AIOHappyBase

Release 1.3.0+11.g9bab1d8.dirty

Jan 29, 2021

Contents

AIOHappyBase is a developer-friendly Python library to interact with Apache HBase. AIOHappyBase is designed for use in standard HBase setups, and offers application developers a Pythonic API to interact with HBase. Below the surface, AIOHappyBase uses the Python ThriftPy2 library to connect to HBase using its Thrift gateway, which is included in the standard HBase 0.9x releases.

Note: From the original HappyBase author, Wouter Bolsterlee:

Do you enjoy HappyBase? Great! You should know that I don't use HappyBase myself anymore, but still maintain it because it's quite popular. Please consider making a small donation to let me know you appreciate my work. Thanks!

CHAPTER 1

Example

The example below illustrates basic usage of the library. The *user guide* contains many more examples.

```
from aiohappybase import Connection

async def main():

    async with Connection('hostname') as connection:
        table = connection.table('table-name')

        await table.put(b'row-key', {
            b'family:qual1': b'value1',
            b'family:qual2': b'value2',
        })

        row = await table.row(b'row-key')
        print(row[b'family:qual1'])  # prints 'value1'

        for key, data in await table.rows([b'row-key-1', b'row-key-2']):
           print(key, data)  # prints row key and data for each row

        async for key, data in table.scan(row_prefix=b'row'):
           print(key, data)  # prints 'value1' and 'value2'

        await table.delete(b'row-key')
```

CHAPTER 2

Core documentation

2.1 Installation guide

This guide describes how to install HappyBase.

On this page

2.1.1 Setting up a virtual environment

The recommended way to install HappyBase and Thrift is to use a virtual environment created by *virtualenv*. Setup and activate a new virtual environment like this:

```
$ python -m venv venv_name
$ source venv_name/bin/activate
```

2.1.2 Installing the AIOHappyBase package

The next step is to install AIOHappyBase. The easiest way is to use *pip* to fetch the package from the Python Package Index (PyPI). This will also install the Thrift package for Python.

```
(venv_name) $ pip install aiohappybase
```

Note: Generating and installing the HBase Thrift Python modules (using `thrift --gen py` on the `.thrift` file) is not necessary, since AIOHappyBase bundles pregenerated versions of those modules.

If you are going to be using AIOHappyBase to communicate with an HBase server that uses Thrift over HTTP, you will need to install the `http` extra as well:

```
(venv_name) $ pip install aiohappybase[http]
```

2.1.3 Testing the installation

Verify that the packages are installed correctly:

```
(venv_name) $ python -c 'import aiohappybase'
```

If you don't see any errors, the installation was successful. Congratulations!

Next steps

Now that you successfully installed AIOHappyBase on your machine, continue with the *user guide* to learn how to use it.

2.2 User guide

This user guide explores the AIOHappyBase API and should provide you with enough information to get you started. Note that this user guide is intended as an introduction to AIOHappyBase, not to HBase in general. Readers should already have a basic understanding of HBase and its data model.

While the user guide does cover most features, it is not a complete reference guide. More information about the AIOHappyBase API is available from the *API documentation*.

On this page

2.2.1 Establishing a connection

We'll get started by connecting to HBase. Just create a new *Connection* instance:

```
from aiohappybase import Connection

connection = Connection('somehost')

# Or better, to ensure the connection is closed later

async with Connection('somehost') as connection:
    # Do your thing
```

In some setups, the *Connection* class needs some additional information about the HBase version it will be connecting to, and which Thrift transport to use. If you're still using HBase 0.90.x, you need to set the *compat* argument to make sure AIOHappyBase speaks the correct wire protocol. Additionally, if you're using HBase 0.94 with a non-standard Thrift transport mode, make sure to supply the right *transport* argument. See the API documentation for the *Connection* class for more information about these arguments and their supported values.

When a *Connection* is created, it can automatically open a socket connection to the HBase Thrift server if *autoconnect* is set to True or the *Connection* is created using a context.

The *Connection* class provides the main entry point to interact with HBase. For instance, to list the available tables, use *Connection.tables()*:

```
print(await connection.tables())
```

Most other methods on the *Connection* class are intended for system management tasks like creating, dropping, enabling and disabling tables. See the *API documentation* for the *Connection* class contains more information. This user guide does not cover those since it's more likely you are already using the HBase shell for these system management tasks.

Note: AIOHappyBase also features a connection pool, which is covered later in this guide.

2.2.2 Working with tables

The *Table* class provides the main API to retrieve and manipulate data in HBase. In the example above, we already asked for the available tables using the *Connection.tables()* method. If there weren't any tables yet, you can create a new one using *Connection.create_table()*:

```
table = await connection.create_table('mytable', {
    'cf1': dict(max_versions=10),
    'cf2': dict(max_versions=1, block_cache_enabled=False),
```

(continues on next page)

(continued from previous page)

```
    'cf3': dict(),  # use defaults
})
```

Note: The HBase shell is often a better alternative for many HBase administration tasks, since the shell is more powerful compared to the limited Thrift API that AIOHappyBase uses.

If the table already exists, you can get a *Table* instance to work with by simply calling *Connection.table()*, and passing it the table name:

```
table = connection.table('mytable')
```

Note that this method *is not async*. Obtaining a *Table* instance does *not* result in a round-trip to the Thrift server, which means application code may ask the *Connection* instance for a new *Table* whenever it needs one, without negative performance consequences. A side effect is that no check is done to ensure that the table exists, since that would involve a round-trip. Expect errors if you try to interact with non-existing tables later in your code. For this guide, we assume the table exists.

Note: The 'heavy' *HTable* HBase class from the Java HBase API, which performs the real communication with the region servers, is at the other side of the Thrift connection. There is no direct mapping between *Table* instances on the Python side and *HTable* instances on the server side.

Using table 'namespaces'

If a single HBase instance is shared by multiple applications, table names used by different applications may collide. A simple solution to this problem is to add a 'namespace' prefix to the names of all tables 'owned' by a specific application, e.g. for a project `myproject` all tables have names like `myproject_XYZ`.

Instead of adding this application-specific prefix each time a table name is passed to AIOHappyBase, the *table_prefix* argument to *Connection* can take care of this. AIOHappyBase will prepend that prefix (and an underscore) to each table name handled by that *Connection* instance. For example:

```
connection = Connection('somehost', table_prefix='myproject')
```

At this point, *Connection.tables()* no longer includes tables in other 'namespaces'. AIOHappyBase will only return tables with a `myproject_` prefix, and will also remove the prefix transparently when returning results, e.g.:

```
print(await connection.tables())  # Table "myproject_XYZ" in HBase will be
                                  # returned as simply "XYZ"
```

This also applies to other methods that take table names, such as *Connection.table()*:

```
table = connection.table('XYZ')  # Operates on myproject_XYZ in HBase
```

The end result is that the table prefix is specified only once in your code, namely in the call to the *Connection* constructor, and that only a single change is necessary in case it needs changing.

2.2.3 Retrieving data

The HBase data model is a multidimensional sparse map. A table in HBase contains column families with column qualifiers containing a value and a timestamp. In most of the AIOHappyBase API, column family and qualifier names

are specified as a single string, e.g. `cf1:col1`, and not as two separate arguments. While column families and qualifiers are different concepts in the HBase data model, they are almost always used together when interacting with data, so treating them as a single string makes the API a lot simpler.

Retrieving rows

The *Table* class offers various methods to retrieve data from a table in HBase. The most basic one is *Table.row()*, which retrieves a single row from the table, and returns it as a dictionary mapping columns to values:

```
row = await table.row(b'row-key')
print(row[b'cf1:col1'])    # prints the value of cf1:col1
```

The *Table.rows()* method works just like *Table.row()*, but takes multiple row keys and returns those as *(key, data)* tuples:

```
rows = await table.rows([b'row-key-1', b'row-key-2'])
for key, data in rows:
    print(key, data)
```

If you want the results that *Table.rows()* returns as a dictionary, you will have to do this yourself. This is really easy though, since the return value can be passed directly to the dictionary constructor. As of Python 3.6 order is not lost when creating a dictionary:

```
rows_as_dict = dict(await table.rows([b'row-key-1', b'row-key-2']))
```

Making more fine-grained selections

HBase's data model allows for more fine-grained selections of the data to retrieve. If you know beforehand which columns are needed, performance can be improved by specifying those columns explicitly to *Table.row()* and *Table.rows()*. The *columns* argument takes a list (or tuple) of column names:

```
row = await table.row(b'row-key', columns=[b'cf1:col1', b'cf1:col2'])
print(row[b'cf1:col1'])
print(row[b'cf1:col2'])
```

Instead of providing both a column family and a column qualifier, items in the *columns* argument may also be just a column family, which means that all columns from that column family will be retrieved. For example, to get all columns and values in the column family *cf1*, use this:

```
row = await table.row(b'row-key', columns=[b'cf1'])
```

In HBase, each cell has a timestamp attached to it. In case you don't want to work with the latest version of data stored in HBase, the methods that retrieve data from the database, e.g. *Table.row()*, all accept a *timestamp* argument that specifies that the results should be restricted to values with a timestamp up to the specified timestamp:

```
row = await table.row(b'row-key', timestamp=123456789)
```

By default, AIOHappyBase does not include timestamps in the results it returns. In your application needs access to the timestamps, simply set the *include_timestamp* argument to `True`. Now, each cell in the result will be returned as a *(value, timestamp)* tuple instead of just a value:

```
row = await table.row(b'row-key', columns=[b'cf1:col1'], include_timestamp=True)
value, timestamp = row[b'cf1:col1']
```

HBase supports storing multiple versions of the same cell. This can be configured for each column family. To retrieve all versions of a column for a given row, *Table.cells()* can be used. This method returns an ordered list of cells, with the most recent version coming first. The *versions* argument specifies the maximum number of versions to return. Just like the methods that retrieve rows, the *include_timestamp* argument determines whether timestamps are included in the result. Example:

```
values = await table.cells(b'row-key', b'cf1:col1', versions=2)
for value in values:
    print("Cell data: {}".format(value))

cells = await table.cells(b'row-key', b'cf1:col1', versions=3, include_timestamp=True)
for value, timestamp in cells:
    print("Cell data at {}: {}".format(timestamp, value))
```

Note that the result may contain fewer cells than requested. The cell may just have fewer versions, or you may have requested more versions than HBase keeps for the column family.

Scanning over rows in a table

In addition to retrieving data for known row keys, rows in HBase can be efficiently iterated over using a table scanner, created using *Table.scan()*. A basic scanner that iterates over all rows in the table looks like this:

```
async for key, data in table.scan():
    print(key, data)
```

Doing full table scans like in the example above is prohibitively expensive in practice. Scans can be restricted in several ways to make more selective range queries. One way is to specify start or stop keys, or both. To iterate over all rows from row *aaa* to the end of the table:

```
async for key, data in table.scan(row_start=b'aaa'):
    print(key, data)
```

To iterate over all rows from the start of the table up to row *xyz*, use this:

```
async for key, data in table.scan(row_stop=b'xyz'):
    print(key, data)
```

To iterate over all rows between row *aaa* (included) and *xyz* (not included), supply both:

```
async for key, data in table.scan(row_start=b'aaa', row_stop=b'xyz'):
    print(key, data)
```

An alternative is to use a key prefix. For example, to iterate over all rows starting with *abc*:

```
async for key, data in table.scan(row_prefix=b'abc'):
    print(key, data)
```

The scanner examples above only limit the results by row key using the *row_start*, *row_stop*, and *row_prefix* arguments, but scanners can also limit results to certain columns, column families, and timestamps, just like *Table.row()* and *Table.rows()*. For advanced users, a filter string can be passed as the *filter* argument. Additionally, the optional *limit* argument defines how much data is at most retrieved, and the *batch_size* argument specifies how big the transferred chunks should be. The *Table.scan()* API documentation provides more information on the supported scanner options.

2.2.4 Manipulating data

HBase does not have any notion of *data types*; all row keys, column names and column values are simply treated as raw byte strings.

By design, AIOHappyBase does *not* do any automatic string conversion. This means that data must be converted to byte strings in your application before you pass it to AIOHappyBase, for instance by calling `s.encode('utf-8')` on text strings (which use Unicode), or by employing more advanced string serialisation techniques like `struct.pack()`. Look for HBase modelling techniques for more details about this. Note that the underlying Thrift library used by AIOHappyBase does some automatic encoding of text strings into bytes, but relying on this "feature" is strongly discouraged, since returned data will not be decoded automatically, resulting in asymmetric and hence confusing behaviour. Having explicit encode and decode steps in your application code is the correct way.

In HBase, all mutations either store data or mark data for deletion; there is no such thing as an in-place *update* or *delete*. AIOHappyBase provides methods to do single inserts or deletes, and a batch API to perform multiple mutations in one go.

Storing data

To store a single cell of data in our table, we can use *Table.put()*, which takes the row key, and the data to store. The data should be a dictionary mapping the column name to a value:

```
await table.put(b'row-key', {b'cf:col1': b'value1', b'cf:col2': b'value2'})
```

Use the *timestamp* argument if you want to provide timestamps explicitly:

```
await table.put(b'row-key', {b'cf:col1': b'value1'}, timestamp=123456789)
```

If omitted, HBase defaults to the current system time.

Deleting data

The *Table.delete()* method deletes data from a table. To delete a complete row, just specify the row key:

```
await table.delete(b'row-key')
```

To delete one or more columns instead of a complete row, also specify the *columns* argument:

```
await table.delete(b'row-key', columns=[b'cf1:col1', b'cf1:col2'])
```

The optional *timestamp* argument restricts the delete operation to data up to the specified timestamp.

Performing batch mutations

The *Table.put()* and *Table.delete()* methods both issue a command to the HBase Thrift server immediately. This means that using these methods is not very efficient when storing or deleting multiple values. It is much more efficient to aggregate a bunch of commands and send them to the server in one go. This is exactly what the *Batch* class, created using *Table.batch()*, does. A *Batch* instance has put and delete methods, just like the *Table* class, but the changes are sent to the server in a single round-trip using *Batch.send()*:

```
b = table.batch()
await b.put(b'row-key-1', {b'cf:col1': b'value1', b'cf:col2': b'value2'})
await b.put(b'row-key-2', {b'cf:col2': b'value2', b'cf:col3': b'value3'})
await b.put(b'row-key-3', {b'cf:col3': b'value3', b'cf:col4': b'value4'})
```

(continues on next page)

(continued from previous page)

```
await b.delete(b'row-key-4')
await b.send()
```

Note: Storing and deleting data for the same row key in a single batch leads to unpredictable results, so don't do that.

While the methods on the *Batch* instance resemble the *put()* and *delete()* methods, they do not take a *timestamp* argument for each mutation. Instead, you can specify a single *timestamp* argument for the complete batch:

```
b = table.batch(timestamp=123456789)
await b.put(...)
await b.delete(...)
await b.send()
```

Batch instances can be used as *context managers*, which are most useful in combination with Python's `with` construct. The example above can be simplified to read:

```
async with table.batch() as b:
    await b.put(b'row-key-1', {b'cf:col1': b'value1', b'cf:col2': b'value2'})
    await b.put(b'row-key-2', {b'cf:col2': b'value2', b'cf:col3': b'value3'})
    await b.put(b'row-key-3', {b'cf:col3': b'value3', b'cf:col4': b'value4'})
    await b.delete(b'row-key-4')
```

As you can see, there is no call to *Batch.send()* anymore. The batch is automatically applied when the `with` code block terminates, even in case of errors somewhere in the `with` block, so it behaves basically the same as a `try/finally` clause. However, some applications require transactional behaviour, sending the batch only if no exception occurred. Without a context manager this would look something like this:

```
b = table.batch()
try:
     await b.put(b'row-key-1', {b'cf:col1': b'value1', b'cf:col2': b'value2'})
     await b.put(b'row-key-2', {b'cf:col2': b'value2', b'cf:col3': b'value3'})
     await b.put(b'row-key-3', {b'cf:col3': b'value3', b'cf:col4': b'value4'})
     await b.delete(b'row-key-4')
     raise ValueError("Something went wrong!")
except ValueError as e:
     # error handling goes here; nothing will be sent to HBase
     pass
else:
     # no exceptions; send data
     await b.send()
```

Obtaining the same behaviour is easier using a `with` block. The *transaction* argument to *Table.batch()* is all you need:

```
try:
     async with table.batch(transaction=True) as b:
          await b.put(b'row-key-1', {b'cf:col1': b'value1', b'cf:col2': b'value2'})
          await b.put(b'row-key-2', {b'cf:col2': b'value2', b'cf:col3': b'value3'})
          await b.put(b'row-key-3', {b'cf:col3': b'value3', b'cf:col4': b'value4'})
          await b.delete(b'row-key-4')
          raise ValueError("Something went wrong!")
except ValueError:
     # error handling goes here; nothing is sent to HBase
     pass
```

(continues on next page)

(continued from previous page)

```
# when no error occurred, the transaction succeeded
```

As you may have imagined already, a *Batch* keeps all mutations in memory until the batch is sent, either by calling *Batch.send()* explicitly, or when the `with` block ends. This doesn't work for applications that need to store huge amounts of data, since it may result in batches that are too big to send in one round-trip, or in batches that use too much memory. For these cases, the *batch_size* argument can be specified. The *batch_size* acts as a threshold: a *Batch* instance automatically sends all pending mutations when there are more than *batch_size* pending operations. For example, this will result in three round-trips to the server (two batches with 1000 cells, and one with the remaining 400):

```
async with table.batch(batch_size=1000) as b:
    for i in range(1200):
        # this put() will result in two mutations (two cells)
        await b.put(b'row-%04d' % i, {
            b'cf1:col1': b'v1',
            b'cf1:col2': b'v2',
        })
```

The appropriate *batch_size* is very application-specific since it depends on the data size, so just experiment to see how different sizes work for your specific use case.

Using atomic counters

The *Table.counter_inc()* and *Table.counter_dec()* methods allow for atomic incrementing and decrementing of 8 byte wide values, which are interpreted as big-endian 64-bit signed integers by HBase. Counters are automatically initialised to 0 upon first use. When incrementing or decrementing a counter, the value after modification is returned. Example:

```
print(await table.counter_inc(b'row-key', b'cf1:counter'))  # prints 1
print(await table.counter_inc(b'row-key', b'cf1:counter'))  # prints 2
print(await table.counter_inc(b'row-key', b'cf1:counter'))  # prints 3

print(await table.counter_dec(b'row-key', b'cf1:counter'))  # prints 2
```

The optional *value* argument specifies how much to increment or decrement by:

```
print(await table.counter_inc(b'row-key', b'cf1:counter', value=3))  # prints 5
```

While counters are typically used with the increment and decrement functions shown above, the *Table.counter_get()* and *Table.counter_set()* methods can be used to retrieve or set a counter value directly:

```
print(await table.counter_get(b'row-key', b'cf1:counter'))  # prints 5

await table.counter_set(b'row-key', b'cf1:counter', 12)
```

Note: An application should *never* *counter_get()* the current value, modify it in code and then *counter_set()* the modified value; use the atomic *counter_inc()* and *counter_dec()* instead!

2.2.5 Using the connection pool

AIOHappyBase comes with a asyncio task-safe connection pool that allows multiple tasks to share and reuse open connections. This is most useful in multi-tasked server applications such as `aiohttp` servers. When a task asks the pool for a connection (using *ConnectionPool.connection()*), it will be granted a lease, during which the task has exclusive access to the connection. After the task is done using the connection, it returns the connection to the pool so that it becomes available for other tasks.

Instantiating the pool

The pool is provided by the *ConnectionPool* class. The *size* argument to the constructor specifies the number of connections in the pool. Additional arguments are passed on to the *Connection* constructor:

```
from aiohappybase import ConnectionPool

pool = ConnectionPool(size=3, host='...', table_prefix='myproject')

# Context instantiation is preferred to make sure connections are cleaned up
async with ConnectionPool(size=3, host='...', table_prefix='myproject') as pool:
    # Do your thing
```

Connections will only be opened as necessary. HappyBase's ConnectionPool would open a single connection immediately to detect issues, but that isn't so easy to do in async because __init__ is always synchronous.

Obtaining connections

Connections can only be obtained using Python's context manager protocol, i.e. using a code block inside a `with` statement. This ensures that connections are actually returned to the pool after use. Example:

```
async with .ConnectionPool(size=3, host='...') as pool:
    async with pool.connection() as connection:
        print(await connection.tables())
```

> **Warning:** Never use the `connection` instance after the `with` block has ended. Even though the variable is still in scope, the connection may have been assigned to another task in the mean time.

Connections should be returned to the pool as quickly as possible, so that other tasks can use them. This means that the amount of code included inside the `with` block should be kept to an absolute minimum. In practice, an application should only load data inside the `with` block, and process the data outside the `with` block:

```
async with pool.connection() as connection:
    table = connection.table('table-name')
    row = await table.row(b'row-key')

process_data(row)
```

An application task can only hold one connection at a time. When a task holds a connection and asks for a connection for a second time (e.g. because a called function also requests a connection from the pool), the same connection instance it already holds is returned, so this does not require any coordination from the application. This means that in the following example, both connection requests to the pool will return the exact same connection:

```
pool = ConnectionPool(size=3, host='...')

async def do_something_else():
    async with pool.connection() as connection:
        pass  # use the connection here

async with pool.connection() as connection:
    # use the connection here, e.g.
    print(await connection.tables())

    # call another function that uses a connection
    do_something_else()

await pool.close()
```

Handling broken connections

The pool tries to detect broken connections and will replace those with fresh ones when the connection is returned to the pool. However, the connection pool does not capture raised exceptions, nor does it automatically retry failed operations. This means that the application still has to handle connection errors.

Next steps

The next step is to try it out for yourself! The *API documentation* can be used as a reference.

2.3 API reference

This chapter contains detailed API documentation for AIOHappyBase. It is suggested to read the *user guide* first to get a general idea about how AIOHappyBase works.

The AIOHappyBase API is organised as follows:

***Connection*:** The *Connection* class is the main entry point for application developers. It connects to the HBase Thrift server and provides methods for table management.

***Table*:** The *Table* class is the main class for interacting with data in tables. This class offers methods for data retrieval and data manipulation. Instances of this class can be obtained using the *Connection.table()* method.

***Batch*:** The *Batch* class implements the batch API for data manipulation, and is available through the *Table.batch()* method.

***ConnectionPool*:** The *ConnectionPool* class implements a thread-safe connection pool that allows an application to (re)use multiple connections.

2.3.1 Connection

class aiohappybase.**Connection**(*host: str = 'localhost', port: int = 9090, timeout: int = None, autoconnect: bool = False, table_prefix: AnyStr = None, table_prefix_separator: AnyStr = b'_', compat: str = '0.98', transport: str = 'buffered', protocol: str = 'binary', client: str = 'socket', **client_kwargs*)

Connection to an HBase Thrift server.

The *host* and *port* arguments specify the host name and TCP port of the HBase Thrift server to connect to. If omitted or `None`, a connection to the default port on `localhost` is made. If specifed, the *timeout* argument specifies the socket timeout in milliseconds.

If *autoconnect* is *True* the connection is made directly during initialization. Otherwise a context manager should be used (with Connection...) or *Connection.open()* must be called explicitly before first use. Note that due to limitations in the Python async framework, a RuntimeError will be raised if it is used inside of a running asyncio event loop.

The optional *table_prefix* and *table_prefix_separator* arguments specify a prefix and a separator string to be prepended to all table names, e.g. when *Connection.table()* is invoked. For example, if *table_prefix* is `myproject`, all tables will have names like `myproject_XYZ`.

The optional *compat* argument sets the compatibility level for this connection. Older HBase versions have slightly different Thrift interfaces, and using the wrong protocol can lead to crashes caused by communication errors, so make sure to use the correct one. This value can be either the string `0.90`, `0.92`, `0.94`, or `0.96` (the default).

The optional *transport* argument specifies the Thrift transport mode to use. Supported values for this argument are `buffered` (the default) and `framed`. Make sure to choose the right one, since otherwise you might see non-obvious connection errors or program hangs when making a connection. HBase versions before 0.94 always use the buffered transport. Starting with HBase 0.94, the Thrift server optionally uses a framed transport, depending on the argument passed to the `hbase-daemon.sh start thrift` command. The default `-threadpool` mode uses the buffered transport; the `-hsha`, `-nonblocking`, and `-threadedselector` modes use the framed transport.

The optional *protocol* argument specifies the Thrift transport protocol to use. Supported values for this argument are `binary` (the default) and `compact`. Make sure to choose the right one, since otherwise you might see non-obvious connection errors or program hangs when making a connection. `TCompactProtocol` is a more compact binary format that is typically more efficient to process as well. `TBinaryProtocol` is the default protocol that AIOHappyBase uses.

The optional *client* argument specifies the type of Thrift client to use. Supported values for this argument are `socket` (the default) and `http`. Make sure to choose the right one, since otherwise you might see non-obvious connection errors or program hangs when making a connection. To check which client you should use, refer to the `hbase.regionserver.thrift.http` setting. If it is `true` use `http`, otherwise use `socket`.

New in version v1.4.0: *client* argument

New in version 0.9: *protocol* argument

New in version 0.5: *timeout* argument

New in version 0.4: *table_prefix_separator* argument

New in version 0.4: support for framed Thrift transports

Parameters

- **host** – The host to connect to
- **port** – The port to connect to
- **timeout** – The socket timeout in milliseconds (optional)
- **autoconnect** – Whether the connection should be opened directly
- **table_prefix** – Prefix used to construct table names (optional)
- **table_prefix_separator** – Separator used for *table_prefix*
- **compat** – Compatibility mode (optional)
- **transport** – Thrift transport mode (optional)

- **protocol** – Thrift protocol mode (optional)
- **client** – Thrift client mode (optional)
- **client_kwargs** – Extra keyword arguments for *make_client()*. See the ThriftPy2 documentation for more information.

close() → None
Close the underlying client to the HBase instance. This method can be safely called more than once. Note that the client is destroyed after it is closed which will cause errors to occur if it is used again before reopening. The *Connection* can be reopened by calling *open()* again.

table(*name: AnyStr*, *use_prefix: bool = True*) → aiohappybase.table.Table
Return a table object.

Returns a `happybase.Table` instance for the table named *name*. This does not result in a round-trip to the server, and the table is not checked for existence.

The optional *use_prefix* argument specifies whether the table prefix (if any) is prepended to the specified *name*. Set this to *False* if you want to use a table that resides in another 'prefix namespace', e.g. a table from a 'friendly' application co-hosted on the same HBase instance. See the *table_prefix* argument to the *Connection* constructor for more information.

Parameters

- **name** – the name of the table
- **use_prefix** – whether to use the table prefix (if any)

Returns Table instance

compact_table(*name: AnyStr*, *major: bool = False*) → None
Compact the specified table.

Parameters

- **name** (*str*) – The table name
- **major** (*bool*) – Whether to perform a major compaction.

create_table(*name: AnyStr, families: Dict[str, Dict[str, Any]]*) → aiohappybase.table.Table
Create a table.

Parameters

- **name** – The table name
- **families** – The name and options for each column family

Returns The created table instance

The *families* argument is a dictionary mapping column family names to a dictionary containing the options for this column family, e.g.

```
families = {
    'cf1': dict(max_versions=10),
    'cf2': dict(max_versions=1, block_cache_enabled=False),
    'cf3': dict(),  # use defaults
}
connection.create_table('mytable', families)
```

These options correspond to the ColumnDescriptor structure in the Thrift API, but note that the names should be provided in Python style, not in camel case notation, e.g. *time_to_live*, not *timeToLive*. The following options are supported:

- `max_versions` (*int*)
- `compression` (*str*)
- `in_memory` (*bool*)
- `bloom_filter_type` (*str*)
- `bloom_filter_vector_size` (*int*)
- `bloom_filter_nb_hashes` (*int*)
- `block_cache_enabled` (*bool*)
- `time_to_live` (*int*)

delete_table(*name: AnyStr*, *disable: bool = False*) → None
Delete the specified table.

New in version 0.5: *disable* argument

In HBase, a table always needs to be disabled before it can be deleted. If the *disable* argument is *True*, this method first disables the table if it wasn't already and then deletes it.

Parameters

- **name** – The table name
- **disable** – Whether to first disable the table if needed

disable_table(*name: AnyStr*) → None
Disable the specified table.

Parameters **name** – The table name

enable_table(*name: AnyStr*) → None
Enable the specified table.

Parameters **name** – The table name

is_table_enabled(*name: AnyStr*) → None
Return whether the specified table is enabled.

Parameters **name** (*str*) – The table name

Returns whether the table is enabled

Return type bool

open() → None
Create and open the underlying client to the HBase instance. This method can safely be called more than once.

tables() → List[bytes]
Return a list of table names available in this HBase instance.

If a *table_prefix* was set for this *Connection*, only tables that have the specified prefix will be listed.

Returns The table names

2.3.2 Table

class aiohappybase.**Table**(*name: bytes*, *connection: Connection*)
HBase table abstraction class.

This class cannot be instantiated directly; use *Connection.table()* instead.

regions() → List[Dict[str, Any]]
Retrieve the regions for this table.

Returns regions for this table

rows(*rows: List[bytes], columns: Iterable[bytes] = None, timestamp: int = None, include_timestamp: bool = False*) → List[Tuple[bytes, Union[Dict[bytes, bytes], Dict[bytes, Tuple[bytes, int]]]]]
Retrieve multiple rows of data.

This method retrieves the rows with the row keys specified in the *rows* argument, which should be should be a list (or tuple) of row keys. The return value is a list of *(row_key, row_dict)* tuples.

The *columns*, *timestamp* and *include_timestamp* arguments behave exactly the same as for *row()*.

Parameters

- **rows** – list of row keys
- **columns** – list of columns (optional)
- **timestamp** – timestamp (optional)
- **include_timestamp** – whether timestamps are returned

Returns List of mappings (columns to values)

scan(*row_start: bytes = None*, *row_stop: bytes = None*, *row_prefix: bytes = None*, *columns: Iterable[bytes] = None*, *filter: bytes = None*, *timestamp: int = None*, *include_timestamp: bool = False*, *batch_size: int = 1000*, *scan_batching: int = None*, *limit: int = None*, *sorted_columns: bool = False*, *reverse: bool = False*) → AsyncGenerator[Tuple[bytes, Dict[bytes, bytes]], None]
Create a scanner for data in the table.

This method returns an iterable that can be used for looping over the matching rows. Scanners can be created in two ways:

- The *row_start* and *row_stop* arguments specify the row keys where the scanner should start and stop. It does not matter whether the table contains any rows with the specified keys: the first row after *row_start* will be the first result, and the last row before *row_stop* will be the last result. Note that the start of the range is inclusive, while the end is exclusive.

  Both *row_start* and *row_stop* can be *None* to specify the start and the end of the table respectively. If both are omitted, a full table scan is done. Note that this usually results in severe performance problems.

- Alternatively, if *row_prefix* is specified, only rows with row keys matching the prefix will be returned. If given, *row_start* and *row_stop* cannot be used.

The *columns*, *timestamp* and *include_timestamp* arguments behave exactly the same as for *row()*.

The *filter* argument may be a filter string that will be applied at the server by the region servers.

If *limit* is given, at most *limit* results will be returned.

The *batch_size* argument specifies how many results should be retrieved per batch when retrieving results from the scanner. Only set this to a low value (or even 1) if your data is large, since a low batch size results in added round-trips to the server.

The optional *scan_batching* is for advanced usage only; it translates to *Scan.setBatching()* at the Java side (inside the Thrift server). By setting this value rows may be split into partial rows, so result rows may be incomplete, and the number of results returned by the scanner may no longer correspond to the number of rows matched by the scan.

If *sorted_columns* is *True*, the columns in the rows returned by this scanner will be retrieved in sorted order, and the data will be stored in *OrderedDict* instances.

If *reverse* is *True*, the scanner will perform the scan in reverse. This means that *row_start* must be lexicographically after *row_stop*. Note that the start of the range is inclusive, while the end is exclusive just as in the forward scan.

Compatibility notes:

- The *filter* argument is only available when using HBase 0.92 (or up). In HBase 0.90 compatibility mode, specifying a *filter* raises an exception.
- The *sorted_columns* argument is only available when using HBase 0.96 (or up).
- The *reverse* argument is only available when using HBase 0.98 (or up).

New in version 1.1.0: *reverse* argument

New in version 0.8: *sorted_columns* argument

New in version 0.8: *scan_batching* argument

Parameters

- **row_start** – the row key to start at (inclusive)
- **row_stop** – the row key to stop at (exclusive)
- **row_prefix** – a prefix of the row key that must match
- **columns** – list of columns (optional)
- **filter** – a filter string (optional)
- **timestamp** – timestamp (optional)
- **include_timestamp** – whether timestamps are returned
- **batch_size** – batch size for retrieving results
- **scan_batching** – server-side scan batching (optional)
- **limit** – max number of rows to return
- **sorted_columns** – whether to return sorted columns
- **reverse** – whether to perform scan in reverse

Returns generator yielding the rows matching the scan

Return type iterable of *(row_key, row_data)* tuples

append(*row: bytes, data: Dict[bytes, bytes], include_timestamp: bool = False*) → Union[Dict[bytes, bytes], Dict[bytes, Tuple[bytes, int]]]

Append data to an existing row.

- This function is only available when using HBase 0.98 (or up).

The *data* argument behaves just like it does in *put()* except that instead of replacing the current values, they are appended to the end. If a specified cell doesn't exist, then the result is the same as calling *put()* for that cell.

Parameters

- **row** – the row key
- **data** – data to append
- **include_timestamp** – include timestamps with the values?

Returns Updated cell values like the output of *row()*

column_family_names() → List[bytes]
Retrieve the column family names for this table

counter_dec(*row: bytes*, *column: bytes*, *value: int = 1*) → int
Atomically decrement (or increments) a counter column.

This method is a shortcut for calling *Table.counter_inc()* with the value negated.

Returns counter value after decrementing

counter_get(*row: bytes*, *column: bytes*) → int
Retrieve the current value of a counter column.

This method retrieves the current value of a counter column. If the counter column does not exist, this function initialises it to *0*.

Note that application code should *never* store a incremented or decremented counter value directly; use the atomic *Table.counter_inc()* and *Table.counter_dec()* methods for that.

Parameters

- **row** – the row key
- **column** – the column name

Returns counter value

counter_inc(*row: bytes*, *column: bytes*, *value: int = 1*) → int
Atomically increment (or decrements) a counter column.

This method atomically increments or decrements a counter column in the row specified by *row*. The *value* argument specifies how much the counter should be incremented (for positive values) or decremented (for negative values). If the counter column did not exist, it is automatically initialised to 0 before incrementing it.

Parameters

- **row** – the row key
- **column** – the column name
- **value** – the amount to increment or decrement by (optional)

Returns counter value after incrementing

counter_set(*row: bytes*, *column: bytes*, *value: int = 0*) → None
Set a counter column to a specific value.

This method stores a 64-bit signed integer value in the specified column.

Note that application code should *never* store a incremented or decremented counter value directly; use the atomic *Table.counter_inc()* and *Table.counter_dec()* methods for that.

Parameters

- **row** – the row key
- **column** – the column name
- **value** – the counter value to set

delete(*row: bytes*, *columns: Iterable[bytes] = None*, *timestamp: int = None*, *wal: bool = True*) → None
Delete data from the table.

This method deletes all columns for the row specified by *row*, or only some columns if the *columns* argument is specified.

Note that, in many situations, `batch()` is a more appropriate method to manipulate data.

New in version 0.7: *wal* argument

Parameters

- **row** – the row key
- **columns** – list of columns (optional)
- **timestamp** – timestamp (optional)
- **wal** – whether to write to the WAL (optional)

families() → Dict[bytes, Dict[str, Any]]
Retrieve the column families for this table.

Returns Mapping from column family name to settings dict

put(*row: bytes, data: Dict[bytes, bytes], timestamp: int = None, wal: bool = True*) → None
Store data in the table.

This method stores the data in the *data* argument for the row specified by *row*. The *data* argument is dictionary that maps columns to values. Column names must include a family and qualifier part, e.g. `b'cf:col'`, though the qualifier part may be the empty string, e.g. `b'cf:'`.

Note that, in many situations, `batch()` is a more appropriate method to manipulate data.

New in version 0.7: *wal* argument

Parameters

- **row** – the row key
- **data** – the data to store
- **timestamp** – timestamp (optional)
- **wal** – whether to write to the WAL (optional)

row(*row: bytes*, *columns: Iterable[bytes] = None*, *timestamp: int = None*, *include_timestamp: bool = False*) → Union[Dict[bytes, bytes], Dict[bytes, Tuple[bytes, int]]]
Retrieve a single row of data.

This method retrieves the row with the row key specified in the *row* argument and returns the columns and values for this row as a dictionary.

The *row* argument is the row key of the row. If the *columns* argument is specified, only the values for these columns will be returned instead of all available columns. The *columns* argument should be a list or tuple containing byte strings. Each name can be a column family, such as `b'cf1'` or `b'cf1:'` (the trailing colon is not required), or a column family with a qualifier, such as `b'cf1:col1'`.

If specified, the *timestamp* argument specifies the maximum version that results may have. The *include_timestamp* argument specifies whether cells are returned as single values or as *(value, timestamp)* tuples.

Parameters

- **row** – the row key
- **columns** – list of columns (optional)
- **timestamp** – timestamp (optional)
- **include_timestamp** – whether timestamps are returned

Returns Mapping of columns (both qualifier and family) to values

cells(*row: bytes*, *column: bytes*, *versions: int = None*, *timestamp: int = None*, *include_timestamp: bool = False*) → Union[List[bytes], List[Tuple[bytes, int]]]

Retrieve multiple versions of a single cell from the table.

This method retrieves multiple versions of a cell (if any).

The *versions* argument defines how many cell versions to retrieve at most.

The *timestamp* and *include_timestamp* arguments behave exactly the same as for *row()*.

Parameters

- **row** – the row key
- **column** – the column name
- **versions** – the maximum number of versions to retrieve
- **timestamp** – timestamp (optional)
- **include_timestamp** – whether timestamps are returned

Returns cell values

batch(*timestamp: int = None*, *batch_size: int = None*, *transaction: bool = False*, *wal: bool = True*) → aiohappybase.batch.Batch

Create a new batch operation for this table.

This method returns a new *Batch* instance that can be used for mass data manipulation. The *timestamp* argument applies to all puts and deletes on the batch.

If given, the *batch_size* argument specifies the maximum batch size after which the batch should send the mutations to the server. By default this is unbounded.

The *transaction* argument specifies whether the returned *Batch* instance should act in a transaction-like manner when used as context manager in a `with` block of code. The *transaction* flag cannot be used in combination with *batch_size*.

The *wal* argument determines whether mutations should be written to the HBase Write Ahead Log (WAL). This flag can only be used with recent HBase versions. If specified, it provides a default for all the put and delete operations on this batch. This default value can be overridden for individual operations using the *wal* argument to *Batch.put()* and *Batch.delete()*.

New in version 0.7: *wal* argument

Parameters

- **transaction** – whether this batch should behave like a transaction (only useful when used as a context manager)
- **batch_size** – batch size (optional)
- **timestamp** – timestamp (optional)
- **wal** – whether to write to the WAL (optional)

Returns Batch instance

2.3.3 Batch

class aiohappybase.**Batch**(*table: Table*, *timestamp: int = None*, *batch_size: int = None*, *transaction: bool = False*, *wal: bool = True*)

Batch mutation class.

This class cannot be instantiated directly; use *Table.batch()* instead.

Initialise a new Batch instance.

put(*row: bytes, data: Dict[bytes, bytes], wal: bool = None*) → None
Store data in the table.

See *Table.put()* for a description of the *row*, *data*, and *wal* arguments. The *wal* argument should normally not be used; its only use is to override the batch-wide value passed to *Table.batch()*.

delete(*row: bytes*, *columns: Iterable[bytes] = None*, *wal: bool = None*) → None
Delete data from the table.

See *Table.put()* for a description of the *row*, *data*, and *wal* arguments. The *wal* argument should normally not be used; its only use is to override the batch-wide value passed to *Table.batch()*.

close() → None
Finalize the batch and make sure all tasks are completed.

counter_inc(*row: bytes*, *column: bytes*, *value: int = 1*) → None
Atomically increment (or decrements) a counter column.

See *Table.counter_inc()* for parameter details. Note that this method cannot return the current value because the change is buffered until send to the server.

send() → None
Send the batch to the server.

counter_dec(*row: bytes*, *column: bytes*, *value: int = 1*) → None
Atomically decrement (or increments) a counter column.

See *Table.counter_dec()* for parameter details. Note that this method cannot return the current value because the change is buffered until send to the server.

2.3.4 Connection pool

class aiohappybase.**ConnectionPool**(*size: int*, ***kwargs*)
Asyncio-safe connection pool.

New in version 0.5.

Connection pools in sync code (like `happybase.ConnectionPool`) work by creating multiple connections and providing one whenever a thread asks. When a thread is done with it, it returns it to the pool to be made available to other threads. In async code, instead of threads, tasks make the request to the pool for a connection.

If a task nests calls to *connection()*, it will get the same connection back, just like in HappyBase.

The *size* argument specifies how many connections this pool manages. Additional keyword arguments are passed unmodified to the `happybase.Connection` constructor, with the exception of the *autoconnect* argument, since maintaining connections is the task of the pool.

Parameters

- **size** (*int*) – the maximum number of concurrently open connections
- **kwargs** – keyword arguments for Connection

QUEUE_TYPE
alias of `asyncio.queues.LifoQueue`

close()
Clean up all pool connections and delete the queue.

connection(*timeout: numbers.Real = None*) → aiohappybase.connection.Connection

Obtain a connection from the pool.

This method *must* be used as a context manager, i.e. with Python's `with` block. Example:

```
async with pool.connection() as connection:
    pass  # do something with the connection
```

If *timeout* is specified, this is the number of seconds to wait for a connection to become available before `NoConnectionsAvailable` is raised. If omitted, this method waits forever for a connection to become available.

Parameters **timeout** – number of seconds to wait (optional)

Returns active connection from the pool

class aiohappybase.**NoConnectionsAvailable**

Exception raised when no connections are available.

This happens if a timeout was specified when obtaining a connection, and no connection became available within the specified timeout.

New in version 0.5.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

2.4 Sync API

To maintain complete backwards compatibility with the original HappyBase and to ease upgrading, this library comes with a synchronous version of the API that is autogenerated from the async API at import time to ensure it doesn't diverge.

The library can be accessed one of two ways:

1. Via the `aiohappybase.sync` subpackage
2. The `happybase.py` module (which simply imports everything from 1)

Note: If you have both HappyBase and AIOHappyBase installed in the same environment, HappyBase should be picked when you `import happybase` (packages always seem to be loaded before modules when they have the same name) but it isn't advised to have both installed.

To ensure you always get the sync version of AIOHappyBase, it is best to use `import aiohappybase.sync as happybase` if you wish to use the `happybase` name. The `happybase.py` module is really only to smooth the transition.

In the sync version, all async methods have been converted to synchronous equivalents. Here are some examples from the user guide, which are basically just removals of the async/await keywords:

```
from aiohappybase.sync import Connection

with Connection('somehost') as connection:
    table = connection.create_table('mytable', {
        'cf1': dict(max_versions=10),
        'cf2': dict(max_versions=1, block_cache_enabled=False),
        'cf3': dict(),  # use defaults
```

(continues on next page)

(continued from previous page)

```
    })

    table.put(b'row-key-1', {b'cf:col1': b'value1', b'cf:col2': b'value2'})
    table.put(b'row-key-2', {b'cf:col1': b'value1', b'cf:col2': b'value2'})

    rows = table.rows([b'row-key-1', b'row-key-2'])
    for key, data in rows:
        print(key, data)
```

2.4.1 Connection

class aiohappybase.sync.**Connection**(*host: str = 'localhost'*, *port: int = 9090*, *timeout: int = None*, *autoconnect: bool = True*, *table_prefix: AnyStr = None*, *table_prefix_separator: AnyStr = b'_'*, *compat: str = '0.98'*, *transport: str = 'buffered'*, *protocol: str = 'binary'*, *client: str = 'socket'*, ***client_kwargs*)

Connection to an HBase Thrift server.

The *host* and *port* arguments specify the host name and TCP port of the HBase Thrift server to connect to. If omitted or `None`, a connection to the default port on `localhost` is made. If specifed, the *timeout* argument specifies the socket timeout in milliseconds.

If *autoconnect* is *True* the connection is made directly during initialization. Otherwise a context manager should be used (with Connection...) or *Connection.open()* must be called explicitly before first use. Note that due to limitations in the Python async framework, a RuntimeError will be raised if it is used inside of a running asyncio event loop.

The optional *table_prefix* and *table_prefix_separator* arguments specify a prefix and a separator string to be prepended to all table names, e.g. when *Connection.table()* is invoked. For example, if *table_prefix* is `myproject`, all tables will have names like `myproject_XYZ`.

The optional *compat* argument sets the compatibility level for this connection. Older HBase versions have slightly different Thrift interfaces, and using the wrong protocol can lead to crashes caused by communication errors, so make sure to use the correct one. This value can be either the string `0.90`, `0.92`, `0.94`, or `0.96` (the default).

The optional *transport* argument specifies the Thrift transport mode to use. Supported values for this argument are `buffered` (the default) and `framed`. Make sure to choose the right one, since otherwise you might see non-obvious connection errors or program hangs when making a connection. HBase versions before 0.94 always use the buffered transport. Starting with HBase 0.94, the Thrift server optionally uses a framed transport, depending on the argument passed to the `hbase-daemon.sh start thrift` command. The default `-threadpool` mode uses the buffered transport; the `-hsha`, `-nonblocking`, and `-threadedselector` modes use the framed transport.

The optional *protocol* argument specifies the Thrift transport protocol to use. Supported values for this argument are `binary` (the default) and `compact`. Make sure to choose the right one, since otherwise you might see non-obvious connection errors or program hangs when making a connection. `TCompactProtocol` is a more compact binary format that is typically more efficient to process as well. `TBinaryProtocol` is the default protocol that AIOHappyBase uses.

The optional *client* argument specifies the type of Thrift client to use. Supported values for this argument are `socket` (the default) and `http`. Make sure to choose the right one, since otherwise you might see non-obvious connection errors or program hangs when making a connection. To check which client you should use, refer to the `hbase.regionserver.thrift.http` setting. If it is `true` use `http`, otherwise use `socket`.

New in version v1.4.0: *client* argument

New in version 0.9: *protocol* argument

New in version 0.5: *timeout* argument

New in version 0.4: *table_prefix_separator* argument

New in version 0.4: support for framed Thrift transports

Parameters

- **host** – The host to connect to
- **port** – The port to connect to
- **timeout** – The socket timeout in milliseconds (optional)
- **autoconnect** – Whether the connection should be opened directly
- **table_prefix** – Prefix used to construct table names (optional)
- **table_prefix_separator** – Separator used for *table_prefix*
- **compat** – Compatibility mode (optional)
- **transport** – Thrift transport mode (optional)
- **protocol** – Thrift protocol mode (optional)
- **client** – Thrift client mode (optional)
- **client_kwargs** – Extra keyword arguments for *make_client()*. See the ThriftPy2 documentation for more information.

close() → None

Close the underlying client to the HBase instance. This method can be safely called more than once. Note that the client is destroyed after it is closed which will cause errors to occur if it is used again before reopening. The *Connection* can be reopened by calling *open()* again.

compact_table(*name: AnyStr*, *major: bool = False*) → None

Compact the specified table.

Parameters

- **name** (*str*) – The table name
- **major** (*bool*) – Whether to perform a major compaction.

create_table(*name: AnyStr*, *families: Dict[str, Dict[str, Any]]*) → aiohappybase.sync.table.Table

Create a table.

Parameters

- **name** – The table name
- **families** – The name and options for each column family

Returns The created table instance

The *families* argument is a dictionary mapping column family names to a dictionary containing the options for this column family, e.g.

```
families = {
    'cf1': dict(max_versions=10),
    'cf2': dict(max_versions=1, block_cache_enabled=False),
    'cf3': dict(),  # use defaults
}
connection.create_table('mytable', families)
```

These options correspond to the ColumnDescriptor structure in the Thrift API, but note that the names should be provided in Python style, not in camel case notation, e.g. *time_to_live*, not *timeToLive*. The following options are supported:

- `max_versions` (*int*)
- `compression` (*str*)
- `in_memory` (*bool*)
- `bloom_filter_type` (*str*)
- `bloom_filter_vector_size` (*int*)
- `bloom_filter_nb_hashes` (*int*)
- `block_cache_enabled` (*bool*)
- `time_to_live` (*int*)

delete_table(*name: AnyStr*, *disable: bool = False*) → None
Delete the specified table.

New in version 0.5: *disable* argument

In HBase, a table always needs to be disabled before it can be deleted. If the *disable* argument is *True*, this method first disables the table if it wasn't already and then deletes it.

Parameters

- **name** – The table name
- **disable** – Whether to first disable the table if needed

disable_table(*name: AnyStr*) → None
Disable the specified table.

Parameters **name** – The table name

enable_table(*name: AnyStr*) → None
Enable the specified table.

Parameters **name** – The table name

is_table_enabled(*name: AnyStr*) → None
Return whether the specified table is enabled.

Parameters **name** (*str*) – The table name

Returns whether the table is enabled

Return type bool

open() → None
Create and open the underlying client to the HBase instance. This method can safely be called more than once.

table(*name: AnyStr*, *use_prefix: bool = True*) → aiohappybase.sync.table.Table
Return a table object.

Returns a `happybase.Table` instance for the table named *name*. This does not result in a round-trip to the server, and the table is not checked for existence.

The optional *use_prefix* argument specifies whether the table prefix (if any) is prepended to the specified *name*. Set this to *False* if you want to use a table that resides in another 'prefix namespace', e.g. a table from a 'friendly' application co-hosted on the same HBase instance. See the *table_prefix* argument to the *Connection* constructor for more information.

Parameters

- **name** – the name of the table
- **use_prefix** – whether to use the table prefix (if any)

Returns Table instance

tables() → List[bytes]
Return a list of table names available in this HBase instance.

If a *table_prefix* was set for this *Connection*, only tables that have the specified prefix will be listed.

Returns The table names

2.4.2 Table

class aiohappybase.sync.**Table**(*name: bytes*, *connection: Connection*)
HBase table abstraction class.

This class cannot be instantiated directly; use *Connection.table()* instead.

append(*row: bytes, data: Dict[bytes, bytes], include_timestamp: bool = False*) → Union[Dict[bytes, bytes], Dict[bytes, Tuple[bytes, int]]]
Append data to an existing row.

- This function is only available when using HBase 0.98 (or up).

The *data* argument behaves just like it does in *put()* except that instead of replacing the current values, they are appended to the end. If a specified cell doesn't exist, then the result is the same as calling *put()* for that cell.

Parameters

- **row** – the row key
- **data** – data to append
- **include_timestamp** – include timestamps with the values?

Returns Updated cell values like the output of *row()*

batch(*timestamp: int = None*, *batch_size: int = None*, *transaction: bool = False*, *wal: bool = True*) → aiohappybase.sync.batch.Batch
Create a new batch operation for this table.

This method returns a new *Batch* instance that can be used for mass data manipulation. The *timestamp* argument applies to all puts and deletes on the batch.

If given, the *batch_size* argument specifies the maximum batch size after which the batch should send the mutations to the server. By default this is unbounded.

The *transaction* argument specifies whether the returned *Batch* instance should act in a transaction-like manner when used as context manager in a `with` block of code. The *transaction* flag cannot be used in combination with *batch_size*.

The *wal* argument determines whether mutations should be written to the HBase Write Ahead Log (WAL). This flag can only be used with recent HBase versions. If specified, it provides a default for all the put and delete operations on this batch. This default value can be overridden for individual operations using the *wal* argument to *Batch.put()* and *Batch.delete()*.

New in version 0.7: *wal* argument

Parameters

- **transaction** – whether this batch should behave like a transaction (only useful when used as a context manager)
- **batch_size** – batch size (optional)
- **timestamp** – timestamp (optional)
- **wal** – whether to write to the WAL (optional)

Returns Batch instance

cells(*row: bytes*, *column: bytes*, *versions: int = None*, *timestamp: int = None*, *include_timestamp: bool = False*) → Union[List[bytes], List[Tuple[bytes, int]]]
Retrieve multiple versions of a single cell from the table.

This method retrieves multiple versions of a cell (if any).

The *versions* argument defines how many cell versions to retrieve at most.

The *timestamp* and *include_timestamp* arguments behave exactly the same as for *row()*.

Parameters

- **row** – the row key
- **column** – the column name
- **versions** – the maximum number of versions to retrieve
- **timestamp** – timestamp (optional)
- **include_timestamp** – whether timestamps are returned

Returns cell values

column_family_names() → List[bytes]
Retrieve the column family names for this table

counter_dec(*row: bytes*, *column: bytes*, *value: int = 1*) → int
Atomically decrement (or increments) a counter column.

This method is a shortcut for calling *Table.counter_inc()* with the value negated.

Returns counter value after decrementing

counter_get(*row: bytes*, *column: bytes*) → int
Retrieve the current value of a counter column.

This method retrieves the current value of a counter column. If the counter column does not exist, this function initialises it to *0*.

Note that application code should *never* store a incremented or decremented counter value directly; use the atomic *Table.counter_inc()* and *Table.counter_dec()* methods for that.

Parameters

- **row** – the row key
- **column** – the column name

Returns counter value

counter_inc(*row: bytes*, *column: bytes*, *value: int = 1*) → int
Atomically increment (or decrements) a counter column.

This method atomically increments or decrements a counter column in the row specified by *row*. The *value* argument specifies how much the counter should be incremented (for positive values) or decremented (for

negative values). If the counter column did not exist, it is automatically initialised to 0 before incrementing it.

Parameters

- **row** – the row key
- **column** – the column name
- **value** – the amount to increment or decrement by (optional)

Returns counter value after incrementing

counter_set(*row: bytes*, *column: bytes*, *value: int = 0*) → None
Set a counter column to a specific value.

This method stores a 64-bit signed integer value in the specified column.

Note that application code should *never* store a incremented or decremented counter value directly; use the atomic *Table.counter_inc()* and *Table.counter_dec()* methods for that.

Parameters

- **row** – the row key
- **column** – the column name
- **value** – the counter value to set

delete(*row: bytes*, *columns: Iterable[bytes] = None*, *timestamp: int = None*, *wal: bool = True*) → None
Delete data from the table.

This method deletes all columns for the row specified by *row*, or only some columns if the *columns* argument is specified.

Note that, in many situations, *batch()* is a more appropriate method to manipulate data.

New in version 0.7: *wal* argument

Parameters

- **row** – the row key
- **columns** – list of columns (optional)
- **timestamp** – timestamp (optional)
- **wal** – whether to write to the WAL (optional)

families() → Dict[bytes, Dict[str, Any]]
Retrieve the column families for this table.

Returns Mapping from column family name to settings dict

put(*row: bytes, data: Dict[bytes, bytes], timestamp: int = None, wal: bool = True*) → None
Store data in the table.

This method stores the data in the *data* argument for the row specified by *row*. The *data* argument is dictionary that maps columns to values. Column names must include a family and qualifier part, e.g. `b'cf:col'`, though the qualifier part may be the empty string, e.g. `b'cf:'`.

Note that, in many situations, *batch()* is a more appropriate method to manipulate data.

New in version 0.7: *wal* argument

Parameters

- **row** – the row key

- **data** – the data to store
- **timestamp** – timestamp (optional)
- **wal** – whether to write to the WAL (optional)

regions() → List[Dict[str, Any]]
Retrieve the regions for this table.

Returns regions for this table

row(*row: bytes*, *columns: Iterable[bytes] = None*, *timestamp: int = None*, *include_timestamp: bool = False*) → Union[Dict[bytes, bytes], Dict[bytes, Tuple[bytes, int]]]
Retrieve a single row of data.

This method retrieves the row with the row key specified in the *row* argument and returns the columns and values for this row as a dictionary.

The *row* argument is the row key of the row. If the *columns* argument is specified, only the values for these columns will be returned instead of all available columns. The *columns* argument should be a list or tuple containing byte strings. Each name can be a column family, such as `b'cf1'` or `b'cf1:'` (the trailing colon is not required), or a column family with a qualifier, such as `b'cf1:col1'`.

If specified, the *timestamp* argument specifies the maximum version that results may have. The *include_timestamp* argument specifies whether cells are returned as single values or as *(value, timestamp)* tuples.

Parameters

- **row** – the row key
- **columns** – list of columns (optional)
- **timestamp** – timestamp (optional)
- **include_timestamp** – whether timestamps are returned

Returns Mapping of columns (both qualifier and family) to values

rows(*rows: List[bytes], columns: Iterable[bytes] = None, timestamp: int = None, include_timestamp: bool = False*) → List[Tuple[bytes, Union[Dict[bytes, bytes], Dict[bytes, Tuple[bytes, int]]]]]
Retrieve multiple rows of data.

This method retrieves the rows with the row keys specified in the *rows* argument, which should be should be a list (or tuple) of row keys. The return value is a list of *(row_key, row_dict)* tuples.

The *columns*, *timestamp* and *include_timestamp* arguments behave exactly the same as for *row()*.

Parameters

- **rows** – list of row keys
- **columns** – list of columns (optional)
- **timestamp** – timestamp (optional)
- **include_timestamp** – whether timestamps are returned

Returns List of mappings (columns to values)

scan(*row_start: bytes = None*, *row_stop: bytes = None*, *row_prefix: bytes = None*, *columns: Iterable[bytes] = None*, *filter: bytes = None*, *timestamp: int = None*, *include_timestamp: bool = False*, *batch_size: int = 1000*, *scan_batching: int = None*, *limit: int = None*, *sorted_columns: bool = False*, *reverse: bool = False*) → AsyncGenerator[Tuple[bytes, Dict[bytes, bytes]], None]
Create a scanner for data in the table.

This method returns an iterable that can be used for looping over the matching rows. Scanners can be created in two ways:

- The *row_start* and *row_stop* arguments specify the row keys where the scanner should start and stop. It does not matter whether the table contains any rows with the specified keys: the first row after *row_start* will be the first result, and the last row before *row_stop* will be the last result. Note that the start of the range is inclusive, while the end is exclusive.

  Both *row_start* and *row_stop* can be *None* to specify the start and the end of the table respectively. If both are omitted, a full table scan is done. Note that this usually results in severe performance problems.

- Alternatively, if *row_prefix* is specified, only rows with row keys matching the prefix will be returned. If given, *row_start* and *row_stop* cannot be used.

The *columns*, *timestamp* and *include_timestamp* arguments behave exactly the same as for `row()`.

The *filter* argument may be a filter string that will be applied at the server by the region servers.

If *limit* is given, at most *limit* results will be returned.

The *batch_size* argument specifies how many results should be retrieved per batch when retrieving results from the scanner. Only set this to a low value (or even 1) if your data is large, since a low batch size results in added round-trips to the server.

The optional *scan_batching* is for advanced usage only; it translates to *Scan.setBatching()* at the Java side (inside the Thrift server). By setting this value rows may be split into partial rows, so result rows may be incomplete, and the number of results returned by the scanner may no longer correspond to the number of rows matched by the scan.

If *sorted_columns* is *True*, the columns in the rows returned by this scanner will be retrieved in sorted order, and the data will be stored in *OrderedDict* instances.

If *reverse* is *True*, the scanner will perform the scan in reverse. This means that *row_start* must be lexicographically after *row_stop*. Note that the start of the range is inclusive, while the end is exclusive just as in the forward scan.

Compatibility notes:

- The *filter* argument is only available when using HBase 0.92 (or up). In HBase 0.90 compatibility mode, specifying a *filter* raises an exception.
- The *sorted_columns* argument is only available when using HBase 0.96 (or up).
- The *reverse* argument is only available when using HBase 0.98 (or up).

New in version 1.1.0: *reverse* argument

New in version 0.8: *sorted_columns* argument

New in version 0.8: *scan_batching* argument

Parameters

- **row_start** – the row key to start at (inclusive)
- **row_stop** – the row key to stop at (exclusive)
- **row_prefix** – a prefix of the row key that must match
- **columns** – list of columns (optional)
- **filter** – a filter string (optional)
- **timestamp** – timestamp (optional)

- **include_timestamp** – whether timestamps are returned
- **batch_size** – batch size for retrieving results
- **scan_batching** – server-side scan batching (optional)
- **limit** – max number of rows to return
- **sorted_columns** – whether to return sorted columns
- **reverse** – whether to perform scan in reverse

Returns generator yielding the rows matching the scan

Return type iterable of *(row_key, row_data)* tuples

2.4.3 Batch

class aiohappybase.sync.**Batch**(*table: Table*, *timestamp: int = None*, *batch_size: int = None*, *transaction: bool = False*, *wal: bool = True*)

Batch mutation class.

This class cannot be instantiated directly; use *Table.batch()* instead.

Initialise a new Batch instance.

close() → None
Finalize the batch and make sure all tasks are completed.

counter_dec(*row: bytes*, *column: bytes*, *value: int = 1*) → None
Atomically decrement (or increments) a counter column.

See *Table.counter_dec()* for parameter details. Note that this method cannot return the current value because the change is buffered until send to the server.

counter_inc(*row: bytes*, *column: bytes*, *value: int = 1*) → None
Atomically increment (or decrements) a counter column.

See *Table.counter_inc()* for parameter details. Note that this method cannot return the current value because the change is buffered until send to the server.

delete(*row: bytes*, *columns: Iterable[bytes] = None*, *wal: bool = None*) → None
Delete data from the table.

See *Table.put()* for a description of the *row*, *data*, and *wal* arguments. The *wal* argument should normally not be used; its only use is to override the batch-wide value passed to *Table.batch()*.

put(*row: bytes, data: Dict[bytes, bytes], wal: bool = None*) → None
Store data in the table.

See *Table.put()* for a description of the *row*, *data*, and *wal* arguments. The *wal* argument should normally not be used; its only use is to override the batch-wide value passed to *Table.batch()*.

send() → None
Send the batch to the server.

2.4.4 Connection pool

class aiohappybase.sync.**ConnectionPool**(*size: int*, ***kwargs*)
Thread-safe connection pool.

New in version 0.5.

Connection pools work by creating multiple connections and providing one whenever a thread asks. When a thread is done with it, it returns it to the pool to be made available to other threads.

If a thread nests calls to *connection()*, it will get the same connection back.

The *size* argument specifies how many connections this pool manages. Additional keyword arguments are passed unmodified to the *Connection* constructor, with the exception of the *autoconnect* argument, since maintaining connections is the task of the pool.

Parameters

- **size** (*int*) – the maximum number of concurrently open connections
- **kwargs** – keyword arguments for Connection

QUEUE_TYPE
alias of `queue.LifoQueue`

close()
Clean up all pool connections and delete the queue.

connection(*timeout: numbers.Real = None*) → aiohappybase.sync.connection.Connection
Obtain a connection from the pool.

This method *must* be used as a context manager, i.e. with Python's `with` block. Example:

```
with pool.connection() as connection:
    pass  # do something with the connection
```

If *timeout* is specified, this is the number of seconds to wait for a connection to become available before *NoConnectionsAvailable* is raised. If omitted, this method waits forever for a connection to become available.

Parameters **timeout** – number of seconds to wait (optional)

Returns active connection from the pool

class aiohappybase.sync.**NoConnectionsAvailable**
Exception raised when no connections are available.

This happens if a timeout was specified when obtaining a connection, and no connection became available within the specified timeout.

New in version 0.5.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

CHAPTER 3

Additional documentation

3.1 Version history

3.1.1 AIOHappyBase 1.3.0

Release date: 2020-03-10

- Added support for the async framed transport and compact protocol that were added in ThriftPy2 4.10 (which is now the new minimum version).
- Implemented counter batching with `Batch.counter_inc()` and `Batch.counter_dec()`.
- Added `Table.append()` to utilize the append Thrift endpoint.
- `Connection` now internally utilizes `make_aio_client()` to create the internal Thrift client.
  - All additional keyword arguments provided to `Connection` instances will be passed to the client.
  - This enables support for SSL sockets and any additional features future versions of `thriftpy2` add.
- Implemented a `sync` subpackage to enable backwards compatibility with the original `HappyBase` synchronized API and ease the transition process.
  - It is mostly autogenerated at runtime from the async code to simplify maintenance and ensure all features are available.
  - A simple `happybase.py` module is included to complete the backwards compatibility. If `HappyBase` is already installed, that *should* take precedence.

3.1.2 AIOHappyBase 1.2.0

Release date: 2019-11-28

First release of the async version of HappyBase!

The version number is the same because the API is almost identical (albeit async) except for a few updates:

- Only Python 3.6+ will be supported (I like f-strings and ordered dictionaries, sue me:P)
- `Connection` and `ConnectionPool` objects can be used as context managers (async and regular).
- Explicitly closing non-context managed `Connection` and `ConnectionPool` objects is now required due the asyncio event loop being mostly unavailable during __del__.
- `Connection.create_table()` now returns the Table instance.
- Support for the framed transport and compact protocol have been dropped until `thriftpy2.contrib.aio` supports them as well.

3.1.3 HappyBase 1.2.0

Release date: 2019-05-14

- Switch from `thriftpy` to its successor `thriftpy2`, which supports Python 3.7. (issue #221, pr 222,

3.1.4 HappyBase 1.1.0

Release date: 2017-04-03

- Set socket timeout unconditionally on `TSocket` (#146)
- Add new '0.98' compatibility mode (#155)
- Add support for reversed scanners (#67, #155)

3.1.5 HappyBase 1.0.0

Release date: 2016-08-13

- First 1.x.y release!

  From now on this library uses a semantic versioning scheme. HappyBase is a mature library, but always used 0.x version numbers for no good reason. This has now changed.
- Finally, Python 3 support. Thanks to all the people who contributed! (issue #40, pr 116, pr 108, pr 111)
- Switch to thriftpy as the underlying Thrift library, which is a much nicer and better maintained library.
- Enable building universal wheels (issue 78)

3.1.6 HappyBase 0.9

Release date: 2014-11-24

- Fix an issue where scanners would return fewer results than expected due to HBase not always behaving as its documentation suggests (issue #72).
- Add support for the Thrift compact protocol (`TCompactProtocol`) in *`Connection`* (issue #70).

3.1.7 HappyBase 0.8

Release date: 2014-02-25

- Add (and default to) '0.96' compatibility mode in *Connection*.
- Add support for retrieving sorted columns, which is possible with the HBase 0.96 Thrift API. This feature uses a new *sorted_columns* argument to *Table.scan()*. An `OrderedDict` implementation is required for this feature; with Python 2.7 this is available from the standard library, but for Python 2.6 a separate `ordereddict` package has to be installed from PyPI. (issue #39)
- The *batch_size* argument to *Table.scan()* is no longer propagated to *Scan.setBatching()* at the Java side (inside the Thrift server). To influence the *Scan.setBatching()* (which may split rows into partial rows) a new *scan_batching* argument to *Table.scan()* has been added. See issue #54, issue #56, and the HBase docs for *Scan.setBatching()* for more details.

3.1.8 HappyBase 0.7

Release date: 2013-11-06

- Added a *wal* argument to various data manipulation methods on the *Table* and *Batch* classes to determine whether to write the mutation to the Write-Ahead Log (WAL). (issue #36)
- Pass batch_size to underlying Thrift Scan instance (issue #38).
- Expose server name and port in *Table.regions()* (recent HBase versions only) (issue #37).
- Regenerated bundled Thrift API modules using a recent upstream Thrift API definition. This is required to expose newly added API.

3.1.9 HappyBase 0.6

Release date: 2013-06-12

- Rewrote exception handling in connection pool. Exception handling is now a lot cleaner and does not introduce cyclic references anymore. (issue #25).
- Regenerated bundled Thrift code using Thrift 0.9.0 with the new-style classes flag (issue #27).

3.1.10 HappyBase 0.5

Release date: 2013-05-24

- Added a thread-safe connection pool (*ConnectionPool*) to keep connections open and share them between threads (issue #21).
- The *Connection.delete_table()* method now features an optional *disable* parameter to make deleting enabled tables easier.
- The debug log message emitted by *Table.scan()* when closing a scanner now includes both the number of rows returned to the calling code, and also the number of rows actually fetched from the server. If scanners are not completely iterated over (e.g. because of a 'break' statement in the for loop for the scanner), these numbers may differ. If this happens often, and the differences are big, this may be a hint that the *batch_size* parameter to *Table.scan()* is not optimal for your application.
- Increased Thrift dependency to at least 0.8. Older versions are no longer available from PyPI. HappyBase should not be used with obsoleted Thrift versions.

- The `Connection` constructor now features an optional *timeout* parameter to to specify the timeout to use for the Thrift socket (issue #15)
- The *timestamp* argument to various methods now also accepts *long* values in addition to *int* values. This fixes problems with large timestamp values on 32-bit systems. (issue #23).
- In some corner cases exceptions were raised during interpreter shutdown while closing any remaining open connections. (issue #18)

3.1.11 HappyBase 0.4

Release date: 2012-07-11

- Add an optional *table_prefix_separator* argument to the `Connection` constructor, to specify the prefix used for the *table_prefix* argument (issue #3)
- Add support for framed Thrift transports using a new optional *transport* argument to `Connection` (issue #6)
- Add the Apache license conditions in the *license statement* (for the included HBase parts)
- Documentation improvements

3.1.12 HappyBase 0.3

Release date: 2012-05-25

New features:

- Improved compatibility with HBase 0.90.x
  - In earlier versions, using `Table.scan()` in combination with HBase 0.90.x often resulted in crashes, caused by incompatibilities in the underlying Thrift protocol.
  - A new *compat* flag to the `Connection` constructor has been added to enable compatibility with HBase 0.90.x.
  - Note that the `Table.scan()` API has a few limitations when used with HBase 0.90.x.
- The *row_prefix* argument to `Table.scan()` can now be used together with *filter* and *timestamp* arguments.

Other changes:

- Lower Thrift dependency to 0.6
- The *setup.py* script no longer installs the tests
- Documentation improvements

3.1.13 HappyBase 0.2

Release date: 2012-05-22

- Fix package installation, so that `pip install happybase` works as expected (issue #1)
- Various small documentation improvements

3.1.14 HappyBase 0.1

Release date: 2012-05-20

- Initial release

3.2 Development

3.2.1 Getting the source

The AIOHappyBase source code repository is hosted on GitHub:

https://github.com/python-happybase/aiohappybase

To grab a copy, use this:

```
$ git clone https://github.com/python-happybase/aiohappybase.git
```

3.2.2 Setting up a development environment

Setting up a development environment from a Git branch is easy:

```
$ cd /path/to/aiohappybase/
$ python -m venv venv
$ source venv/bin/activate
(venv) $ pip install -r test-requirements.txt
(venv) $ pip install -e .
```

3.2.3 Running the tests

The tests use the *asynctest* test suite. To execute the tests, run:

```
(venv) $ make test
```

Test outputs are shown on the console. A test code coverage report is saved in *coverage/index.html*.

If the Thrift server is not running on localhost, you can specify these environment variables (both are optional) before running the tests:

```
(venv) $ export AIOHAPPYBASE_HOST=host.example.org
(venv) $ export AIOHAPPYBASE_PORT=9091
```

To test the HBase 0.90 compatibility mode, use this:

```
(venv) $ export AIOHAPPYBASE_COMPAT=0.90
```

To test the framed Thrift transport mode, use this:

```
(venv) $ export AIOHAPPYBASE_TRANSPORT=framed
```

3.2.4 Contributing

Feel free to report any issues on GitHub. Patches and merge requests are also most welcome.

3.3 To-do list and possible future work

This document lists some ideas that the developers thought of, but have not yet implemented. The topics described below may be implemented (or not) in the future, depending on time, demand, and technical possibilities.

- Improved error handling instead of just propagating the errors from the Thrift layer. Maybe wrap the errors in a HappyBase.Error?
- Automatic retries for failed operations (but only those that can be retried)
- Port HappyBase over to the (still experimental) HBase Thrift2 API when it becomes mainstream, and expose more of the underlying features nicely in the HappyBase API.

3.4 Frequently asked questions

3.4.1 I love AIOHappyBase! Can I donate?

While I am not accepting donations at this time, the original author is:

From the original HappyBase author, Wouter Bolsterlee:

Thanks, I'm glad to hear that you appreciate my work! If you feel like, please make a small donation to sponsor my (spare time!) work on HappyBase. Small gestures are really motivating for me and help me keep this project going!

3.4.2 Why not use the Thrift API directly?

While the HBase Thrift API can be used directly from Python using (automatically generated) HBase Thrift service classes, application code doing so is very verbose, cumbersome to write, and hence error-prone. The reason for this is that the HBase Thrift API is a flat, language-agnostic interface API closely tied to the RPC going over the wire-level protocol. In practice, this means that applications using Thrift directly need to deal with many imports, sockets, transports, protocols, clients, Thrift types and mutation objects. For instance, look at the code required to connect to HBase and store two values:

```
import asyncio as aio

from thriftpy2.contrib.aio.client import TAsyncClient
from thriftpy2.contrib.aio.socket import TAsyncSocket
from thriftpy2.contrib.aio.transport.buffered import TAsyncBufferedTransport
from thriftpy2.contrib.aio.protocol.binary import TAsyncBinaryProtocol

from hbase import Hbase, Mutation

async def main():

    sock = TAsyncSocket('hostname', 9090)
    transport = TAsyncBufferedTransport(sock)
    protocol = TAsyncBinaryProtocol(transport)
    client = TAsyncClient(Hbase, protocol)
    transport.open()

    mutations = [
        Mutation(column=b'family:qual1', value=b'value1'),
        Mutation(column=b'family:qual2', value=b'value2'),
    ]
```

(continues on next page)

(continued from previous page)

```
    await client.mutateRow(b'table-name', b'row-key', mutations)

aio.run(main())
```

PEP 20 taught us that simple is better than complex, and as you can see, Thrift is certainly complex. AIOHappyBase hides all the Thrift cruft below a friendly API. The resulting application code will be cleaner, more productive to write, and more maintainable. With AIOHappyBase, the example above can be simplified to this:

```
import asyncio as aio

from aiohappybase import Connection

async def main():
    async with Connection('hostname') as conn:
        table = conn.table(b'table-name')
        await table.put(b'row-key', {
             b'family:qual1': b'value1',
             b'family:qual2': b'value2',
        })

aio.run(main())
```

If you're not convinced and still think the Thrift API is not that bad, please try to accomplish some other common tasks, e.g. retrieving rows and scanning over a part of a table, and compare that to the AIOHappyBase equivalents. If you're still not convinced by then, we're sorry to inform you that AIOHappyBase is not the project for you, and we wish you all of luck maintaining your code – or is it just Thrift boilerplate?

3.5 License

AIOHappyBase itself is licensed under a MIT License. AIOHappyBase contains code originating from HBase sources, licensed under the Apache License (version 2.0). Both license texts are included below.

3.5.1 AIOHappyBase License

(This is the MIT License.)

3.5.2 HBase License

(This is the Apache License, version 2.0, January 2004.)

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

   "Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

   "Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

   "Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a

lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

   (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

   You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to

indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

CHAPTER 4

External links

- Online Documentation (Read the Docs)
- Downloads (PyPI)
- Source Code (Github)
- HappyBase Online Documentation (Read the Docs)
- HappyBase Downloads (PyPI)
- HappyBase Source Code (Github)

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Index

F

I

N

O

P

Q

R

S

T

W