# kt Documentation

***Release 0.7.0***

**charles leifer**

**Nov 21, 2018**

# Contents:

*kt* is a fast client library for use with Kyoto Tycoon and Tokyo Tyrant. *kt* is designed to be performant and simple-to-use.

- Binary protocols implemented as C extension.
- Thread-safe and greenlet-safe.
- Simple APIs.
- Full-featured implementation of protocol.

CHAPTER 1

# Installation

*kt* can be installed using `pip`:

```
$ pip install kt
```

## 1.1 Dependencies

- *cython* - for building the binary protocol C extension.

These libraries are installed automatically if you install *kt* with pip. To install these dependencies manually, run:

```
$ pip install cython
```

## 1.2 Installing with git

To install the latest version with git:

```
$ git clone https://github.com/coleifer/kt
$ cd kt/
$ python setup.py install
```

## 1.3 Installing Kyoto Tycoon or Tokyo Tyrant

If you're using a debian-based linux distribution, you can install using `apt-get`:

```
$ sudo apt-get install kyototycoon tokyotyrant
```

Alternatively you can use the following Docker images:

```
$ docker run -it --rm -v kyoto:/var/lib/kyototycoon -p 1978:1978 coleifer/kyototycoon
$ docker run -it --rm -v tokyo:/var/lib/tokyotyrant -p 9871:9871 coleifer/tokyohash
```

To build from source and read about the various command-line options, see the project documentation:

- Kyoto Tycoon documentation
- Tokyo Tyrant documentation

CHAPTER 2

# Usage

This document describes how to use *kt* with Kyoto Tycoon and Tokyo Tyrant.

## 2.1 Common Features

This section describes features and APIs that are common to both the *KyotoTycoon* client and the *TokyoTyrant* client. For simplicity, we'll use the *EmbeddedServer*, which sets up the database server in a subprocess and makes it easy to develop.

```
>>> from kt import EmbeddedServer
>>> server = EmbeddedServer()
>>> server.run()  # Starts "ktserver" in a subprocess.
True
>>> client = server.client  # Get a client for use with our embedded server.
```

As you would expect for a key/value database, the client implements *get()*, *set()* and *remove()*:

```
>>> client.set('k1', 'v1')
1
>>> client.get('k1')
'v1'
>>> client.remove('k1')
1
```

It is not an error to try to get or delete a key that doesn't exist:

```
>>> client.get('not-here')  # Returns None
>>> client.remove('not-here')
0
```

To check whether a key exists we can use *exists()*:

```
>>> client.set('k1', 'v1')
>>> client.exists('k1')
True
>>> client.exists('not-here')
False
```

In addition, there are also efficient methods for bulk operations: *get_bulk()*, *set_bulk()* and *remove_bulk()*:

```
>>> client.set_bulk({'k1': 'v1', 'k2': 'v2', 'k3': 'v3'})
3
>>> client.get_bulk(['k1', 'k2', 'k3', 'not-here'])
{'k1': 'v1', 'k2': 'v2', 'k3': 'v3'}
>>> client.remove_bulk(['k1', 'k2', 'k3', 'not-here'])
3
```

The client libraries also support a dict-like interface:

```
>>> client['k1'] = 'v1'
>>> print(client['k1'])
v1
>>> del client['k1']
>>> client.update({'k1': 'v1', 'k2': 'v2', 'k3': 'v3'})
3
>>> client.pop('k1')
'v1'
>>> client.pop('k1')  # Returns None
>>> 'k1' in client
False
>>> len(client)
2
```

To remove all records, you can use the *clear()* method:

```
>>> client.clear()
True
```

### 2.1.1 Serialization

By default the client will assume that keys and values should be encoded as UTF-8 byte-strings and decoded to unicode upon retrieval. You can set the `serializer` parameter when creating your client to use a different value serialization. *kt* provides the following:

- `KT_BINARY` - **default**, treat values as unicode and serialize as UTF-8.
- `KT_JSON` - use JSON to serialize values.
- `KT_MSGPACK` - use msgpack to serialize values.
- `KT_PICKLE` - use pickle to serialize values.
- `KT_NONE` - no serialization, values must be bytestrings.

For example, to use the pickle serializer:

```
>>> from kt import KT_PICKLE, KyotoTycoon
>>> client = KyotoTycoon(serializer=KT_PICKLE)
```

(continues on next page)

(continued from previous page)

```
>>> client.set('k1', {'this': 'is', 'a': ['python object']})
1
>>> client.get('k1')
{'this': 'is', 'a': ['python object']}
```

## 2.2 Kyoto Tycoon

The Kyoto Tycoon section continues from the previous section, and assumes that you are running an *EmbeddedServer* and accessing it through it's *client* property.

### 2.2.1 Database filenames

Kyoto Tycoon determines the database type by looking at the filename of the database(s) specified when `ktserver` is executed. Additionally, for in-memory databases, you use special symbols instead of filenames.

- `hash_table.kch` - on-disk hash table ("kch").
- `btree.kct` - on-disk b-tree ("kct").
- `dirhash.kcd` - directory hash ("kcd").
- `dirtree.kcf` - directory b-tree ("kcf").
- `*` - cache-hash, in-memory hash-table with LRU deletion.
- `%` - cache-tree, in-memory b-tree (ordered cache).
- `:` - stash db, in-memory database with lower memory usage.
- `-` - prototype hash, simple in-memory hash using `std::unordered_map`.
- `+` - prototype tree, simple in-memory hash using `std::map` (ordered).

Generally:

- For unordered collections, use either the cache-hash (`*`) or the file-hash (`.kch`).
- For ordered collections or indexes, use either the cache-tree (`%`) or the file b-tree (`.kct`).
- I avoid the prototype hash and btree as the entire data-structure is locked during writes (as opposed to an individual record or page).

For more information about the above database types, their algorithmic complexity, and the unit of locking, see kyoto-cabinet db chart.

### 2.2.2 Key Expiration

Kyoto Tycoon servers feature a built-in expiration mechanism, allowing you to use it as a cache. Whenever setting a value or otherwise writing to the database, you can also specify an expiration time (in seconds):

```
>>> client.set('k1', 'v1', expire_time=5)
>>> client.get('k1')
'v1'
>>> time.sleep(5)
>>> client.get('k1')  # Returns None
```

### 2.2.3 Multiple Databases

Kyoto Tycoon can also be used as the front-end for multiple databases. For example, to start `ktserver` with an in-memory hash-table and an in-memory b-tree, you would run:

```
$ ktserver \* \%
```

By default, the *KyotoTycoon* client assumes you are working with the first database (starting from zero, our hash-table would be `0` and the b-tree would be `1`).

The client can be initialized to use a different database by default:

```
>>> client = KyotoTycoon(default_db=1)
```

To change the default database at run-time, you can call the *set_database()* method:

```
>>> client = KyotoTycoon()
>>> client.set_database(1)
```

Lastly, to perform a one-off operation against a specific database, all methods accept a `db` parameter which you can use to specify the database:

```
>>> client.set('k1', 'v1', db=1)
>>> client.get('k1', db=0)  # Returns None
>>> client.get('k1', db=1)
'v1'
```

Similarly, if a `tuple` is passed into the dictionary APIs, it is assumed that the key consists of `(key, db)` and the value of `(value, expire)`:

```
>>> client['k1', 1] = 'v1'  # Set k1=v1 in db1.
>>> client['k1', 1]
'v1'
>>> client['k2'] = ('v2', 10)  # Set k2=v2 in default db with 10s expiration.
>>> client['k2', 0] = ('v2', 20)  # Set k2=v2 in db0 with 20s expiration.
>>> del client['k1', 1]  # Delete 'k1' in db1.
```

### 2.2.4 Lua Scripts

Kyoto Tycoon can be scripted using lua. To run a Lua script from the client, you can use the *script()* method. In Kyoto Tycoon, a script may receive arbitrary key/value-pairs as parameters, and may return arbitrary key/value pairs:

```
>>> client.script('myfunction', {'key': 'some-key', 'data': 'etc'})
{'data': 'returned', 'by': 'user-script'}
```

To simplify script execution, you can also use the `lua()` helper, which provides a slightly more Pythonic API:

```
>>> lua = client.lua
>>> lua.myfunction(key='some-key', data='etc')
{'data': 'returned', 'by': 'user-script'}
>>> lua.another_function(key='another-key')
{}
```

Learn more about scripting Kyoto Tycoon by reading the lua doc.

## 2.3 Tokyo Tyrant

To experiment with Tokyo Tyrant, an easy way to get started is to use the *EmbeddedTokyoTyrantServer*, which sets up the database server in a subprocess and makes it easy to develop.

```
>>> from kt import EmbeddedTokyoTyrantServer
>>> server = EmbeddedTokyoTyrantServer()
>>> server.run()
True
>>> client = server.client
```

**Note:** Unlike Kyoto Tycoon, the Tokyo Tyrant server process can only embed a single database, and does not support expiration.

### 2.3.1 Database filenames

Tokyo Tyrant determines the database type by looking at the filename of the database(s) specified when `ttserver` is executed. Additionally, for in-memory databases, you use special symbols instead of filenames.

- `hash_table.tch` - on-disk hash table ("tch").
- `btree.tcb` - on-disk b-tree ("tcb").
- `*` - in-memory hash-table.
- `+` - in-memory tree (ordered).

There are two additional database-types, but their usage is beyond the scope of this document:

- `table.tct` - on-disk table database ("tct").
- `table.tcf` - fixed-length database ("tcf").

The table database is neat, as it you can store another layer of key/value pairs in the value field. These key/value pairs are serialized using `0x0` as the delimiter. *TokyoTyrant* provides a special serializer, `TT_TABLE`, which properly handles reading and writing data dictionaries to a table database.

For more information about the above database types, their algorithmic complexity, and the unit of locking, see ttserver documentation.

### 2.3.2 Lua Scripts

Tokyo Tyrant can be scripted using lua. To run a Lua script from the client, you can use the *script()* method. In Tokyo Tyrant, a script may receive a key and a value parameter, and will return a byte-string as a result:

```
>>> client.script('incr', key='counter', value='1')
'1'
>>> client.script('incr', 'counter', '4')
'5'
```

To simplify script execution, you can also use the `lua()` helper, which provides a slightly more Pythonic API:

```
>>> lua = client.lua
>>> lua.incr(key='counter', value='2')
'7'
```

(continues on next page)

(continued from previous page)

```
>>> lua.incr('counter', '1')
'8'
```

Learn more about scripting Tokyo Tyrant by reading the lua docs.

CHAPTER 3

# API

## 3.1 Serializers

**KT_BINARY**
Default value serialization. Serializes values as UTF-8 byte-strings and deserializes to unicode.

**KT_JSON**
Serialize values as JSON (encoded as UTF-8).

**KT_MSGPACK**
Uses `msgpack` to serialize and deserialize values.

**KT_NONE**
No serialization or deserialization. Values must be byte-strings.

**KT_PICKLE**
Serialize and deserialize using Python's pickle module.

**TT_TABLE**
Special serializer for use with TokyoTyrant's remote table database. Values are represented as dictionaries.

## 3.2 Kyoto Tycoon client

**class KyotoTycoon**(*host='127.0.0.1'*, *port=1978*, *serializer=KT_BINARY*, *decode_keys=True*, *timeout=None*, *default_db=0*)

**Parameters**

- **host** (*str*) – server host.
- **port** (*int*) – server port.
- **serializer** – serialization method to use for storing/retrieving values. Accepts `KT_BINARY`, `KT_JSON`, `KT_MSGPACK`, `KT_NONE` or `KT_PICKLE`.
- **decode_keys** (*bool*) – allow unicode keys, encoded as UTF-8.

- **timeout** (*int*) – socket timeout (optional).
- **default_db** (*int*) – default database to operate on.

Client for interacting with Kyoto Tycoon database.

**checkin**()
Return the communication socket to the pool for re-use.

**close**()
Close the connection to the server.

**get**(*key*, *db=None*)

**Parameters**

- **key** (*str*) – key to look-up
- **db** (*int or None*) – database index

**Returns** deserialized value or `None` if key does not exist.

**get_raw**(*key*, *db=None*)

**Parameters**

- **key** (*str*) – key to look-up
- **db** (*int or None*) – database index

**Returns** raw bytestring value or `None` if key does not exist.

**set**(*key*, *value*, *db=None*, *expire_time=None*)

**Parameters**

- **key** (*str*) – key to set
- **value** – value to store (will be serialized using serializer)
- **db** (*int or None*) – database index
- **expire_time** (*int or None*) – expiration time in seconds

**Returns** number of rows set (1)

**remove**(*key*, *db=None*)

**Parameters**

- **key** (*str*) – key to remove
- **db** (*int or None*) – database index

**Returns** number of rows removed

**get_bulk**(*keys*, *db=None*)

**Parameters**

- **keys** (*list*) – list of keys to look-up
- **db** (*int or None*) – database index

**Returns** dictionary of all key/value pairs that were found

**Return type** dict

**get_bulk_raw**(*keys*, *db=None*)

**Parameters**

- **keys** (*list*) – list of keys to look-up
- **db** (*int or None*) – database index

**Returns** dictionary of all key/raw-value pairs that were found

**Return type** dict

**set_bulk**(*__data=None*, *db=None*, *expire_time=None*, ***kwargs*)

**Parameters**

- **__data** (*dict*) – mapping of key/value pairs to set.
- **db** (*int or None*) – database index
- **expire_time** (*int or None*) – expiration time in seconds
- **kwargs** – mapping of key/value pairs to set, expressed as keyword arguments

**Returns** number of keys that were set

**remove_bulk**(*keys*, *db=None*)

**Parameters**

- **keys** (*list*) – list of keys to remove
- **db** (*int or None*) – database index

**Returns** number of keys that were removed

**script**(*name*, *__data=None*, *encode_values=True*, ***kwargs*)

**Parameters**

- **name** (*str*) – name of lua function to call
- **__data** (*dict*) – mapping of key/value pairs to pass to lua function.
- **encode_values** (*bool*) – serialize values passed to lua function.
- **kwargs** – mapping of key/value pairs to pass to lua function, expressed as keyword arguments

**Returns** dictionary of key/value pairs returned by function

**Return type** dict

Execute a lua function. Kyoto Tycoon lua extensions accept arbitrary key/value pairs as input, and return a result dictionary. If `encode_values` is `True`, the input values will be serialized and the result values will be deserialized using the client's serializer.

**clear**(*db=None*)

**Parameters** **db** (*int or None*) – database index

**Returns** boolean indicating success

Remove all keys from the database.

**status**(*db=None*)

**Parameters** **db** (*int or None*) – database index

**Returns** status fields and values

**Return type** dict

Obtain status information from the server about the selected database.

**report**()

**Returns** status fields and values

**Return type** dict

Obtain report on overall status of server, including all databases.

**synchronize**(*hard=False*, *command=None*, *db=None*)

**Parameters**

- **hard** (*bool*) – perform a "hard" synchronization
- **command** (*str*) – command to run after synchronization
- **db** (*int or None*) – database index

**Returns** boolean indicating success

**vacuum**(*step=0*, *db=None*)

**Parameters**

- **step** (*int*) – number of steps, default is 0
- **db** (*int or None*) – database index

**Returns** boolean indicating success

**add**(*key*, *value*, *db=None*, *expire_time=None*)

**Parameters**

- **key** (*str*) – key to add
- **value** – value to store (will be serialized using serializer)
- **db** (*int or None*) – database index
- **expire_time** (*int or None*) – expiration time in seconds

**Returns** boolean indicating if key could be added or not

**Return type** bool

Add a key/value pair to the database. This operation will only succeed if the key does not already exist in the database.

**replace**(*key*, *value*, *db=None*, *expire_time=None*)

**Parameters**

- **key** (*str*) – key to replace
- **value** – value to store (will be serialized using serializer)
- **db** (*int or None*) – database index
- **expire_time** (*int or None*) – expiration time in seconds

**Returns** boolean indicating if key could be replaced or not

**Return type** bool

Replace a key/value pair to the database. This operation will only succeed if the key alreadys exist in the database.

**append**(*key*, *value*, *db=None*, *expire_time=None*)

**Parameters**

- **key** (*str*) – key to append value to
- **value** – data to append (will be serialized using serializer)
- **db** (*int or None*) – database index
- **expire_time** (*int or None*) – expiration time in seconds

**Returns** boolean indicating if value was appended

**Return type** bool

Appends data to an existing key/value pair. If the key does not exist, this is equivalent to *set()*.

**exists**(*key*, *db=None*)

**Parameters**

- **key** (*str*) – key to test
- **db** (*int or None*) – database index

**Returns** boolean indicating if key exists

**Return type** bool

**seize**(*key*, *db=None*)

**Parameters**

- **key** (*str*) – key to remove
- **db** (*int or None*) – database index

**Returns** value stored at given key or `None` if key does not exist.

Get and remove the data stored in a given key.

**cas**(*key*, *old_val*, *new_val*, *db=None*, *expire_time=None*)

**Parameters**

- **key** (*str*) – key to append value to
- **old_val** – original value to test
- **old_val** – new value to store
- **db** (*int or None*) – database index
- **expire_time** (*int or None*) – expiration time in seconds

**Returns** boolean indicating if compare-and-swap succeeded.

**Return type** bool

Compare-and-swap the value stored at a given key.

**incr**(*key*, *n=1*, *orig=None*, *db=None*, *expire_time=None*)

**Parameters**

- **key** (*str*) – key to increment
- **n** (*int*) – value to add
- **orig** (*int*) – default value if key does not exist
- **db** (*int or None*) – database index
- **expire_time** (*int or None*) – expiration time in seconds

**Returns** new value at key

**Return type** int

**incr_double**(*key*, *n=1.*, *orig=None*, *db=None*, *expire_time=None*)

**Parameters**

- **key** (*str*) – key to increment
- **n** (*float*) – value to add
- **orig** (*float*) – default value if key does not exist
- **db** (*int or None*) – database index
- **expire_time** (*int or None*) – expiration time in seconds

**Returns** new value at key

**Return type** float

**__getitem__**(*key_or_keydb*)
Item-lookup based on either `key` or a 2-tuple consisting of `(key, db)`. Follows same semantics as *get()*.

**__setitem__**(*key_or_keydb*, *value_or_valueexpire*)
Item-setting based on either `key` or a 2-tuple consisting of `(key, db)`. Value consists of either a `value` or a 2-tuple consisting of `(value, expire_time)`. Follows same semantics as *set()*.

**__delitem__**(*key_or_keydb*)
Item-deletion based on either `key` or a 2-tuple consisting of `(key, db)`. Follows same semantics as *remove()*.

**__contains__**(*key_or_keydb*)
Check if key exists. Accepts either `key` or a 2-tuple consisting of `(key, db)`. Follows same semantics as *exists()*.

**__len__**()

**Returns** total number of keys in the default database.

**Return type** int

**count**(*db=None*)

**Parameters** **db** (*int or None*) – database index

**Returns** total number of keys in the database.

**Return type** int

Count total number of keys in the database.

**update**(*__data=None*, *db=None*, *expire_time=None*, ***kwargs*)
See *KyotoTycoon.set_bulk()*.

**pop**(*key*, *db=None*)
See *KyotoTycoon.seize()*.

**match_prefix**(*prefix*, *max_keys=None*, *db=None*)

**Parameters**

- **prefix** (*str*) – key prefix to match
- **max_keys** (*int*) – maximum number of results to return (optional)

- **db** (*int or None*) – database index

**Returns** list of keys that matched the given prefix.

**Return type** list

**match_regex**(*regex*, *max_keys=None*, *db=None*)

**Parameters**

- **regex** (*str*) – regular-expression to match
- **max_keys** (*int*) – maximum number of results to return (optional)
- **db** (*int or None*) – database index

**Returns** list of keys that matched the given regular expression.

**Return type** list

**match_similar**(*origin*, *distance=None*, *max_keys=None*, *db=None*)

**Parameters**

- **origin** (*str*) – source string for comparison
- **distance** (*int*) – maximum edit-distance for similarity (optional)
- **max_keys** (*int*) – maximum number of results to return (optional)
- **db** (*int or None*) – database index

**Returns** list of keys that were within a certain edit-distance of origin

**Return type** list

**cursor**(*db=None*, *cursor_id=None*)

**Parameters**

- **db** (*int or None*) – database index
- **cursor_id** (*int or None*) – cursor id (will be automatically created if None)

**Returns** `Cursor` object

**keys**(*db=None*)

**Parameters** **db** (*int or None*) – database index

**Returns** all keys in database

**Return type** generator

**values**(*db=None*)

**Parameters** **db** (*int or None*) – database index

**Returns** all values in database

**Return type** generator

**items**(*db=None*)

**Parameters** **db** (*int or None*) – database index

**Returns** all key/value tuples in database

**Return type** generator

**size**
Property which exposes the size information returned by the *status()* API, for the default database.

**path**
Property which exposes the filename/path returned by the *status()* API, for the default database.

**set_database**(*db*)

**Parameters** **db** (*int*) – database index

Specify the default database for the client.

## 3.3 Tokyo Tyrant client

**class TokyoTyrant**(*host='127.0.0.1'*, *port=1978*, *serializer=KT_BINARY*, *decode_keys=True*, *timeout=None*)

**Parameters**

- **host** (*str*) – server host.
- **port** (*int*) – server port.
- **serializer** – serialization method to use for storing/retrieving values. Accepts KT_BINARY, KT_JSON, KT_MSGPACK, KT_NONE or KT_PICKLE.
- **decode_keys** (*bool*) – allow unicode keys, encoded as UTF-8.
- **timeout** (*int*) – socket timeout (optional).
- **default_db** (*int*) – default database to operate on.

Client for interacting with Tokyo Tyrant database.

**checkin**()
Return the communication socket to the pool for re-use.

**close**()
Close the connection to the server.

**get**(*key*)

**Parameters** **key** (*str*) – key to look-up

**Returns** deserialized value or None if key does not exist.

**get_raw**(*key*)

**Parameters** **key** (*str*) – key to look-up

**Returns** raw binary value or None if key does not exist.

**set**(*key*, *value*)

**Parameters**

- **key** (*str*) – key to set
- **value** – value to store (will be serialized using serializer)

**Returns** boolean indicating success

**remove**(*key*)

**Parameters** **key** (*str*) – key to remove

**Returns** number of rows removed

**get_bulk**(*keys*)

**Parameters** **keys** (*list*) – list of keys to look-up

**Returns** dictionary of all key/value pairs that were found

**Return type** dict

**get_bulk_raw**(*keys*)

**Parameters** **keys** (*list*) – list of keys to look-up

**Returns** dictionary of all key/raw-value pairs that were found

**Return type** dict

**set_bulk**(*__data=None*, ***kwargs*)

**Parameters**

- **__data** (*dict*) – mapping of key/value pairs to set.
- **kwargs** – mapping of key/value pairs to set, expressed as keyword arguments

**Returns** boolean indicating success

**remove_bulk**(*keys*)

**Parameters** **keys** (*list*) – list of keys to remove

**Returns** boolean indicating success

**script**(*name*, *key=None*, *value=None*, *lock_records=False*, *lock_all=False*, *encode_value=True*, *decode_result=False*)

**Parameters**

- **name** (*str*) – name of lua function to call
- **key** (*str*) – key to pass to lua function (optional)
- **value** (*str*) – value to pass to lua function (optional)
- **lock_records** (*bool*) – lock records modified during script execution
- **lock_all** (*bool*) – lock all records during script execution
- **encode_value** (*bool*) – serialize the value before sending to the script
- **decode_result** (*bool*) – deserialize the script return value

**Returns** byte-string or obj returned by function (depending on decode_result)

Execute a lua function. Tokyo Tyrant lua extensions accept two parameters, a key and a value, and return a result byte-string.

**clear**()

**Returns** boolean indicating success

Remove all keys from the database.

**status**()

**Returns** status fields and values

**Return type** dict

Obtain status information from the server.

**add**(*key*, *value*)

**Parameters**

- **key** (*str*) – key to add
- **value** – value to store (will be serialized using serializer)

**Returns** boolean indicating if key could be added or not

**Return type** bool

Add a key/value pair to the database. This operation will only succeed if the key does not already exist in the database.

**append**(*key*, *value*)

**Parameters**

- **key** (*str*) – key to append value to
- **value** – data to append (will be serialized using serializer)

**Returns** boolean indicating if value was appended

**Return type** bool

Appends data to an existing key/value pair. If the key does not exist, this is equivalent to *set()*.

**addshl**(*key*, *value*, *width*)

**Parameters**

- **key** (*str*) – key to append value to
- **value** – data to append (will be serialized using serializer)
- **width** (*int*) – number of bytes to shift

**Returns** boolean indicating success

**Return type** bool

Concatenate a value at the end of the existing record and shift it to the left by *width* bytes.

**setnr**(*key*, *value*)

**Parameters**

- **key** (*str*) – key to set
- **value** – value to store (will be serialized using serializer)

**Returns** no return value

Set with no server response.

**setnr_bulk**(*__data=None*, ***kwargs*)

**Parameters**

- **__data** (*dict*) – mapping of key/value pairs to set.
- **kwargs** – mapping of key/value pairs to set, expressed as keyword arguments

**Returns** no return value

Set multiple key/value pairs using the same no-response API as *TokyoTyrant.setnr()*.

**setdup**(*key*, *value*)

**Parameters**

- **key** (*str*) – key to set
- **value** – value to store (will be serialized using serializer)

**Returns** boolean indicating success

Set key/value pair. If using a BTree and the key already exists, the new value will be added to the end.

**setdupback**(*key*, *value*)

**Parameters**

- **key** (*str*) – key to set
- **value** – value to store (will be serialized using serializer)

**Returns** boolean indicating success

Set key/value pair. If using a BTree and the key already exists, the new value will be added to the front.

**get_part**(*key*, *start=None*, *end=None*)

**Parameters**

- **key** (*str*) – key to look-up
- **start** (*int*) – start offset
- **end** (*int*) – number of characters to retrieve (after start).

**Returns** the substring portion of value requested or `False` if the value does not exist or the start index exceeded the value length.

**exists**(*key*)

**Parameters** **key** (*str*) – key to test

**Returns** boolean indicating if key exists

**Return type** bool

**length**(*key*)

**Parameters** **key** (*str*) – key to test

**Returns** length of value stored at key (or None if key does not exist)

**Return type** int

**incr**(*key*, *n=1*)

**Parameters**

- **key** (*str*) – key to increment
- **n** (*int*) – value to add

**Returns** new value at key

**Return type** int

**incr_double**(*key*, *n=1.*)

**Parameters**

- **key** (*str*) – key to increment
- **n** (*float*) – value to add

**Returns** new value at key

**Return type** float

**misc**(*cmd*, *args=None*, *update_log=True*)

**Parameters**

- **cmd** (*str*) – Command to execute
- **args** (*list*) – Zero or more bytestring arguments to misc function.
- **update_log** (*bool*) – Add misc command to update log.

Run a miscellaneous command using the "misc" API. Returns a list of zero or more bytestrings.

**count**()

**Returns** number of key/value pairs in the database

**Return type** int

**__getitem__**(*key*)
Get value at given `key`. Identical to *get()*.

**Note:** If the database is a tree, a slice of keys can be used to retrieve an ordered range of values.

**__setitem__**(*key*, *value*)
Set value at given `key`. Identical to *set()*.

**__delitem__**(*key*)
Remove the given `key`. Identical to *remove()*.

**__contains__**(*key*)
Check if given `key` exists. Identical to *exists()*.

**__len__**()

**Returns** total number of keys in the database.

**Return type** int

**update**(*__data=None*, *db=None*, *expire_time=None*, ***kwargs*)
See *TokyoTyrant.set_bulk()*.

**size**
Property which exposes the size of the database.

**error**
Return a 2-tuple of error code and message for the last error reported by the server (if set).

**optimize**(*options*)

**Parameters** **options** (*str*) – option format string to use when optimizing database.

**Returns** boolean indicating success

**synchronize**()

**Returns** boolean indicating success

Synchronize data to disk.

**copy**(*path*)

**Parameters** **path** (*str*) – destination for copy of database.

**Returns** boolean indicating success

Copy the database file to the given path.

**get_range**(*start*, *stop=None*, *max_keys=0*)

**Parameters**

- **start** (*str*) – start-key for range
- **stop** (*str*) – stop-key for range (optional)
- **max_keys** (*int*) – maximum keys to fetch

**Returns** a mapping of key-value pairs falling within the given range.

**Return type** dict

---

**Note:** Only works with tree databases.

---

**match_prefix**(*prefix*, *max_keys=1024*)

**Parameters**

- **prefix** (*str*) – key prefix to match
- **max_keys** (*int*) – maximum number of results to return

**Returns** list of keys that matched the given prefix.

**Return type** list

**match_regex**(*regex*, *max_keys=1024*)

**Parameters**

- **regex** (*str*) – regular-expression to match
- **max_keys** (*int*) – maximum number of results to return

**Returns** list of keys that matched the given regular expression.

**Return type** list

**iter_from**(*start_key*)

**Parameters** **start_key** – key to start iteration.

**Returns** list of key/value pairs obtained by iterating from start-key.

**Return type** dict

**keys**()

**Returns** list of all keys in database

**Return type** list

**items**()

**Returns** list of all key/value tuples in database

**Return type** list

**set_index**(*name*, *index_type*, *check_exists=False*)

**Parameters**

- **name** (*str*) – column name to index

- **index_type** (*int*) – see *Index types* for values
- **check_exists** (*bool*) – if true, an error will be raised if the index already exists.

**Returns** boolean indicating success

Create an index on the given column in a table database.

**optimize_index**(*name*)

**Parameters** **name** (*str*) – column name index to optimize

**Returns** boolean indicating success

Optimize the index on a given column.

**delete_index**(*name*)

**Parameters** **name** (*str*) – column name index to delete

**Returns** boolean indicating success

Delete the index on a given column.

**search**(*expressions*, *cmd=None*)

**Parameters**

- **expressions** (*list*) – zero or more search expressions
- **cmd** (*str*) – extra command to apply to search results

**Returns** varies depending on `cmd`.

Perform a search on a table database. Rather than call this method directly, it is recommended that you use the *QueryBuilder* to construct and execute table queries.

**genuid**()

**Returns** integer id

Generate a unique ID.

**class QueryBuilder**

Construct and execute table queries.

**filter**(*column*, *op*, *value*)

**Parameters**

- **column** (*str*) – column name to filter on
- **op** (*int*) – operation, see *Filter types* for available values
- **value** – value for filter expression

Add a filter expression to the query.

**order_by**(*column*, *ordering=None*)

**Parameters**

- **column** (*str*) – column name to order by
- **ordering** (*int*) – ordering method, defaults to lexical ordering. See *Ordering types* for available values.

Specify ordering of query results.

**limit**(*limit=None*)

**Parameters** **limit** (*int*) – maximum number of results

Limit the number of results returned by query.

**offset**(*offset=None*)

**Parameters** **offset** (*int*) – number of results to skip over.

Skip over results returned by query.

**execute**(*client*)

**Parameters** **client** (TokyoTyrant) – database client

**Returns** list of keys matching query criteria

**Return type** list

Execute the query and return a list of the keys of matching records.

**delete**(*client*)

**Parameters** **client** (TokyoTyrant) – database client

**Returns** boolean indicating success

Delete records that match the query criteria.

**get**(*client*)

**Parameters** **client** (TokyoTyrant) – database client

**Returns** list of 2-tuples consisting of `key, value`.

**Rtype list**

Execute query and return a list of keys and values for records matching the query criteria.

**count**(*client*)

**Parameters** **client** (TokyoTyrant) – database client

**Returns** number of query results

Return count of matching records.

### 3.3.1 Index types

**INDEX_STR**

**INDEX_NUM**

**INDEX_TOKEN**

**INDEX_QGRAM**

### 3.3.2 Filter types

**OP_STR_EQ**

**OP_STR_CONTAINS**

**OP_STR_STARTSWITH**

**OP_STR_ENDSWITH**

**OP_STR_ALL**

**OP_STR_ANY**

**OP_STR_ANYEXACT**

**OP_STR_REGEX**

**OP_NUM_EQ**

**OP_NUM_GT**

**OP_NUM_GE**

**OP_NUM_LT**

**OP_NUM_LE**

**OP_NUM_BETWEEN**

**OP_NUM_ANYEXACT**

**OP_FTS_PHRASE**

**OP_FTS_ALL**

**OP_FTS_ANY**

**OP_FTS_EXPRESSION**

**OP_NEGATE**
Combine with other operand using bitwise-or to negate the filter.

**OP_NOINDEX**
Combine with other operand using bitwise-or to prevent using an index.

### 3.3.3 Ordering types

**ORDER_STR_ASC**

**ORDER_STR_DESC**

**ORDER_NUM_ASC**

**ORDER_NUM_DESC**

## 3.4 Embedded Servers

**class EmbeddedServer**(*server='ktserver'*, *host='127.0.0.1'*, *port=None*, *database='*'*, *server_args=None*)

**Parameters**

- **server** (*str*) – path to ktserver executable
- **host** (*str*) – host to bind server on
- **port** (*int*) – port to use (optional)
- **database** (*str*) – database filename, default is in-memory hash table
- **server_args** (*list*) – additional command-line arguments for server

Create a manager for running an embedded (sub-process) Kyoto Tycoon server. If the port is not specified, a random high port will be used.

Example:

```
>>> from kt import EmbeddedServer
>>> server = EmbeddedServer()
>>> server.run()
True
>>> client = server.client
>>> client.set('k1', 'v1')
1
>>> client.get('k1')
'v1'
>>> server.stop()
True
```

**run**()

**Returns** boolean indicating if server successfully started

Run `ktserver` in a sub-process.

**stop**()

**Returns** boolean indicating if server was stopped

Stop the running embedded server.

**client**

*KyotoTycoon* client bound to the embedded server.

**class EmbeddedTokyoTyrantServer**(*server='ttserver'*, *host='127.0.0.1'*, *port=None*, *database='*'*, *server_args=None*)

**Parameters**

- **server** (*str*) – path to ttserver executable
- **host** (*str*) – host to bind server on
- **port** (*int*) – port to use (optional)
- **database** (*str*) – database filename, default is in-memory hash table
- **server_args** (*list*) – additional command-line arguments for server

Create a manager for running an embedded (sub-process) Tokyo Tyrant server. If the port is not specified, a random high port will be used.

Example:

```
>>> from kt import EmbeddedTokyoTyrantServer
>>> server = EmbeddedTokyoTyrantServer()
>>> server.run()
True
>>> client = server.client
>>> client.set('k1', 'v1')
True
>>> client.get('k1')
'v1'
>>> server.stop()
True
```

**run**()

**Returns** boolean indicating if server successfully started

Run `ttserver` in a sub-process.

**stop**()

**Returns** boolean indicating if server was stopped

Stop the running embedded server.

**client**

*TokyoTyrant* client bound to the embedded server.

CHAPTER 4

# Indices and tables

- genindex
- modindex
- search

# Index

## Symbols



## A



## C



## D



## E



## F



## G



## I



## K