# ukt Documentation

***Release 0.1.2***

**charles leifer**

**Sep 27, 2019**

# Contents:

# µkt

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

- Full-featured implementation of protocol.
- Performant serialization via C extension.
- Thread-safe and greenlet-safe.
- Socket pooling.
- Simple APIs.

CHAPTER 1

# Installation

*ukt* can be installed using `pip`:

```
$ pip install ukt
```

## 1.1 Dependencies

*ukt* has no dependencies. For development, you will need to install *cython* in order to build the serializer extension.

## 1.2 Installing with git

To install the latest version with git:

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

## 1.3 Installing Kyoto Tycoon

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

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

Alternatively you can use the following Docker images:

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

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

- Kyoto Tycoon documentation

CHAPTER 2

# Usage

This document describes how to use *ukt* with Kyoto Tycoon.

## 2.1 Features

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

```
>>> from ukt 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')
```

(continues on next page)

(continued from previous page)

```
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 library also supports 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. *ukt* 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 ukt import KT_PICKLE, KyotoTycoon
>>> client = KyotoTycoon(serializer=KT_PICKLE)
>>> client.set('k1', {'this': 'is', 'a': ['python object']})
1
```

(continues on next page)

(continued from previous page)

```
>>> 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
```

Expiration time can be specified in the following ways:

- Integers less than 6 months (in seconds) are treated as relative timestamps.

- Integers larger are treated as unix timestamps.
- `datetime.datetime` objects specify the expire time.
- `datetime.timedelta` objects specify a the time relative to now.

### 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.

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.

## 3.2 Kyoto Tycoon client

**class KyotoTycoon**(*host='127.0.0.1'*, *port=1978*, *timeout=None*, *default_db=0*, *decode_keys=True*, *serializer=None*, *encode_value=None*, *decode_value=None*, *max_age=3600*)

**Parameters**

- **host** (*str*) – server host.
- **port** (*int*) – server port.
- **timeout** (*int*) – socket timeout for database connection.
- **default_db** (*int*) – default database index.
- **decode_keys** (*bool*) – decode keys as utf8-encoded unicode.

- **serializer** – serialization method to use for storing/retrieving values. Default is KT_BINARY, which treats values as utf8-encoded unicode. KT_NONE disables all serialization, or use one of KT_JSON, KT_MSGPACK or KT_PICKLE.
- **encode_value** – custom serializer for encoding values as bytestrings.
- **decode_value** – custom deserializer for decoding bytestrings.
- **max_age** (*int*) – max idle time for socket in connection pool.

Client for interacting with Kyoto Tycoon database.

**set_database**(*db*)

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

Set the default database for the client. The Kyoto Tycoon server can be run by specifying multiple database paths. This method allows you to specify which database the client communicates with by default, though most methods accept a db parameter which can override the default for a given call.

**close_all**()

**Returns** number of connections that were closed.

Close all connections in the connection pool. The pool maintains two sets of connections:

- Binary protocol connections.
- HTTP client connections for the HTTP API.

Since the binary protocol only implements a subset of the total commands, *ukt* will transparently use the appropriate connection type for a given method.

**serialize_dict**(*d*, *encode_values=True*)

**Parameters**

- **d** (*dict*) – arbitrary data.
- **encode_values** (*bool*) – serialize the values using the configured serialization scheme.

**Returns** serialized data.

Serialize a dict as a sequence of bytes compatible with KT's built-in lua mapdump function and the *Hash* container type.

**deserialize_dict**(*data*, *decode_values=True*)

**Parameters**

- **data** (*bytes*) – serialized data.
- **decode_values** (*bool*) – decode values using the configured serialization scheme.

**Returns** data dict.

Deserialize a sequence of bytes into a dictionary, optionally decoding the values as unicode strings. Compatible with KT's built-in lua mapload function and the *Hash* container type.

**serialize_list**(*l*, *encode_values=True*)

**Parameters**

- **l** (*list*) – arbitrary data.
- **encode_values** (*bool*) – serialize the values using the configured serialization scheme.

**Returns** serialized data.

Serialize a `list` as a sequence of bytes compatible with KT's built-in lua `arraydump` function and the *List* container type.

**deserialize_list**(*data*, *decode_values=True*)

**Parameters**

- **data** (*bytes*) – serialized data.
- **decode_values** (*bool*) – decode values using the configured serialization scheme.

**Returns** data `list`.

Deserialize a a sequence of bytes into a list, optionally decoding the values as unicode strings. Compatible with KT's built-in lua `arrayload` function and the *List* container type.

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

**Parameters**

- **keys** (*list*) – keys to retrieve.
- **db** (*int*) – database index.
- **decode_values** (*bool*) – decode values using the configured serialization scheme.

**Returns** result dictionary

Efficiently retrieve multiple key/value pairs from a database. If a key does not exist, it will not be present in the result dictionary.

**get_bulk_details**(*db_key_list*, *decode_values=True*)

**Parameters**

- **db_key_list** (*list*) – a list of `(db, key)` tuples to fetch.
- **decode_values** (*bool*) – decode values using the configured serialization scheme.

**Returns** list of tuples: `(db index, key, value, expire time)`

Like *get_bulk()*, but the return value is a list of tuples with additional information for each key. Since each key is of the form `(db, key)`, this method can be used to efficiently fetch records from multiple databases.

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

**Parameters**

- **key** (*str*) – key to look-up
- **db** (*int*) – database index
- **decode_value** (*bool*) – decode value using serializer.

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

Fetch and (optionally) deserialize the value for the given key.

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

**Parameters**

- **key** (*str*) – key to look-up
- **db** (*int*) – database index

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

Fetch the value for the given key. The resulting value will **not** be deserialized.

**set_bulk**(*data*, *db=None*, *expire_time=None*, *no_reply=False*, *encode_values=True*)

**Parameters**

- **data** (*dict*) – mapping of key/value pairs to set.
- **db** (*int*) – database index
- **expire_time** (*int*) – expiration time in seconds
- **no_reply** (*bool*) – execute the operation without a server acknowledgment.
- **encode_values** (*bool*) – serialize the values using the configured serialization scheme (e.g., `KT_MSGPACK`).

**Returns** number of keys that were set, or `None` if `no_reply`.

Efficiently set multiple key/value pairs. If given, the provided `db` and `expire_time` values will be used for all key/value pairs being set.

**set_bulk_details**(*data*, *no_reply=False*, *encode_values=True*)

**Parameters**

- **data** (*list*) – a list of 4-tuples: `(db, key, value, expire-time)`
- **no_reply** (*bool*) – execute the operation without a server acknowledgment.
- **encode_values** (*bool*) – serialize the values using the configured serialization scheme (e.g., `KT_MSGPACK`).

**Returns** number of keys that were set, or `None` if `no_reply`.

Efficiently set multiple key/value pairs. Unlike *set_bulk()*, this method can be used to set key/value pairs in multiple databases in a single call, and each key can specify its own expire time.

**set**(*key*, *value*, *db=None*, *expire_time=None*, *no_reply=False*, *encode_value=True*)

**Parameters**

- **key** (*str*) – key to set.
- **value** – value to store.
- **db** (*int*) – database index.
- **expire_time** (*int*) – expiration time in seconds.
- **no_reply** (*bool*) – execute the operation without a server acknowledgment.
- **encode_value** (*bool*) – encode value using serializer.

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

Set a single key/value pair.

**set_bytes**(*key*, *value*, *db=None*, *expire_time=None*, *no_reply=False*)

**Parameters**

- **key** (*str*) – key to set.
- **value** (*bytes*) – raw bytes to store.
- **db** (*int*) – database index.
- **expire_time** (*int*) – expiration time in seconds.

- **no_reply** (*bool*) – execute the operation without a server acknowledgment.

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

Set a single key/value pair, without serializing the value.

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

**Parameters**

- **keys** (*list*) – list of keys to remove
- **db** (*int*) – database index
- **no_reply** (*bool*) – execute the operation without a server acknowledgment.

**Returns** number of keys that were removed

Remove multiple keys from a database in a single operation.

**remove_bulk_details**(*db_key_list*, *no_reply=False*)

**Parameters**

- **db_key_list** – a list of 2-tuples to retrieve: `(db index, key)`
- **no_reply** (*bool*) – execute the operation without a server acknowledgment.

**Returns** number of keys that were removed

Like *remove_bulk()*, but allows keys to be removed from multiple databases in a single call. The input is a list of `(db, key)` tuples.

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

**Parameters**

- **key** (*str*) – key to remove
- **db** (*int*) – database index
- **no_reply** (*bool*) – execute the operation without a server acknowledgment.

**Returns** number of rows removed

Remove a single key from the database.

**script**(*name*, *data=None*, *no_reply=False*, *encode_values=True*, *decode_values=True*)

**Parameters**

- **name** (*str*) – name of lua function to call.
- **data** (*dict*) – mapping of key/value pairs to pass to lua function.
- **no_reply** (*bool*) – execute the operation without a server acknowledgment.
- **encode_values** (*bool*) – serialize values passed to lua function.
- **decode_values** (*bool*) – deserialize values returned by lua function.

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

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. Likewise, if `decode_values` is `True` the values returned by the Lua function will be deserialized using the configured serializer.

**raw_script**(*name*, *data=None*, *no_reply=False*)

**Parameters**

- **name** (*str*) – name of lua function to call.
- **data** (*dict*) – mapping of key/value pairs to pass to lua function.
- **no_reply** (*bool*) – execute the operation without a server acknowledgment.

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

Execute a lua function and return the result with no post-processing or serialization.

**report**()

**Returns** status fields and values

**Return type** dict

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

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

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

**Returns** status fields and values

**Return type** dict

Obtain status information from the server about the selected database.

**list_databases**()

**Returns** a list of `(database path, status dict)` for each configured database.

Return the list of databases and their status information.

**databases**

Returns the list of paths for the configured databases.

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

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

**Returns** boolean indicating success

Remove all keys from the database.

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

**Parameters**

- **hard** (*bool*) – perform a "hard" synchronization.
- **command** (*str*) – command to execute after synchronization.
- **db** (*int*) – database index.

**Returns** boolean indicating success.

Synchronize the database, optionally executing the given command upon success. This can be used to create hot backups, for example.

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

**Parameters**

- **key** (*str*) – key to add.
- **value** – value to store.
- **db** (*int*) – database index.

- **expire_time** (*int*) – expiration time in seconds.
- **encode_value** (*bool*) – serialize the value using the configured serialization method.

**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*, *encode_value=True*)

**Parameters**

- **key** (*str*) – key to replace.
- **value** – value to store.
- **db** (*int*) – database index.
- **expire_time** (*int*) – expiration time in seconds.
- **encode_value** (*bool*) – serialize the value using the configured serialization method.

**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*, *encode_value=True*)

**Parameters**

- **key** (*str*) – key to append value to.
- **value** – data to append.
- **db** (*int*) – database index.
- **expire_time** (*int*) – expiration time in seconds.
- **encode_value** (*bool*) – serialize the value using the configured serialization method.

**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()*.

**increment**(*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*) – database index.
- **expire_time** (*int*) – expiration time in seconds.

**Returns** new value at key.

**Return type** int

Increment the value stored in the given key.

**increment_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*) – database index.
- **expire_time** (*int*) – expiration time in seconds.

**Returns** new value at key.

**Return type** float

Increment the floating-point value stored in the given key.

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

**Parameters**

- **key** (*str*) – key to append value to.
- **old_val** – original value to test.
- **new_val** – new value to store.
- **db** (*int*) – database index.
- **expire_time** (*int*) – expiration time in seconds.
- **encode_value** (*bool*) – serialize the old and new values using the configured serialization method.

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

**Return type** bool

Perform an atomic compare-and-set the value stored at a given key.

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

**Parameters**

- **key** (*str*) – key to test.
- **db** (*int*) – database index.

**Returns** boolean indicating if key exists.

Return whether or not the given key exists in the database.

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

**Parameters**

- **key** (*str*) – key.
- **db** (*int*) – database index.

**Returns** length of the value in bytes, or `None` if not found.

Return the length of the raw value stored at the given key. If the key does not exist, returns `None`.

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

**Parameters**

- **key** (*str*) – key to remove.
- **db** (*int*) – database index.
- **decode_value** (*bool*) – deserialize the value using the configured serialization method.

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

Perform atomic get-and-remove the value stored in a given key. This method is also available as *KyotoTycoon.pop()* if that's easier to remember.

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

**Parameters**

- **step** (*int*) – number of steps, default is 0
- **db** (*int*) – database index

**Returns** boolean indicating success

Vacuum the database.

**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*) – database index.

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

**Return type** list

Return sorted list of keys that match the given prefix.

**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*) – database index

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

**Return type** list

Return sorted list of keys that match the given regular expression.

**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*) – database index

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

**Return type** list

Return sorted list of keys that are within a given edit distance from a string.

**ulog_list**()

**Returns** a list of 3-tuples describing the files in the update log.

Returns a list of metadata about the state of the update log. For each file in the update log, a 3-tuple is returned. For example:

```
>>> kt.ulog_list()
[('/var/lib/database/ulog/kt/0000000037.ulog',
  '67150706',
  datetime.datetime(2019, 1, 4, 1, 28, 42, 43000)),
 ('/var/lib/database/ulog/kt/0000000038.ulog',
  '14577366',
  datetime.datetime(2019, 1, 4, 1, 41, 7, 245000))]
```

**ulog_remove**(*max_dt*)

**Parameters** **max_dt** (*datetime*) – maximum datetime to preserve

**Returns** boolean indicating success

Removes all update-log files older than the given datetime.

**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.

**size**(*db=None*)

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

**Returns** size of database in bytes.

Property which exposes the size information returned by the *status()* API.

**__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

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

**Parameters**

- **__data** (*dict*) – optionally provide data as a dictionary.
- **kwargs** – provide data as keyword arguments.

**Returns** number of keys that were set.

Efficiently set or update multiple key/value pairs. Provided for compatibility with `dict` interface. For more control use the *set_bulk()*.

**pop**(*key*, *db=None*, *decode_value=True*)
Get and remove the data stored in a given key in a single operation.

See *KyotoTycoon.seize()*.

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

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

**Returns** all keys in database

**Return type** generator

**Warning:** The `keys()` method uses a cursor and can be very slow.

**keys_nonlazy**(*db=None*)

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

**Returns** all keys in database

**Return type** list

Non-lazy implementation of *keys()*. Behind-the-scenes, calls *match_prefix()* with an empty string as the prefix.

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

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

**Returns** all values in database

**Return type** generator

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

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

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

**Return type** generator

**__iter__**()
Iterating over the database yields an iterator over the keys of the database. Equivalent to *keys()*.

**touch**(*key*, *xt=None*, *db=None*)

**Parameters**

- **key** (*str*) – key to update.
- **xt** (*int*) – new expire time (or None).
- **db** (*int*) – database index.

**Returns** old expire time or None if key not found.

Run a lua function (touch) defined in scripts/kt.lua that allows one to update the TTL / expire time of a key.

The old expire time is returned. If the key does not exist, then None is returned.

**touch_bulk**(*keys*, *xt=None*, *db=None*)

**Parameters**

- **keys** (*list*) – keys to update.
- **xt** (*int*) – new expire time (or None).
- **db** (*int*) – database index.

**Returns** a dict of key -> old expire time.

Run a lua function (touch_bulk) defined in scripts/kt.lua that allows one to update the TTL / expire time of multiple keys.

The return value is a dictionary of key -> old expire time. If the key does not exist, then the key is omitted from the return value.

**touch_relative**(*key*, *n*, *db=None*)

**Parameters**

- **key** (*str*) – key to update.
- **n** (*int*) – seconds to increase expire-time.
- **db** (*int*) – database index.

**Returns** new expire time or None if key not found.

Run a lua function (touch_bulk_relative) defined in scripts/kt.lua that allows one to increment the TTL / expire time of a key.

The new expire time is returned. If the key does not exist, then None is returned.

**touch_bulk_relative**(*keys*, *n*, *db=None*)

**Parameters**

- **keys** (*list*) – keys to update.
- **n** (*int*) – seconds to increase expire-time.
- **db** (*int*) – database index.

**Returns** a dict of key -> new expire time.

Run a lua function (touch_bulk_relative) defined in scripts/kt.lua that allows one to update the TTL / expire time of multiple keys.

The return value is a dictionary of key -> new expire time. If the key does not exist, then the key is omitted from the return value.

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

**Parameters**

- **key** (*str*) – key to check.
- **db** (*int*) – database index

**Returns** expire timestamp or None if key not found.

Get the expire time by running a lua function (expire_time) defined in scripts/kt.lua.

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

**Parameters**

- **key** (*str*) – key to check.
- **db** (*int*) – database index

**Returns** expire `datetime` or None if key not found.

Get the expire time as a `datetime`.

**error**(*db=None*)

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

**Returns** a 2-tuple of (code, message)

Get the last error code and message.

If the last command was successful, then (0, 'success') is returned.

**Hash**(*key*, *encode_values=True*, *decode_values=True*, *db=None*)

**Parameters**

- **key** (*str*) – key to store the hash table.
- **encode_values** (*bool*) – serialize the hash values using the configured serializer.
- **decode_values** (*bool*) – de-serialize the hash values using the configured serializer.
- **db** (*int*) – database index.

Create a *Hash* container instance.

**List**(*key*, *encode_values=True*, *decode_values=True*, *db=None*)

**Parameters**

- **key** (*str*) – key to store the list.
- **encode_values** (*bool*) – serialize the list items using the configured serializer.
- **decode_values** (*bool*) – de-serialize the list items using the configured serializer.
- **db** (*int*) – database index.

Create a *List* container instance.

**Set**(*key*, *encode_values=True*, *decode_values=True*, *db=None*)

**Parameters**

- **key** (*str*) – key to store the set.
- **encode_values** (*bool*) – serialize the set keys using the configured serializer.
- **decode_values** (*bool*) – de-serialize the set keys using the configured serializer.
- **db** (*int*) – database index.

Create a *Set* container instance.

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

**Parameters**

- **key** (*str*) – key to use for the queue metadata.
- **db** (*int*) – database index.

Create a *Queue*, which provides efficient operations for implementing a priority queue.

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

**Parameters**

- **key** (*str*) – key to use for the schedule metadata.
- **db** (*int*) – database index.

Create a *Schedule*, which provides efficient operations for implementing a sorted schedule.

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

**Parameters**

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

**Returns** *Cursor* object

**class Cursor**(*protocol*, *cursor_id*, *db=None*, *decode_values=True*, *encode_values=True*)

**Parameters**

- **protocol** (KyotoTycoon) – client instance.
- **cursor_id** (*int*) – cursor unique identifier.
- **db** (*int*) – database index.
- **decode_values** (*bool*) – decode values using client serializer when reading from the cursor.
- **encode_values** (*bool*) – encode values using client serializer when writing to the cursor.

Create a helper for working with the database using the cursor interface.

**jump**(*key=None*)

**Parameters** **key** (*str*) – key to jump to or None.

**Returns** boolean indicating success.

Jump to the given key. If not provided, will jump to the first key in the database.

**jump_back**(*key=None*)

**Parameters** **key** (*str*) – key to jump backwards to or None.

**Returns** boolean indicating success.

Jump backwards to the given key. If not provided, will jump to the last key in the database.

**step**()

**Returns** boolean indicating success.

Step to the next key. Returns False when past the last key of the database.

**step_back**()

**Returns** boolean indicating success.

Step to the previous key. Returns False when past the first key of the database.

**key**(*step=False*)

**Parameters** **step** (*bool*) – step to next record after reading.

**Returns** key of the currently-selected record.

**value**(*step=False*)

**Parameters** **step** (*bool*) – step to next record after reading.

**Returns** value of the currently-selected record.

**get**(*step=False*)

**Parameters** **step** (*bool*) – step to next record after reading.

**Returns** `(key, value)` of the currently-selected record.

**set_value**(*value*, *step=False*, *expire_time=None*)

**Parameters**

- **value** – value to set
- **step** (*bool*) – step to next record after writing.
- **expire_time** (*int*) – optional expire time for record.

**Returns** boolean indicating success.

Set the value at the currently-selected record.

**remove**()

**Returns** boolean indicating success.

Remove the currently-selected record.

**seize**()

**Returns** `(key, value)` of the currently-selected record.

Get and remove the currently-selected record.

**close**()

**Returns** boolean indicating success.

Close the cursor.

**class Queue**(*client*, *key*, *db=None*)

**Parameters**

- **client** (KyotoTycoon) – client instance.
- **key** (*str*) – key to store queue data.
- **db** (*int*) – database index.

Priority queue implementation using lua functions (provided in the `scripts/kt.lua` module).

**add**(*item*, *score=None*)

**Parameters**

- **item** – item to add to queue.
- **score** (*int*) – score (for priority support), higher values will be dequeued first. If not provided, defaults to `0`.

**Returns** id of newly-added item.

**extend**(*items*, *score=None*)

**Parameters**

- **items** (*list*) – list of items to add to queue.
- **score** (*int*) – score (for priority support), higher values will be dequeued first. If not provided, defaults to `0`.

**Returns** number of items added to queue.

**pop**(*n=1*, *min_score=None*)

**Parameters**

- **n** (*int*) – number of items to remove from queue.
- **min_score** (*int*) – minimum priority score. If not provided, all items will be considered regardless of score.

**Returns** either a single item or a list of items (depending on `n`).

Pop one or more items from the head of the queue.

**rpop**(*n=1*, *min_score=None*)

**Parameters**

- **n** (*int*) – number of items to remove from end of queue.
- **min_score** (*int*) – minimum priority score. If not provided, all items will be considered regardless of score.

**Returns** either a single item or a list of items (depending on `n`).

Pop one or more items from the end of the queue.

**bpop**(*timeout=None*, *min_score=None*)

**Parameters**

- **timeout** (*int*) – seconds to block before giving up.
- **min_score** (*int*) – minimum priority score. If not provided, all items will be considered regardless of score.

**Returns** item from the head of the queue, or if no items are added before the timeout, `None` is returned.

Pop an item from the queue, blocking if no items are available.

**peek**(*n=1*, *min_score=None*)

**Parameters**

- **n** (*int*) – number of items to read from queue.
- **min_score** (*int*) – minimum priority score. If not provided, all items will be considered regardless of score.

**Returns** either a single item or a list of items (depending on `n`).

Read (without removing) one or more items from the head of the queue.

**rpeek**(*n=1*, *min_score=None*)

**Parameters**

- **n** (*int*) – number of items to read from end of queue.

- **min_score** (*int*) – minimum priority score. If not provided, all items will be considered regardless of score.

**Returns** either a single item or a list of items (depending on `n`).

Read (without removing) one or more items from the end of the queue.

**count**()

**Returns** number of items in the queue.

**remove**(*data*, *n=None*, *min_score=None*)

**Parameters**

- **data** – value to remove from queue.
- **n** (*int*) – max occurrences to remove.
- **min_score** (*int*) – minimum priority score. If not provided, all items will be considered regardless of score.

**Returns** number of items removed.

Remove one or more items by value, starting from the head of the queue.

**rremove**(*data*, *n=None*, *min_score=None*)

**Parameters**

- **data** – value to remove from end of queue.
- **n** (*int*) – max occurrences to remove.
- **min_score** (*int*) – minimum priority score. If not provided, all items will be considered regardless of score.

**Returns** number of items removed.

Remove one or more items by value, starting from the end of the queue.

**transfer**(*dest*, *n=1*)

**Parameters**

- **dest** – destination queue key or *Queue* instance.
- **n** (*int*) – number of items to transfer.

**Returns** either the item that was transferred or the list of items that was transferred, depending on `n`.

Transfer items from the head of the queue to the tail of the destination queue. Priority scores are preserved. If the source queue is empty, then either `None` or an empty list will be returned (depending on whether `n=1`).

**set_priority**(*data*, *score*, *n=None*)

**Parameters**

- **data** – value to remove from end of queue.
- **score** (*int*) – new score for the item.
- **n** (*int*) – max occurrences to update.

Update the priority of one or more items in the queue, by value.

**clear**()

**Returns** number of items in queue when cleared.

Remove all items from queue.

**class Schedule**(*client*, *key*, *db=None*)

**Parameters**

- **client** (KyotoTycoon) – client instance.
- **key** (*str*) – key to store schedule data.
- **db** (*int*) – database index.

Prioritized schedule implementation using lua functions (provided in the `scripts/kt.lua` module).

**add**(*item*, *score=0*)

**Parameters**

- **item** – add an item to the schedule.
- **score** (*int*) – score (arrival time) of item.

Add an item to the schedule, with a given score / arrival time.

**read**(*score=None*, *n=None*)

**Parameters**

- **score** (*int*) – score threshold or arrival time
- **n** (*int*) – maximum number of items to read.

**Returns** a list of items

Destructively read up-to `n` items from the schedule, whose item score is below the given `score`.

**clear**()

Clear the schedule, removing all items.

**count**()

**Returns** number of items in the schedule.

Return the number of items in the schedule.

**items**(*n=None*)

**Parameters** **n** (*int*) – limit the number of items to read.

**Returns** a list of up-to `n` items from the schedule.

Non-destructively read up-to `n` items from the schedule, in order of score.

## 3.3 Container types

Simple container types that emulate Python or Redis types, and rely on Kyoto Tycoon's lua serialization helpers. Behind-the scenes, these types are using lua functions to read the entire value into a Lua table and write it back. Because the full data must be deserialized for reading, and re-serialized for writing, all operations are O(n).

These container types support transparent serialization using the configured serializer (`KT_PICKLE`, `KT_MSGPACK`, etc).

**class Hash**(*kt*, *key*, *encode_values=True*, *decode_values=True*, *db=None*)

**Parameters**

- **kt** (KyotoTycoon) – client
- **key** (*str*) – key to store hash data
- **encode_values** (*bool*) – values should be serialized using the configured serializer (e.g., KT_PICKLE, KT_MSGPACK, etc).
- **decode_values** (*bool*) – values should be deserialized using the configured serializer.
- **db** (*int*) – database index to store hash. If not specified, will use the default db configured for the kt client.

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

**Parameters**

- **__data** (*dict*) – provide data as a dictionary.
- **kwargs** – or provide data keyword arguments.

**Returns** number of keys that were set.

Update the data stored in the hash.

**get_bulk**(*keys*)

**Parameters** **keys** – an iterable of keys to fetch.

**Returns** a dictionary of key/value pairs. If a requested key is not found, it is not included in the returned data.

**remove_bulk**(*keys*)

**Parameters** **keys** – an iterable of keys to remove.

**Returns** number of key/value pairs that were removed.

**get_all**()

**Returns** dictionary of all data stored in the hash

A more efficient implementation utilizes the Python implementation of the lua serializers. Use *Hash.get_raw()*.

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

**Parameters**

- **key** (*str*) – key to store
- **value** – data

Set a single key/value pair in the hash. Returns number of records written (1).

**setnx**(*key*, *value*)

**Parameters**

- **key** (*str*) – key to store
- **value** – data

**Returns** 1 on success, 0 if key already exists.

Set a single key/value pair in the hash only if the key does not already exist.

**get**(*key*)

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

**Returns** value, if key exists, or `None`.

**remove**(*key*)

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

**Returns** number of keys removed, 1 on success, 0 if key not found.

**length**()

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

**contains**(*key*)

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

**Returns** boolean indicating whether the given key exists.

**unpack**(*prefix=None*)

**Parameters** **prefix** (*str*) – prefix for unpacked-keys

**Returns** number of keys that were written

Unpack the key/value pairs in the hash into top-level key/value pairs in the database, optionally prefixing the unpacked keys with the given prefix.

**pack**(*start=None*, *stop=None*, *count=None*)

**Parameters**

- **start** (*str*) – start key, or will be first key in the database
- **stop** (*str*) – stop key, or will be last key in the database
- **count** (*int*) – limit number of keys to pack

**Returns** number of keys that were packed

Pack a range of key/value pairs in the database into a hash.

**pack_keys**(*key*)

**Parameters** **key** (*str*) – destination key for *List* of keys.

**Returns** number of keys that were written to the list

Pack the keys of the hash into a *List* at the given key.

**pack_values**(*key*)

**Parameters** **key** (*str*) – destination key for *List* of values.

**Returns** number of values that were written to the list

Pack the values of the hash into a *List* at the given key.

**__len__**()
See *length()*.

**__contains__**()
See *contains()*.

**__getitem__**()
See *get()*.

**__setitem__**()
See *set()*.

**__detitem__**()
See *remove()*.

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

**get_raw**()

**Returns** dictionary of all data stored in hash, or `None` if empty.

Utilize a more-efficient implementation for fetching all data stored in the hash. Rather than going through Lua, we read the raw value of the serialized hash, then deserialize it using an equivalent format to KT's internal `mapload` format.

**set_raw**(*d*)

**Parameters** **d** (*dict*) – dictionary of all data to store in hash.

Utilize a more-efficient implementation for setting the data stored in the hash. Rather than going through Lua, we write the raw value of the serialized hash, using an equivalent format to KT's internal `mapdump` format.

**class List**(*kt*, *key*, *encode_values=True*, *decode_values=True*, *db=None*)

**Parameters**

- **kt** (*KyotoTycoon*) – client
- **key** (*str*) – key to store list data
- **encode_values** (*bool*) – values should be serialized using the configured serializer (e.g., KT_PICKLE, KT_MSGPACK, etc).
- **decode_values** (*bool*) – values should be deserialized using the configured serializer.
- **db** (*int*) – database index to store list. If not specified, will use the default db configured for the kt client.

**appendleft**(*value*)

**Parameters** **value** – value to append to left-side (head) of list.

**Returns** length of list after operation.

**appendright**(*value*)

**Parameters** **value** – value to append to right-side (tail) of list.

**Returns** length of list after operation.

**append**(*value*)
Alias for *appendright()*.

**extend**(*values*)

**Parameters** **values** – an iterable of values to add to the tail of the list.

**Returns** length of list after operation.

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

**Parameters**

- **start** (*int*) – start index (0 for first element)
- **stop** (*int*) – stop index. Supports negative values.

**Returns** a list of items corresponding to the given range.

Slicing operation equivalent to Python's list slice behavior. If the start or stop indices are out-of-bounds, the return value will be an empty list.

**index**(*index*)

**Parameters** **index** (*int*) – item index to fetch. Supports negative values.

**Returns** the value at the given index

Indexing operation equivalent to Python's list item lookup. If the index is out-of-bounds, an `IndexError` will be raised.

**insert**(*index*, *value*)

**Parameters**

- **index** (*int*) – index at which new value should be inserted. Supports negative values.
- **value** – value to insert

**Returns** length of list after operation

Insert an item into the list at the given index. If the index is out-of-bounds, an `IndexError` will be raised.

**remove**(*index*)

**Parameters** **index** (*int*) – item index to remove. Supports negative values.

**Returns** the value at the given index

Remove and return an item from the list by index. If the index is out-of-bounds, an `IndexError` will be raised.

**remove_range**(*start=None*, *stop=None*)

**Parameters**

- **start** (*int*) – start index to remove. Supports negative values.
- **stop** (*int*) – stop index of range to remove. Supports negative values.

**Returns** length of list after operation

Remove a range of values by index.

**popleft**()

**Returns** item at head of list or `None` if list is empty.

**popright**()

**Returns** item at tail of list or `None` if list is empty.

**pop**(*index=None*)

**Parameters** **index** (*int*) – index to pop (optional), or `None` to remove the item at the tail of the list.

**Returns** item removed or `None` if list is empty or the index is out-of-bounds.

**lpoprpush**(*dest=None*)

**Parameters** **dest** – destination key (or *List* object). If unspecified, the destination will be the current list and the operation is equivalent to a rotation.

**Returns** item that was moved, if source is not empty. If source list is empty, an `IndexError` is raised.

Pop the item at the head of the current list and push it to the tail of the dest list.

**rpoplpush**(*dest=None*)

**Parameters** **dest** – destination key (or *List* object). If unspecified, the destination will be the current list and the operation is equivalent to a rotation.

**Returns** item that was moved, if source is not empty. If source list is empty, an `IndexError` is raised.

Pop the item at the tail of the current list and push it to the head of the dest list.

**length**()

**Returns** length of the list.

**set**(*index*, *value*)

**Parameters**

- **index** (*int*) – index to set. Supports negative values.
- **value** – value to set at given index

Set the value at the given index. If the index is out-of-bounds, an `IndexError` will be raised.

**find**(*value*)

**Parameters** **value** – value to search for

**Returns** index of first occurrance of value starting from head of list.

**rfind**(*value*)

**Parameters** **value** – value to search for

**Returns** index of first occurrance of value starting from tail of list.

**unpack**(*start=None*, *stop=None*, *prefix=None*, *fmt=None*)

**Parameters**

- **start** (*int*) – start index of range to unpack
- **stop** (*int*) – stop index of range to unpack
- **prefix** (*str*) – prefix for output values
- **fmt** (*str*) – lua format-string for index, e.g. *'%08d'*.

Unpack the items in the list into top-level keys in the database. The key will begin with the provided prefix, and optionally accepts a format-string for formatting the index.

**pack**(*start=None*, *stop=None*, *count=None*)

**Parameters**

- **start** (*str*) – start key, or will be first key in the database
- **stop** (*str*) – stop key, or will be last key in the database
- **count** (*int*) – limit number of keys to pack

**Returns** number of keys that were packed

Pack the values for a range of keys in the database into a list.

**__len__**()

See *length()*.

**__contains__**()
See *find()*.

**__getitem__**()
Supports item indexes or slices. See *index()* and *get_range()*.

**__setitem__**()
See *set()*.

**__detitem__**()
See *remove()*.

**get_raw**()

**Returns** list of all data stored in list, or `None` if empty.

Utilize a more-efficient implementation for fetching all data stored in the list. Rather than going through Lua, we read the raw value of the serialized list, then deserialize it using an equivalent format to KT's internal `arrayload` format.

**set_raw**(*l*)

**Parameters** **l** (*list*) – list of all data to store in list.

Utilize a more-efficient implementation for setting the data stored in the list. Rather than going through Lua, we write the raw value of the serialized list, using an equivalent format to KT's internal `arraydump` format.

## 3.4 Embedded Servers

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

**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.
- **serializer** – serializer to use, e.g. `KT_BINARY` or `KT_MSGPACK`.
- **server_args** (*list*) – additional command-line arguments for server
- **quiet** (*bool*) – minimal logging and output.

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')
```

(continues on next page)

(continued from previous page)

```
'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.

CHAPTER 4

# Tuning Kyoto Tycoon

These are notes based on the documentation and distilled into something hopefully a bit easier to follow.

Supported by all databases:

- `log` - path to logfile
- `logkinds` - debug, info, warn or error
- `logpx` - prefix for each log message

Other options:

- `bnum` - number of buckets in the hash table
- `capcnt` - set capacity by record number
- `capsiz` - set capacity by memory usage
- `msiz` - size of internal memory-mapped region, typically should be set to a significant percentage of total memory (e.g., if you have 16g set to 12g).
- `psiz` - page size, defaults to 8192
- `pccap` - page-cache capacity
- `opts` - s=small, l=linear, c=compress - l=linked-list for hash collisions.
- `zcomp` - compression algorithm: zlib, def (deflate), gz, lzo, lzma or arc
- `zkey` - cipher key for compression (?)
- `rcomp` - comparator function: lex, dec (decimal), lexdesc, decdesc
- `apow` - record alignment, as power of 2, e.g. apow=3 == 8 bytes.
- `fpow` - maximum elements in the free-block pool
- `dfunit` - unit step for auto-defragmentation, default=0 (disabled).

## 4.1 Databases and available parameters

- Stash (`:`): bnum
- Cache hash (`*`): opts, bnum, zcomp, capcnt, capsiz, zkey
- Cache tree (`%`): opts, bnum, zcomp, zkey, psiz, rcomp, pccap
- File hash (`.kch`): opts, bnum, apow, fpow, msiz, dfunit, zcomp, zkey
- File tree (`.kct`): opts, bnum, apow, fpow, msiz, dfunit, zcomp, zkey, psiz
- Dir hash (`.kcd`): opts, zcomp, zkey
- Dir tree (`.kcf`): opts, zcomp, zkey, psiz, rcomp, pccap
- Plain-text (`.kcx`): n/a
- Prototype hash (−): n/a, not recommended (std::unorderedmap)
- Prototype tree (+): n/a, not recommended (std::map)

Information available here: https://fallabs.com/kyotocabinet/spex.html#tips

When choosing a database:

- Do you need persistence? If not, use one of the in-memory databases.
- Are the order of keys important? If so, use one of the tree databases.
- Do you want LRU eviction with an upper-bound of memory usage? Use cache hash.
- Are your values very large? Consider using the filesystem or the directory hash/tree database.

In-memory databases:

- time efficiency: Cache hash > Stash > Proto hash > Proto tree > Cache tree
- space efficiency: Cache tree > Stash > Cache hash > Proto hash > Proto tree

Persistent databases:

- time efficiency: Hash > Tree > Dir hash > Dir tree
- space efficiency: Tree > Hash > Dir tree > Dir hash

### 4.1.1 Stash database

Stash database is stored in memory and is a little bit more efficient than the cache hash (`*`), however it handles eviction slightly differently. The cache hash retains metadata so that it is able to do LRU eviction, whereas the stash database evicts random records.

- `bnum` - default is ~1M, should be 80%-400% of total records

### 4.1.2 CacheHashDB

Stored in-memory and supports LRU eviction, uses a doubly-linked hash map.

- `bnum`: default ~1M. Should be 50% - 400% of total records. Collision chaining is binary search
- `opts`: useful to reduce memory at expense of time effciency. Use compression if the key and value of each record is greater-than 1KB
- `capcnt` and/or `capsiz`: keep memory usage constant by expiring old records.

- supports compression, which is recommended when values are larger than 1KB.

### 4.1.3 CacheTreeDB

Inherits all tuning options from the CacheHashDB, since each node of the btree is serialized as a page-buffer and treated as a record in the cache hash db.

- `psiz`: default is 8192
- `pccap`: page-cache capacity, default is 64MB
- `rcomp`: comparator, default is lexical ordering

### 4.1.4 HashDB

On-disk hash table.

- `bnum`: default ~1M. Suggested ratio is twice the total number of records, but can be anything from 100% - 400%.
- `msiz`: Size of internal memory-mapped region. Default is 64MB. It is very advisable to set this to a value larger than the expected size of the database, e.g. 12G if you have 16G of memory available.
- `dfunit`: Unit step number of auto-defragmentation. Auto-defrag is disabled by default.
- `apow`: Power of the alignment of record size. Default=3, so the address of each record is aligned to a multiple of 8 (`2^3`) bytes.
- `fpow`: Power of the capacity of the free block pool. Default=10, rarely needs to be modified.

apow, fpow, opts and bnum *must* be specified before a DB is opened and cannot be changed after the fact.

### 4.1.5 TreeDB

Inherits tuning parameters from the HashDB, as the B-Tree is implemented on top of the file hash database. Supports the following additional tuning parameters:

- `bnum`: default 64K. Bucket number should be calculated by the number of pages, such that the bucket number is 10% of the total record count.
- `pccap`: page-cache capacity, default is 64MB. If there is additional RAM, this can be increased, but it is better to assign RAM using the internal memory-mapped region using the `msiz` parameter.
- `psiz`: default is 8192, specified before opening db and cannot be changed.
- `rcomp`: record comparator, default is lexical

Unlike the HashDB, the default alignment power (`apow`) is 256 (2^8), and the default bucket number is 64K.

## 4.2 Server configuration

Complete list of available options for running ktserver.

Note that the durability options may have a significant impact on performance:

- `-oat` - automatic transactions
- `-asi` / `-uasi` - automatic synchronization of database and update logs

- `-ash` - physical synchronization

Also refer to the "tips" document, which covers things like binary logging, snapshots, replication, etc.

Notes from alticelabs readme:

- Don't use the `capsiz` option with on-disk databases as the server will temporarily stop responding to free up space when the maximum capacity is reached. In this case, try to keep the database size under control using auto-expiring keys instead.
- On-disk databases are sensitive to disk write performance (impacting record updates as well as reads). Enabling transactions and/or synchronization makes this worse, as does increasing the number of buckets for hash databases (larger structures to write). Having a disk controller with some kind of battery-backed write-cache makes these issues mute.
- Choose your on-disk database tuning options carefully and don't tune unless you need to. Some options can be modified by a simple restart of the server (e.g. `pccap`, `msiz`) but others require creating the database from scratch (e.g. `bnum`, `opts=c`).
- Make sure you have enough disk space to store your on-disk databases as they grow. The server uses `mmap()` for file access and handles out-of-space conditions by terminating immediately. The database should still be consistent if this happens, so don't fret too much about it.
- The unique server ID (`-sid`) is used to break replication loops (a server instance ignores keys with its own SID). Keep this in mind when restoring failed master-master instances. The documentation recommends always choosing a new SID but this doesn't seem a good idea in this case. If the existing master still has keys from the failed master with the old SID pending replication, the new master with a new SID will propagate them back.

Examples:

Standalone b-tree database with compression and binary logging enabled. The `pccap=256m` option increases the default page-cache memory to 256mb:

```
$ /usr/local/bin/ktserver -ls -th 16 -port 1978 -pid /data/kyoto/kyoto.pid \
                          -log /data/kyoto/ktserver.log -oat -uasi 10 -asi 10 -ash \
                          -sid 1001 -ulog /data/kyoto/db -ulim 104857600 \
                          '/data/kyoto/db/db.kct#opts=c#pccap=256m#dfunit=8'
```

If you have a good idea of how many objects you are storing, you can use a persistent hash. The `bnum=1m` configures 1 million hash buckets (about 2x the number of expected keys), and `msiz=256m` sets the size of the memory-mapped region (larger is better, depending on availability of RAM).

```
$ /usr/local/bin/ktserver -ls -th 16 -port 1978 -pid /data/kyoto/kyoto.pid \
                          -log /data/kyoto/ktserver.log -oat -uasi 10 -asi 10 -ash \
                          -sid 1001 -ulog /data/kyoto/db -ulim 104857600 \
                          '/data/kyoto/db/db.kch#opts=l#bnum=1000000#msiz=256m
↪#dfunit=8'
```

In-memory cache limited to 256mb with LRU eviction:

```
$ /usr/local/bin/ktserver -log /var/log/ktserver.log -ls '*#bnum=100000#capsiz=256m'
```

To enable simultaneous support for the memcached protocol, use the `-plsv` and `-plex` options. The `opts=f` enables flags support for memcached, which are stored as the last 4 bytes of the value (take care when mixing protocols!).

```
$ /usr/local/bin/ktserver -log /var/log/ktserver.log -ls \
                          -plsv /usr/local/libexec/ktplugservmemc.so \
                          -plex 'port=11211#opts=f' \
                          '*#bnum=100000#capsiz=256m'
```

CHAPTER 5

# Indices and tables

- genindex
- modindex
- search

# Index

## Symbols



## A



## B



## C



## D



## E



## F



## G



## H

## U



## V