and `total_timeout` is reached before the transaction completes, the transaction will return error `AEROSPIKE_ERR_TIMEOUT`. If `total_timeout` is zero, there will be no total time limit.

  Default: `0`
- **deserialize**
  `bool` Should raw bytes representing a list or map be deserialized to a list or dictionary.
  Set to *False* for backup programs that just need access to raw bytes.
  Default: `True`

Query Options

options
A `dict` of optional scan options which are applicable to *`Query.foreach()`* and *`Query.results()`*.

- **nobins** `bool` whether to return the *bins* portion of the *Record Tuple*. Default `False`.

New in version 3.0.0.

1.6 `aerospike.predicates` — Query Predicates

`aerospike.predicates.`**between**(*bin*, *min*, *max*)
Represent a *bin* **BETWEEN** *min* **AND** *max* predicate.

Parameters

- **bin** (*str*) – the bin name.
- **min** (*int*) – the minimum value to be matched with the between operator.
- **max** (*int*) – the maximum value to be matched with the between operator.

Returns `tuple()` to be used in *`aerospike.Query.where()`*.

```
from __future__ import print_function
import aerospike
from aerospike import predicates as p
```

```
config = { 'hosts': [ ('127.0.0.1', 3000)]}
client = aerospike.client(config).connect()
query = client.query('test', 'demo')
query.where(p.between('age', 20, 30))
res = query.results()
print(res)
client.close()
```

aerospike.predicates.**equals**(*bin*, *val*)

Represent a *bin* = *val* predicate.

Parameters

- **bin** (*str*) – the bin name.
- **val** (str or int) – the value to be matched with an equals operator.

Returns tuple() to be used in *aerospike.Query.where()*.

```
from __future__ import print_function
import aerospike
from aerospike import predicates as p

config = { 'hosts': [ ('127.0.0.1', 3000)]}
client = aerospike.client(config).connect()
query = client.query('test', 'demo')
query.where(p.equals('name', 'that guy'))
res = query.results()
print(res)
client.close()
```

aerospike.predicates.**geo_within_geojson_region**(*bin*, *shape*[, *index_type*])

Predicate for finding any point in bin which is within the given shape. Requires a geo2dsphere index (*index_geo2dsphere_create()*) over a *bin* containing *GeoJSON* point data.

Parameters

- **bin** (*str*) – the bin name.
- **shape** (*str*) – the shape formatted as a GeoJSON string.
- **index_type** – Optional. Possible aerospike.INDEX_TYPE_* values are detailed in *Miscellaneous*.

Returns tuple() to be used in *aerospike.Query.where()*.

Note: Requires server version >= 3.7.0

```
from __future__ import print_function
import aerospike
from aerospike import GeoJSON
from aerospike import predicates as p

config = { 'hosts': [ ('127.0.0.1', 3000)]}
client = aerospike.client(config).connect()

client.index_geo2dsphere_create('test', 'pads', 'loc', 'pads_loc_geo')
bins = {'pad_id': 1,
        'loc': aerospike.geojson('{"type":"Point", "coordinates":[-80.604333, 28.
↪608389]}')}
```

```
client.put(('test', 'pads', 'launchpad1'), bins)

# Create a search rectangle which matches screen boundaries:
# (from the bottom left corner counter-clockwise)
scrn = GeoJSON({ 'type': "Polygon",
                 'coordinates': [
                  [[-80.590000, 28.60000],
                   [-80.590000, 28.61800],
                   [-80.620000, 28.61800],
                   [-80.620000, 28.60000],
                   [-80.590000, 28.60000]]]})

# Find all points contained in the rectangle.
query = client.query('test', 'pads')
query.select('pad_id', 'loc')
query.where(p.geo_within_geojson_region('loc', scrn.dumps()))
records = query.results()
print(records)
client.close()
```

New in version 1.0.58.

aerospike.predicates.**geo_within_radius**(*bin*, *long*, *lat*, *radius_meters*[, *index_type*])

Predicate helper builds an AeroCircle GeoJSON shape, and returns a 'within GeoJSON region' predicate. Requires a geo2dsphere index (*index_geo2dsphere_create()*) over a *bin* containing *GeoJSON* point data.

Parameters

- **bin** (*str*) – the bin name.
- **long** (*float*) – the longitude of the center point of the AeroCircle.
- **lat** (*float*) – the latitude of the center point of the AeroCircle.
- **radius_meters** (*float*) – the radius length in meters of the AeroCircle.
- **index_type** – Optional. Possible aerospike.INDEX_TYPE_* values are detailed in *Miscellaneous*.

Returns tuple() to be used in *aerospike.Query.where()*.

Note: Requires server version >= 3.7.0

```
from __future__ import print_function
import aerospike
from aerospike import GeoJSON
from aerospike import predicates as p

config = { 'hosts': [ ('127.0.0.1', 3000)]}
client = aerospike.client(config).connect()

client.index_geo2dsphere_create('test', 'pads', 'loc', 'pads_loc_geo')
bins = {'pad_id': 1,
        'loc': aerospike.geojson('{"type":"Point", "coordinates":[-80.604333, 28.
↪608389]}')}
client.put(('test', 'pads', 'launchpad1'), bins)

query = client.query('test', 'pads')
```

```
query.select('pad_id', 'loc')
query.where(p.geo_within_radius('loc', -80.605000, 28.60900, 400.0))
records = query.results()
print(records)
client.close()
```

New in version 1.0.58.

aerospike.predicates.**geo_contains_geojson_point**(*bin*, *point*[, *index_type*])

Predicate for finding any regions in the bin which contain the given point. Requires a geo2dsphere index (*index_geo2dsphere_create()*) over a *bin* containing *GeoJSON* point data.

Parameters

- **bin** (*str*) – the bin name.
- **point** (*str*) – the point formatted as a GeoJSON string.
- **index_type** – Optional. Possible aerospike.INDEX_TYPE_* values are detailed in *Miscellaneous*.

Returns tuple() to be used in *aerospike.Query.where()*.

Note: Requires server version >= 3.7.0

```
from __future__ import print_function
import aerospike
from aerospike import GeoJSON
from aerospike import predicates as p

config = { 'hosts': [ ('127.0.0.1', 3000)]}
client = aerospike.client(config).connect()

client.index_geo2dsphere_create('test', 'launch_centers', 'area', 'launch_area_geo
↪')
rect = GeoJSON({ 'type': "Polygon",
                 'coordinates': [
                  [[-80.590000, 28.60000],
                   [-80.590000, 28.61800],
                   [-80.620000, 28.61800],
                   [-80.620000, 28.60000],
                   [-80.590000, 28.60000]]]})
bins = {'area': rect}
client.put(('test', 'launch_centers', 'kennedy space center'), bins)

# Find all geo regions containing a point
point = GeoJSON({'type': "Point",
                 'coordinates': [-80.604333, 28.608389]})
query = client.query('test', 'launch_centers')
query.where(p.geo_contains_geojson_point('area', point.dumps()))
records = query.results()
print(records)
client.close()
```

New in version 1.0.58.

aerospike.predicates.**geo_contains_point**(*bin*, *long*, *lat*[, *index_type*])

Predicate helper builds a GeoJSON point, and returns a 'contains GeoJSON point' predicate. Requires a

geo2dsphere index (*index_geo2dsphere_create()*) over a *bin* containing *GeoJSON* point data.

Parameters

- **bin** (*str*) – the bin name.
- **long** (*float*) – the longitude of the point.
- **lat** (*float*) – the latitude of the point.
- **index_type** – Optional. Possible `aerospike.INDEX_TYPE_*` values are detailed in *Miscellaneous*.

Returns `tuple()` to be used in *aerospike.Query.where()*.

Note: Requires server version >= 3.7.0

```
from __future__ import print_function
import aerospike
from aerospike import GeoJSON
from aerospike import predicates as p

config = { 'hosts': [ ('127.0.0.1', 3000)]}
client = aerospike.client(config).connect()

client.index_geo2dsphere_create('test', 'launch_centers', 'area', 'launch_area_geo
↪')
rect = GeoJSON({ 'type': "Polygon",
                 'coordinates': [
                  [[-80.590000, 28.60000],
                   [-80.590000, 28.61800],
                   [-80.620000, 28.61800],
                   [-80.620000, 28.60000],
                   [-80.590000, 28.60000]]]})
bins = {'area': rect}
client.put(('test', 'launch_centers', 'kennedy space center'), bins)

# Find all geo regions containing a point
query = client.query('test', 'launch_centers')
query.where(p.geo_contains_point('area', -80.604333, 28.608389))
records = query.results()
print(records)
client.close()
```

New in version 1.0.58.

`aerospike.predicates.`**contains**(*bin*, *index_type*, *val*)

Represent the predicate *bin* **CONTAINS** *val* for a bin with a complex (list or map) type.

Parameters

- **bin** (*str*) – the bin name.
- **index_type** – Possible `aerospike.INDEX_TYPE_*` values are detailed in *Miscellaneous*.
- **val** (`str` or `int`) – match records whose *bin* is an *index_type* (ex: list) containing *val*.

Returns `tuple()` to be used in *aerospike.Query.where()*.

Warning: This functionality will become production-ready in a future release of the Aerospike server.

```
from __future__ import print_function
import aerospike
from aerospike import predicates as p

config = { 'hosts': [ ('127.0.0.1', 3000)]}
client = aerospike.client(config).connect()

# assume the bin fav_movies in the set test.demo bin should contain
# a dict { (str) _title_ : (int) _times_viewed_ }
# create a secondary index for string values of test.demo records whose 'fav_
↪movies' bin is a map
client.index_map_keys_create('test', 'demo', 'fav_movies', aerospike.INDEX_STRING,
↪ 'demo_fav_movies_titles_idx')
# create a secondary index for integer values of test.demo records whose 'fav_
↪movies' bin is a map
client.index_map_values_create('test', 'demo', 'fav_movies', aerospike.INDEX_
↪NUMERIC, 'demo_fav_movies_views_idx')

client.put(('test','demo','Dr. Doom'), {'age':43, 'fav_movies': {'12 Monkeys': 1,
↪'Brasil': 2}})
client.put(('test','demo','The Hulk'), {'age':38, 'fav_movies': {'Blindness': 1,
↪'Eternal Sunshine': 2}})

query = client.query('test', 'demo')
query.where(p.contains('fav_movies', aerospike.INDEX_TYPE_MAPKEYS, '12 Monkeys'))
res = query.results()
print(res)
client.close()
```

aerospike.predicates.**range**(*bin*, *index_type*, *min*, *max*)

Represent the predicate *bin* **CONTAINS** values **BETWEEN** *min* **AND** *max* for a bin with a complex (list or map) type.

Parameters

- **bin** (*str*) – the bin name.
- **index_type** – Possible `aerospike.INDEX_TYPE_*` values are detailed in *Miscellaneous*.
- **min** (*int*) – the minimum value to be used for matching with the range operator.
- **max** (*int*) – the maximum value to be used for matching with the range operator.

Returns `tuple()` to be used in *aerospike.Query.where()*.

Warning: This functionality will become production-ready in a future release of the Aerospike server.

```
from __future__ import print_function
import aerospike
from aerospike import predicates as p

config = { 'hosts': [ ('127.0.0.1', 3000)]}
client = aerospike.client(config).connect()
```

```
# create a secondary index for numeric values of test.demo records whose 'age'␣
↪bin is a list
client.index_list_create('test', 'demo', 'age', aerospike.INDEX_NUMERIC, 'demo_
↪age_nidx')

# query for records whose 'age' bin has a list with numeric values between 20 and␣
↪30
query = client.query('test', 'demo')
query.where(p.range('age', aerospike.INDEX_TYPE_LIST, 20, 30))
res = query.results()
print(res)
client.close()
```

1.7 GeoJSON Class — GeoJSON

1.7.1 GeoJSON

class aerospike.**GeoJSON**

Starting with version 3.7.0, the Aerospike server supports storing GeoJSON data. A Geo2DSphere index can be built on a bin which contains GeoJSON data, enabling queries for the points contained within given shapes using *geo_within_geojson_region()* and *geo_within_radius()*, and for the regions which contain a point using *geo_contains_geojson_point()* and *geo_contains_point()*.

On the client side, wrapping geospatial data in an instance of the *aerospike.GeoJSON* class enables serialization of the data into the correct type during write operation, such as *put()*. On reading a record from the server, bins with geospatial data it will be deserialized into a *GeoJSON* instance.

See also:

Geospatial Index and Query.

```
from __future__ import print_function
import aerospike
from aerospike import GeoJSON

config = { 'hosts': [ ('127.0.0.1', 3000)]}
client = aerospike.client(config).connect()
client.index_geo2dsphere_create('test', 'pads', 'loc', 'pads_loc_geo')
# Create GeoJSON point using WGS84 coordinates.
latitude = 28.608389
longitude = -80.604333
loc = GeoJSON({'type': "Point",
               'coordinates': [longitude, latitude]})
print(loc)
# Alternatively create the GeoJSON point from a string
loc = aerospike.geojson('{"type": "Point", "coordinates": [-80.604333, 28.608389]}
↪')

# Create a user record.
bins = {'pad_id': 1,
        'loc': loc}

# Store the record.
client.put(('test', 'pads', 'launchpad1'), bins)
```

```
# Read the record.
(k, m, b) = client.get(('test', 'pads', 'launchpad1'))
print(b)
client.close()
```

class GeoJSON([*geo_data*])
Optionally initializes an object with a GeoJSON `str` or a `dict` of geospatial data.

wrap(*geo_data*)
Sets the geospatial data of the *GeoJSON* wrapper class.

Parameters **geo_data** (*dict*) – a `dict` representing the geospatial data.

unwrap() → dict of geospatial data
Gets the geospatial data contained in the *GeoJSON* class.

Returns a `dict` representing the geospatial data.

loads(*raw_geo*)
Sets the geospatial data of the *GeoJSON* wrapper class from a GeoJSON string.

Parameters **raw_geo** (*str*) – a GeoJSON string representation.

dumps() → a GeoJSON string
Gets the geospatial data contained in the *GeoJSON* class as a GeoJSON string.

Returns a GeoJSON `str` representing the geospatial data.

New in version 1.0.53.

1.8 aerospike.exception — Aerospike Exceptions

```
from __future__ import print_function

import aerospike
from aerospike.exception import *

try:
    config = { 'hosts': [ ('127.0.0.1', 3000)], 'policies': { 'total_timeout': 1200}}
    client = aerospike.client(config).connect()
    client.close()
except ClientError as e:
    print("Error: {0} [{1}]".format(e.msg, e.code))
```

New in version 1.0.44.

exception aerospike.exception.**AerospikeError**
The parent class of all exceptions raised by the Aerospike client, inherits from `exceptions.Exception`

code
The associated status code.

msg
The human-readable error message.

file

line

exception aerospike.exception.**ClientError**
Exception class for client-side errors, often due to mis-configuration or misuse of the API methods. Subclass of *AerospikeError*.

exception aerospike.exception.**InvalidHostError**
Subclass of *ClientError*.

exception aerospike.exception.**ParamError**
Subclass of *ClientError*.

exception aerospike.exception.**ServerError**
The parent class for all errors returned from the cluster.

exception aerospike.exception.**InvalidRequest**
Protocol-level error. Subclass of *ServerError*.

exception aerospike.exception.**ServerFull**
The server node is running out of memory and/or storage device space reserved for the specified namespace. Subclass of *ServerError*.

exception aerospike.exception.**NoXDR**
XDR is not available for the cluster. Subclass of *ServerError*.

exception aerospike.exception.**UnsupportedFeature**
Encountered an unimplemented server feature. Subclass of *ServerError*.

exception aerospike.exception.**DeviceOverload**
The server node's storage device(s) can't keep up with the write load. Subclass of *ServerError*.

exception aerospike.exception.**NamespaceNotFound**
Namespace in request not found on server. Subclass of *ServerError*.

exception aerospike.exception.**ForbiddenError**
Operation not allowed at this time. Subclass of *ServerError*.

exception aerospike.exception.**RecordError**
The parent class for record and bin exceptions exceptions associated with read and write operations. Subclass of *ServerError*.

key
The key identifying the record.

bin
Optionally the bin associated with the error.

exception aerospike.exception.**RecordKeyMismatch**
Record key sent with transaction did not match key stored on server. Subclass of *RecordError*.

exception aerospike.exception.**RecordNotFound**
Record does not exist in database. May be returned by read, or write with policy *aerospike.POLICY_EXISTS_UPDATE*. Subclass of *RecordError*.

exception aerospike.exception.**RecordGenerationError**
Generation of record in database does not satisfy write policy. Subclass of *RecordError*.

exception aerospike.exception.**RecordGenerationError**
Record already exists. May be returned by write with policy *aerospike.POLICY_EXISTS_CREATE*. Subclass of *RecordError*.

exception aerospike.exception.**RecordBusy**
Record being (re-)written can't fit in a storage write block. Subclass of *RecordError*.

exception aerospike.exception.**RecordTooBig**
Too may concurrent requests for one record - a "hot-key" situation. Subclass of *RecordError*.

exception aerospike.exception.**BinNameError**
Length of bin name exceeds the limit of 14 characters. Subclass of *RecordError*.

exception aerospike.exception.**BinExistsError**
Bin already exists. Occurs only if the client has that check enabled. Subclass of *RecordError*.

exception aerospike.exception.**BinNotFound**
Bin-level replace-only supported on server but not on client. Subclass of *RecordError*.

exception aerospike.exception.**BinIncompatibleType**
Bin modification operation can't be done on an existing bin due to its value type (for example appending to an integer). Subclass of *RecordError*.

exception aerospike.exception.**IndexError**
The parent class for indexing exceptions. Subclass of *ServerError*.

index_name
The name of the index associated with the error.

exception aerospike.exception.**IndexNotFound**
Subclass of *IndexError*.

exception aerospike.exception.**IndexFoundError**
Subclass of *IndexError*.

exception aerospike.exception.**IndexOOM**
The index is out of memory. Subclass of *IndexError*.

exception aerospike.exception.**IndexNotReadable**
Subclass of *IndexError*.

exception aerospike.exception.**IndexNameMaxLen**
Subclass of *IndexError*.

exception aerospike.exception.**IndexNameMaxCount**
Reached the maximum allowed number of indexes. Subclass of *IndexError*.

exception aerospike.exception.**QueryError**
Exception class for query errors. Subclass of *AerospikeError*.

exception aerospike.exception.**QueryQueueFull**
Subclass of *QueryError*.

exception aerospike.exception.**QueryTimeout**
Subclass of *QueryError*.

exception aerospike.exception.**ClusterError**
Cluster discovery and connection errors. Subclass of *AerospikeError*.

exception aerospike.exception.**ClusterChangeError**
A cluster state change occurred during the request. This may also be returned by scan operations with the fail-on-cluster-change flag set. Subclass of *ClusterError*.

exception aerospike.exception.**AdminError**
The parent class for exceptions of the security API.

exception aerospike.exception.**ExpiredPassword**
Subclass of *AdminError*.

exception aerospike.exception.**ForbiddenPassword**
Subclass of *AdminError*.

exception aerospike.exception.**IllegalState**
Subclass of *AdminError*.

exception aerospike.exception.**InvalidCommand**
Subclass of *AdminError*.

exception aerospike.exception.**InvalidCredential**
Subclass of *AdminError*.

exception aerospike.exception.**InvalidField**
Subclass of *AdminError*.

exception aerospike.exception.**InvalidPassword**
Subclass of *AdminError*.

exception aerospike.exception.**InvalidPrivilege**
Subclass of *AdminError*.

exception aerospike.exception.**InvalidRole**
Subclass of *AdminError*.

exception aerospike.exception.**InvalidUser**
Subclass of *AdminError*.

exception aerospike.exception.**NotAuthenticated**
Subclass of *AdminError*.

exception aerospike.exception.**RoleExistsError**
Subclass of *AdminError*.

exception aerospike.exception.**RoleViolation**
Subclass of *AdminError*.

exception aerospike.exception.**SecurityNotEnabled**
Subclass of *AdminError*.

exception aerospike.exception.**SecurityNotSupported**
Subclass of *AdminError*.

exception aerospike.exception.**SecuritySchemeNotSupported**
Subclass of *AdminError*.

exception aerospike.exception.**UserExistsError**
Subclass of *AdminError*.

exception aerospike.exception.**UDFError**
The parent class for UDF exceptions exceptions. Subclass of *ServerError*.

module
The UDF module associated with the error.

func
Optionally the name of the UDF function.

exception aerospike.exception.**UDFNotFound**
Subclass of *UDFError*.

exception aerospike.exception.**LuaFileNotFound**
Subclass of *UDFError*.

exception aerospike.exception.**LDTError**
The parent class for Large Data Type exceptions. Subclass of *ServerError*.

key
The key identifying the record.

bin
The bin containing the LDT.

exception aerospike.exception.**LargeItemNotFound**
Subclass of *LDTError*.

exception aerospike.exception.**LDTInternalError**
Subclass of *LDTError*.

exception aerospike.exception.**LDTNotFound**
Subclass of *LDTError*.

exception aerospike.exception.**LDTUniqueKeyError**
Subclass of *LDTError*.

exception aerospike.exception.**LDTInsertError**
Subclass of *LDTError*.

exception aerospike.exception.**LDTSearchError**
Subclass of *LDTError*.

exception aerospike.exception.**LDTDeleteError**
Subclass of *LDTError*.

exception aerospike.exception.**LDTInputParamError**
Subclass of *LDTError*.

exception aerospike.exception.**LDTTypeMismatch**
Subclass of *LDTError*.

exception aerospike.exception.**LDTBinNameNull**
Subclass of *LDTError*.

exception aerospike.exception.**LDTBinNameNotString**
Subclass of *LDTError*.

exception aerospike.exception.**LDTBinNameTooLong**
Subclass of *LDTError*.

exception aerospike.exception.**LDTTooManyOpenSubrecs**
Subclass of *LDTError*.

exception aerospike.exception.**LDTTopRecNotFound**
Subclass of *LDTError*.

exception aerospike.exception.**LDTSubRecNotFound**
Subclass of *LDTError*.

exception aerospike.exception.**LDTBinNotFound**
Subclass of *LDTError*.

exception aerospike.exception.**LDTBinExistsError**
Subclass of *LDTError*.

exception aerospike.exception.**LDTBinDamaged**
Subclass of *LDTError*.

exception aerospike.exception.**LDTSubrecPoolDamaged**
Subclass of *LDTError*.

exception aerospike.exception.**LDTSubrecDamaged**
Subclass of *LDTError*.

exception aerospike.exception.**LDTSubrecOpenError**
Subclass of *LDTError*.

exception aerospike.exception.**LDTSubrecUpdateError**
Subclass of *LDTError*.

exception aerospike.exception.**LDTSubrecCreateError**
Subclass of *LDTError*.

exception aerospike.exception.**LDTSubrecDeleteError**
Subclass of *LDTError*.

exception aerospike.exception.**LDTSubrecCloseError**
Subclass of *LDTError*.

exception aerospike.exception.**LDTToprecUpdateError**
Subclass of *LDTError*.

exception aerospike.exception.**LDTToprecCreateError**
Subclass of *LDTError*.

exception aerospike.exception.**LDTFilterFunctionBad**
Subclass of *LDTError*.

exception aerospike.exception.**LDTFilterFunctionNotFound**
Subclass of *LDTError*.

exception aerospike.exception.**LDTKeyFunctionBad**
Subclass of *LDTError*.

exception aerospike.exception.**LDTKeyFunctionNotFound**
Subclass of *LDTError*.

exception aerospike.exception.**LDTTransFunctionBad**
Subclass of *LDTError*.

exception aerospike.exception.**LDTTransFunctionNotFound**
Subclass of *LDTError*.

exception aerospike.exception.**LDTUntransFunctionBad**
Subclass of *LDTError*.

exception aerospike.exception.**LDTUntransFunctionNotFound**
Subclass of *LDTError*.

exception aerospike.exception.**LDTUserModuleBad**
Subclass of *LDTError*.

exception aerospike.exception.**LDTUserModuleNotFound**
Subclass of *LDTError*.

1.8.1 Exception Hierarchy

```
AerospikeError (*)
 +-- TimeoutError (9)
 +-- ClientError (-1)
 |    +-- InvalidHost (-4)
 |    +-- ParamError (-2)
```

```
+-- ServerError (1)
     +-- InvalidRequest (4)
     +-- ServerFull (8)
     +-- NoXDR (10)
     +-- UnsupportedFeature (16)
     +-- DeviceOverload (18)
     +-- NamespaceNotFound (20)
     +-- ForbiddenError (22)
     +-- RecordError (*)
     |    +-- RecordKeyMismatch (19)
     |    +-- RecordNotFound (2)
     |    +-- RecordGenerationError (3)
     |    +-- RecordExistsError (5)
     |    +-- RecordTooBig (13)
     |    +-- RecordBusy (14)
     |    +-- BinNameError (21)
     |    +-- BinExistsError (6)
     |    +-- BinNotFound (17)
     |    +-- BinIncompatibleType (12)
     +-- IndexError (204)
     |    +-- IndexNotFound (201)
     |    +-- IndexFoundError (200)
     |    +-- IndexOOM (202)
     |    +-- IndexNotReadable (203)
     |    +-- IndexNameMaxLen (205)
     |    +-- IndexNameMaxCount (206)
     +-- QueryError (213)
     |    +-- QueryQueueFull (211)
     |    +-- QueryTimeout (212)
     +-- ClusterError (11)
     |    +-- ClusterChangeError (7)
     +-- AdminError (*)
     |    +-- SecurityNotSupported (51)
     |    +-- SecurityNotEnabled (52)
     |    +-- SecuritySchemeNotSupported (53)
     |    +-- InvalidCommand (54)
     |    +-- InvalidField (55)
     |    +-- IllegalState (56)
     |    +-- InvalidUser (60)
     |    +-- UserExistsError (61)
     |    +-- InvalidPassword (62)
     |    +-- ExpiredPassword (63)
     |    +-- ForbiddenPassword (64)
     |    +-- InvalidCredential (65)
     |    +-- InvalidRole (70)
     |    +-- RoleExistsError (71)
     |    +-- RoleViolation (81)
     |    +-- InvalidPrivilege (72)
     |    +-- NotAuthenticated (80)
     +-- UDFError (*)
     |    +-- UDFNotFound (1301)
     |    +-- LuaFileNotFound (1302)
     +-- LDTError (*)
          +-- LargeItemNotFound (125)
          +-- LDTInternalError (1400)
          +-- LDTNotFound (1401)
          +-- LDTUniqueKeyError (1402)
          +-- LDTInsertError (1403)
```

```
          +-- LDTSearchError (1404)
          +-- LDTDeleteError (1405)
          +-- LDTInputParamError (1409)
          +-- LDTTypeMismatch (1410)
          +-- LDTBinNameNull (1411)
          +-- LDTBinNameNotString (1412)
          +-- LDTBinNameTooLong (1413)
          +-- LDTTooManyOpenSubrecs (1414)
          +-- LDTTopRecNotFound (1415)
          +-- LDTSubRecNotFound (1416)
          +-- LDTBinNotFound (1417)
          +-- LDTBinExistsError (1418)
          +-- LDTBinDamaged (1419)
          +-- LDTSubrecPoolDamaged (1420)
          +-- LDTSubrecDamaged (1421)
          +-- LDTSubrecOpenError (1422)
          +-- LDTSubrecUpdateError (1423)
          +-- LDTSubrecCreateError (1424)
          +-- LDTSubrecDeleteError (1425)
          +-- LDTSubrecCloseError (1426)
          +-- LDTToprecUpdateError (1427)
          +-- LDTToprecCreateError (1428)
          +-- LDTFilterFunctionBad (1430)
          +-- LDTFilterFunctionNotFound (1431)
          +-- LDTKeyFunctionBad (1432)
          +-- LDTKeyFunctionNotFound (1433)
          +-- LDTTransFunctionBad (1434)
          +-- LDTTransFunctionNotFound (1435)
          +-- LDTUntransFunctionBad (1436)
          +-- LDTUntransFunctionNotFound (1437)
          +-- LDTUserModuleBad (1438)
          +-- LDTUserModuleNotFound (1439)
```

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

W