pycassa Documentation

Release 1.8.0

Jonathan Hseu

April 25, 2013

CONTENTS

pycassa is a Python client for Apache Cassandra. The latest release of pycassa is fully compatible with Cassandra 0.8 to 1.1, and is compatible with the data API of Cassandra 0.7.

pycassa is open source under the MIT license. The source code repository for pycassa can be found on Github.

CHAPTER

ONE

CONTENTS

CHAPTER

TWO

HELP

Mailing Lists

- User list: mail to pycassa-discuss@googlegroups.com or view online.
- Developer list: mail to pycassa-devel@googlegroups.com or view online.

IRC

- Use #cassandra on irc.freenode.net. If you don't have an IRC client, you can use freenode's web based client.

CHAPTER

THREE

ISSUES

Bugs and feature requests for pycassa are currently tracked through the github issue tracker.

CHAPTER

FOUR

CONTRIBUTING

You are encouraged to offer any contributions or ideas you have. Contributing to the documentation or examples, reporting bugs, requesting features, and (of course) improving the code are all equally welcome. To contribute, fork the project on github and make a pull request.

CHAPTER

FIVE

ABOUT THIS DOCUMENTATION

This documentation is generated using the Sphinx documentation generator. The source files for the documentation are located in the *doc/* directory of pycassa. To generate the documentation, run the following command from the root directory of pycassa:

```
$ python setup.py doc
```

5.1 Installing

5.1.1 Requirements

You need to have either Python 2.6 or 2.7 installed.

5.1.2 Installing from PyPi

If you have `pip` installed, you can simply do:

```
$ pip install pycassa
```

This will also install the Thrift python bindings automatically.

5.1.3 Manual Installation

Make sure that you have Thrift's python bindings installed:

```
$ pip install thrift
```

You can download a release from github or check out the latest source from github:

```
$ git clone git://github.com/pycassa/pycassa.git
```

You can simply copy the pycassa directory into your project, or you can install pycassa system-wide:

```
$ cd pycassa/
$ sudo python setup.py install
```

5.2 Tutorial

This tutorial is intended as an introduction to working with Cassandra and **pycassa**.

5.2.1 Prerequisites

Before we start, make sure that you have **pycassa** *installed*. In the Python shell, the following should run without raising an exception:

```
>>> import pycassa
```

This tutorial also assumes that a Cassandra instance is running on the default host and port. Read the instructions for getting started with Cassandra if you need help with this.

You can start Cassandra like so:

```
$ pwd
~/cassandra
$ bin/cassandra -f
```

5.2.2 Creating a Keyspace and Column Families

We need to create a keyspace and some column families to work with. There are two good ways to do this: using cassandra-cli, or using pycassaShell. Both are documented below.

Using cassandra-cli

The cassandra-cli utility is included with Cassandra. It allows you to create and modify the schema, explore or modify data, and examine a few things about your cluster. Here's how to create the keyspace and column family we need for this tutorial:

```
user@~ $ cassandra-cli
Welcome to cassandra CLI.

Type 'help;' or '?' for help. Type 'quit;' or 'exit;' to quit.
[default@unknown] connect localhost/9160;
Connected to: "Test Cluster" on localhost/9160
[default@unknown] create keyspace Keyspace1;
4f9e42c4-645e-11e0-ad9e-e700f669bcfc
Waiting for schema agreement...
... schemas agree across the cluster
[default@unknown] use Keyspace1;
Authenticated to keyspace: Keyspace1
[default@Keyspace1] create column family ColumnFamily1;
632cf985-645e-11e0-ad9e-e700f669bcfc
Waiting for schema agreement...
... schemas agree across the cluster
[default@Keyspace1] quit;
user@~ $
```

This connects to a local instance of Cassandra and creates a keyspace named 'Keyspace1' with a column family named 'ColumnFamily1'.

You can find further documentation for the CLI online.

Using pycassaShell

pycassaShell is an interactive Python shell that is included with **pycassa**. Upon starting, it sets up many of the objects that you typically work with when using **pycassa**. It provides most of the functionality that cassandra-cli does, but also gives you a full Python environment to work with.

Here's how to create the keyspace and column family:

```
user@~ $ pycassaShell
---------------------------------
Cassandra Interactive Python Shell
---------------------------------
Keyspace: None
Host: localhost:9160

ColumnFamily instances are only available if a keyspace is specified with -k/--keyspace

Schema definition tools and cluster information are available through SYSTEM_MANAGER.

>>> SYSTEM_MANAGER.create_keyspace('Keyspace1', strategy_options={"replication_factor": "1"})
>>> SYSTEM_MANAGER.create_column_family('Keyspace1', 'ColumnFamily1')
```

5.2.3 Connecting to Cassandra

The first step when working with **pycassa** is to connect to the running cassandra instance:

```
>>> from pycassa.pool import ConnectionPool
>>> pool = ConnectionPool('Keyspace1')
```

The above code will connect by default to `localhost:9160`. We can also specify the host (or hosts) and port explicitly as follows:

```
>>> pool = ConnectionPool('Keyspace1', ['localhost:9160'])
```

This creates a small connection pool for use with a `ColumnFamily` . See Connection Pooling for more details.

5.2.4 Getting a ColumnFamily

A column family is a collection of rows and columns in Cassandra, and can be thought of as roughly the equivalent of a table in a relational database. We'll use one of the column families that are included in the default schema file:

```
>>> from pycassa.pool import ConnectionPool
>>> from pycassa.columnfamily import ColumnFamily
>>>
>>> pool = ConnectionPool('Keyspace1')
>>> col_fam = ColumnFamily(pool, 'ColumnFamily1')
```

If you get an error about the keyspace or column family not existing, make sure you created the keyspace and column family as shown above.

5.2.5 Inserting Data

To insert a row into a column family we can use the `insert()` method:

```
>>> col_fam.insert('row_key', {'col_name': 'col_val'})
1354459123410932
```

We can also insert more than one column at a time:

```
>>> col_fam.insert('row_key', {'col_name':'col_val', 'col_name2':'col_val2'})
1354459123410932
```

And we can insert more than one row at a time:

```
>>> col_fam.batch_insert({'row1': {'name1': 'val1', 'name2': 'val2'},
...                       'row2': {'foo': 'bar'}})
1354491238721387
```

5.2.6 Getting Data

There are many more ways to get data out of Cassandra than there are to insert data.

The simplest way to get data is to use `get()`:

```
>>> col_fam.get('row_key')
{'col_name': 'col_val', 'col_name2': 'col_val2'}
```

Without any other arguments, `get()` returns every column in the row (up to *column_count*, which defaults to 100). If you only want a few of the columns and you know them by name, you can specify them using a *columns* argument:

```
>>> col_fam.get('row_key', columns=['col_name', 'col_name2'])
{'col_name': 'col_val', 'col_name2': 'col_val2'}
```

We may also get a slice (or subrange) of the columns in a row. To do this, use the *column_start* and *column_finish* parameters. One or both of these may be left empty to allow the slice to extend to one or both ends. Note that *column_finish* is inclusive.

```
>>> for i in range(1, 10):
...     col_fam.insert('row_key', {str(i): 'val'})
...
1302542571215334
1302542571218485
1302542571220599
1302542571221991
1302542571223388
1302542571224629
1302542571225859
1302542571227029
1302542571228472
>>> col_fam.get('row_key', column_start='5', column_finish='7')
{'5': 'val', '6': 'val', '7': 'val'}
```

Sometimes you want to get columns in reverse sorted order. A common example of this is getting the last N columns from a row that represents a timeline. To do this, set *column_reversed* to `True`. If you think of the columns as being sorted from left to right, when *column_reversed* is `True`, *column_start* will determine the right end of the range while *column_finish* will determine the left.

Here's an example of getting the last three columns in a row:

```
>>> col_fam.get('row_key', column_reversed=True, column_count=3)
{'9': 'val', '8': 'val', '7': 'val'}
```

There are a few ways to get multiple rows at the same time. The first is to specify them by name using `multiget()`:

```
>>> col_fam.multiget(['row1', 'row2'])
{'row1': {'name1': 'val1', 'name2': 'val2'}, 'row_key2': {'foo': 'bar'}}
```

Another way is to get a range of keys at once by using `get_range()`. The parameter *finish* is also inclusive here, too. Assuming we've inserted some rows with keys 'row_key1' through 'row_key9', we can do this:

```
>>> result = col_fam.get_range(start='row_key5', finish='row_key7')
>>> for key, columns in result:
...     print key, '=>', columns
...
'row_key5' => {'name':'val'}
'row_key6' => {'name':'val'}
'row_key7' => {'name':'val'}
```

Note: Cassandra must be using an OrderPreservingPartitioner for you to be able to get a meaningful range of rows; the default, RandomPartitioner, stores rows in the order of the MD5 hash of their keys. See http://www.datastax.com/docs/1.1/cluster_architecture/partitioning.

The last way to get multiple rows at a time is to take advantage of secondary indexes by using `get_indexed_slices()`, which is described in the *Secondary Indexes* section.

It's also possible to specify a set of columns or a slice for `multiget()` and `get_range()` just like we did for `get()`.

5.2.7 Counting

If you just want to know how many columns are in a row, you can use `get_count()`:

```
>>> col_fam.get_count('row_key')
3
```

If you only want to get a count of the number of columns that are inside of a slice or have particular names, you can do that as well:

```
>>> col_fam.get_count('row_key', columns=['foo', 'bar'])
2
>>> col_fam.get_count('row_key', column_start='foo')
3
```

You can also do this in parallel for multiple rows using `multiget_count()`:

```
>>> col_fam.multiget_count(['fib0', 'fib1', 'fib2', 'fib3', 'fib4'])
{'fib0': 1, 'fib1': 1, 'fib2': 2, 'fib3': 3, 'fib4': 5'}

>>> col_fam.multiget_count(['fib0', 'fib1', 'fib2', 'fib3', 'fib4'],
...                        columns=['col1', 'col2', 'col3'])
{'fib0': 1, 'fib1': 1, 'fib2': 2, 'fib3': 3, 'fib4': 3'}

>>> col_fam.multiget_count(['fib0', 'fib1', 'fib2', 'fib3', 'fib4'],
...                        column_start='col1', column_finish='col3')
{'fib0': 1, 'fib1': 1, 'fib2': 2, 'fib3': 3, 'fib4': 3'}
```

5.2.8 Typed Column Names and Values

Within a column family, column names have a specified *comparator type* which controls how they are sorted. Column values and row keys may also have a *validation class*, which validates that inserted values are the correct type.

The different types available include ASCII strings, integers, dates, UTF8, raw bytes, UUIDs, and more. See `pycassa.types` for a full list.

Cassandra requires you to pack column names and values into a format it can understand by using something like `struct.pack()`. Fortunately, when **pycassa** sees that a column family has a particular comparator type or validation class, it knows to pack and unpack these data types automatically for you. So, if we want to write to the StandardInt column family, which has an IntegerType comparator, we can do the following:

```
>>> col_fam = pycassa.ColumnFamily(pool, 'StandardInt')
>>> col_fam.insert('row_key', {42: 'some_val'})
1354491238721387
>>> col_fam.get('row_key')
{42: 'some_val'}
```

Notice that 42 is an integer here, not a string.

As mentioned above, Cassandra also offers validators on column values and keys with the same set of types. Column value validators can be set for an entire column family, for individual columns, or both. **pycassa** knows to pack these column values automatically too. Suppose we have a *Users* column family with two columns, `name` and `age`, with types UTF8Type and IntegerType:

```
>>> col_fam = pycassa.ColumnFamily(pool, 'Users')
>>> col_fam.insert('thobbs', {'name': 'Tyler', 'age': 24})
1354491238782746
>>> col_fam.get('thobbs')
{'name': 'Tyler', 'age': 24}
```

Of course, if **pycassa**'s automatic behavior isn't working for you, you can turn it off or change it using `autopack_names`, `autopack_values`, `column_name_class`, `default_validation_class`, and so on.

5.2.9 Connection Pooling

Pycassa uses connection pools to maintain connections to Cassandra servers. The `ConnectionPool` class is used to create the connection pool. After creating the pool, it may be used to create multiple `ColumnFamily` objects.

```
>>> pool = pycassa.ConnectionPool('Keyspace1', pool_size=20)
>>> standard_cf = pycassa.ColumnFamily(pool, 'Standard1')
>>> standard_cf.insert('key', {'col': 'val'})
1354491238782746
>>> super_cf = pycassa.ColumnFamily(pool, 'Super1')
>>> super_cf.insert('key2', {'column' : {'col': 'val'}})
1354491239779182
>>> standard_cf.get('key')
{'col': 'val'}
>>> pool.dispose()
```

Automatic retries (or "failover") happen by default with ConectionPools. This means that if any operation fails, it will be transparently retried on other servers until it succeeds or a maximum number of failures is reached.

5.3 Twissandra Example

This example shows you how to work with Twissandra, a Twitter-like example Cassandra application.

5.3.1 Setup

To be completed...

5.4 API Documentation

5.4.1 Pycassa Modules

pycassa - Exceptions and Enums

exception pycassa.**AuthenticationException**
The credentials supplied when creating a connection did not validate, indicating a bad username or password.

exception pycassa.**AuthorizationException**
The user that is currently logged in for a connection was not permitted to perform an action.

exception pycassa.**InvalidRequestException**
Something about the request made was invalid or malformed. The request should not be repeated without modification. Sometimes checking the server logs may help debug what was wrong with the request.

exception pycassa.**NotFoundException**
The row requested does not exist, or the slice requested was empty.

exception pycassa.**UnavailableException**
Not enough replicas are up to satisfy the requested consistency level.

exception pycassa.**TimedOutException**
The replica node did not respond to the coordinator node within `rpc_timeout_in_ms` (as configured in `cassandra.yaml`), typically indicating that the replica is overloaded or just went down.

class pycassa.**ConsistencyLevel**

ANY
Only requires that one replica receives the write *or* the coordinator stores a hint to replay later. Valid only for writes.

ONE
Only one replica needs to respond to consider the operation a success

QUORUM
ceil(RF/2) replicas must respond to consider the operation a success

ALL
All replicas must respond to consider the operation a success

LOCAL_QUORUM
Requres a quorum of replicas in the local datacenter

EACH_QUORUM
Requres a quorum of replicas in each datacenter

TWO
Two replicas must respond to consider the operation a success

THREE
Three replicas must respond to consider the operation a success

pycassa.pool – Connection Pooling

Connection pooling for Cassandra connections.

class pycassa.pool.**ConnectionPool**(*keyspace, server_list=['localhost:9160'], credentials=None, timeout=0.5, use_threadlocal=True, pool_size=5, prefill=True, socket_factory=<function default_socket_factory at 0x18d81b8>, transport_factory=<function default_transport_factory at 0x18fa2a8>, **kwargs*)

A pool that maintains a queue of open connections.

All connections in the pool will be opened to *keyspace*.

server_list is a sequence of servers in the form `"host:port"` that the pool will connect to. The port defaults to 9160 if excluded. The list will be randomly shuffled before being drawn from sequentially. *server_list* may also be a function that returns the sequence of servers.

If authentication or authorization is required, *credentials* must be supplied. This should be a dictionary containing 'username' and 'password' keys with appropriate string values.

timeout specifies in seconds how long individual connections will block before timing out. If set to `None`, connections will never timeout.

If *use_threadlocal* is set to `True`, repeated calls to `get()` within the same application thread will return the same `ConnectionWrapper` object if one is already checked out from the pool. Be careful when setting *use_threadlocal* to `False` in a multithreaded application, especially with retries enabled. Synchronization may be required to prevent the connection from changing while another thread is using it.

The pool will keep up *pool_size* open connections in the pool at any time. When a connection is returned to the pool, the connection will be discarded is the pool already contains *pool_size* connections. The total number of simultaneous connections the pool will allow is `pool_size + max_overflow`, and the number of "sleeping" connections the pool will allow is `pool_size`.

A good choice for *pool_size* is a multiple of the number of servers passed to the Pool constructor. If a size less than this is chosen, the last `(len(server_list) - pool_size)` servers may not be used until either overflow occurs, a connection is recycled, or a connection fails. Similarly, if a multiple of `len(server_list)` is not chosen, those same servers would have a decreased load. By default, overflow is disabled.

If *prefill* is set to `True`, *pool_size* connections will be opened when the pool is created.

Example Usage:

```
>>> pool = pycassa.ConnectionPool(keyspace='Keyspace1', server_list=['10.0.0.4:9160', '10.0.0.5:
>>> cf = pycassa.ColumnFamily(pool, 'Standard1')
>>> cf.insert('key', {'col': 'val'})
1287785685530679
```

max_overflow
: Whether or not a new connection may be opened when the pool is empty is controlled by *max_overflow*. This specifies how many additional connections may be opened after the pool has reached *pool_size*; keep in mind that these extra connections will be discarded upon checkin until the pool is below *pool_size*. This may be set to -1 to indicate no overflow limit. The default value is 0, which does not allow for overflow.

pool_timeout = 30
: If `pool_size + max_overflow` connections have already been checked out, an attempt to retrieve a new connection from the pool will wait up to *pool_timeout* seconds for a connection to be returned to the pool before giving up. Note that this setting is only meaningful when you are accessing the pool concurrently, such as with multiple threads. This may be set to 0 to fail immediately or -1 to wait forever. The default value is 30.

recycle = 10000
After performing *recycle* number of operations, connections will be replaced when checked back in to the pool. This may be set to -1 to disable connection recycling. The default value is 10,000.

max_retries = 5
When an operation on a connection fails due to an `TimedOutException` or `UnavailableException`, which tend to indicate single or multiple node failure, the operation will be retried on different nodes up to *max_retries* times before an `MaximumRetryException` is raised. Setting this to 0 disables retries and setting to -1 allows unlimited retries. The default value is 5.

logging_name = None
By default, each pool identifies itself in the logs using `id(self)`. If multiple pools are in use for different purposes, setting *logging_name* will help individual pools to be identified in the logs.

get()
Gets a connection from the pool.

put(*conn*)
Returns a connection to the pool.

execute(*f*, **args*, ***kwargs*)
Get a connection from the pool, execute *f* on it with **args* and ***kwargs*, return the connection to the pool, and return the result of *f*.

fill()
Adds connections to the pool until at least `pool_size` connections exist, whether they are currently checked out from the pool or not. New in version 1.2.0.

dispose()
Closes all checked in connections in the pool.

set_server_list(*server_list*)
Sets the server list that the pool will make connections to.

server_list should be sequence of servers in the form `"host:port"` that the pool will connect to. The list will be randomly permuted before being used. *server_list* may also be a function that returns the sequence of servers.

size()
Returns the capacity of the pool.

overflow()
Returns the number of overflow connections that are currently open.

checkedin()
Returns the number of connections currently in the pool.

checkedout()
Returns the number of connections currently checked out from the pool.

add_listener(*listener*)
Add a `PoolListener`-like object to this pool.

listener may be an object that implements some or all of `PoolListener`, or a dictionary of callables containing implementations of some or all of the named methods in `PoolListener`.

exception `pycassa.pool.`**AllServersUnavailable**
Raised when none of the servers given to a pool can be connected to.

exception `pycassa.pool.`**NoConnectionAvailable**
Raised when there are no connections left in a pool.

exception pycassa.pool.**MaximumRetryException**
Raised when a ConnectionWrapper has retried the maximum allowed times before being returned to the pool; note that all of the retries do not have to be on the same operation.

exception pycassa.pool.**InvalidRequestError**
Pycassa was asked to do something it can't do.

This error generally corresponds to runtime state errors.

class pycassa.pool.**ConnectionWrapper**(*pool*, *max_retries*, **args*, ***kwargs*)
Creates a wrapper for a Connection object, adding pooling related functionality while still allowing access to the thrift API calls.

These should not be created directly, only obtained through Pool's get() method.

get_keyspace_description(*keyspace=None*, *use_dict_for_col_metadata=False*)
Describes the given keyspace.

If *use_dict_for_col_metadata* is True, the column metadata will be stored as a dictionary instead of a list

A dictionary of the form {column_family_name: CfDef} is returned.

return_to_pool()
Returns this to the pool.

This has the same effect as calling ConnectionPool.put() on the wrapper.

class pycassa.pool.**PoolListener**
Hooks into the lifecycle of connections in a ConnectionPool.

Usage:

```
class MyListener(PoolListener):
    def connection_created(self, dic):
        '''perform connect operations'''
    # etc.

# create a new pool with a listener
p = ConnectionPool(..., listeners=[MyListener()])

# or add a listener after the fact
p.add_listener(MyListener())
```

Listeners receive a dictionary that contains event information and is indexed by a string describing that piece of info. For example, all event dictionaries include 'level', so dic['level'] will return the prescribed logging level.

There is no need to subclass PoolListener to handle events. Any class that implements one or more of these methods can be used as a pool listener. The ConnectionPool will inspect the methods provided by a listener object and add the listener to one or more internal event queues based on its capabilities. In terms of efficiency and function call overhead, you're much better off only providing implementations for the hooks you'll be using.

Each of the PoolListener methods wil be called with a dict as the single parameter. This dict may contain the following fields:

- *connection*: The ConnectionWrapper object that persistently manages the connection
- *message*: The reason this event happened
- *error*: The Exception that caused this event
- *pool_id*: The id of the ConnectionPool that this event came from
- *level*: The prescribed logging level for this event. Can be 'debug', 'info', 'warn', 'error', or 'critical'

Entries in the `dict` that are specific to only one event type are detailed with each method.

connection_checked_in(*dic*)
Called when a connection returns to the pool.

Fields: *pool_id*, *level*, and *connection*.

connection_checked_out(*dic*)
Called when a connection is retrieved from the Pool.

Fields: *pool_id*, *level*, and *connection*.

connection_created(*dic*)
Called once for each new Cassandra connection.

Fields: *pool_id*, *level*, and *connection*.

connection_disposed(*dic*)
Called when a connection is closed.

`dic['message']`: A reason for closing the connection, if any.

Fields: *pool_id*, *level*, *connection*, and *message*.

connection_failed(*dic*)
Called when a connection to a single server fails.

`dic['server']`: The server the connection was made to.

Fields: *pool_id*, *level*, *error*, *server*, and *connection*.

connection_recycled(*dic*)
Called when a connection is recycled.

`dic['old_conn']`: The `ConnectionWrapper` that is being recycled

`dic['new_conn']`: The `ConnectionWrapper` that is replacing it

Fields: *pool_id*, *level*, *old_conn*, and *new_conn*.

pool_at_max(*dic*)
Called when an attempt is made to get a new connection from the pool, but the pool is already at its max size.

`dic['pool_max']`: The max number of connections the pool will keep open at one time.

Fields: *pool_id*, *pool_max*, and *level*.

pool_disposed(*dic*)
Called when a pool is disposed.

Fields: *pool_id*, and *level*.

server_list_obtained(*dic*)
Called when the pool finalizes its server list.

`dic['server_list']`: The randomly permuted list of servers that the pool will choose from.

Fields: *pool_id*, *level*, and *server_list*.

pycassa.columnfamily – Column Family

Provides an abstraction of Cassandra's data model to allow for easy manipulation of data inside Cassandra.

See Also:

pycassa.columnfamilymap

static columnfamily.**gm_timestamp**()
Gets the current GMT timestamp in microseconds.

class pycassa.columnfamily.**ColumnFamily**(*pool*, *column_family*)
An abstraction of a Cassandra column family or super column family. Operations on this, such as get() or insert() will get data from or insert data into the corresponding Cassandra column family.

pool is a ConnectionPool that the column family will use for all operations. A connection is drawn from the pool before each operations and is returned afterwards.

column_family should be the name of the column family that you want to use in Cassandra. Note that the keyspace to be used is determined by the pool.

read_consistency_level = 1
The default consistency level for every read operation, such as get() or get_range(). This may be overridden per-operation. This should be an instance of ConsistencyLevel. The default level is ONE.

write_consistency_level = 1
The default consistency level for every write operation, such as insert() or remove(). This may be overridden per-operation. This should be an instance of ConsistencyLevel. The default level is ONE.

autopack_names = True
Controls whether column names are automatically converted to or from their natural type to the binary string format that Cassandra uses. The data type used is controlled by column_name_class for column names and super_column_name_class for super column names. By default, this is True.

autopack_values = True
Whether column values are automatically converted to or from their natural type to the binary string format that Cassandra uses. The data type used is controlled by default_validation_class and column_validators. By default, this is True.

autopack_keys = True
Whether row keys are automatically converted to or from their natural type to the binary string format that Cassandra uses. The data type used is controlled by key_validation_class. By default, this is True.

column_name_class
The data type of column names, which pycassa will use to determine how to pack and unpack them.

This is set automatically by inspecting the column family's comparator_type, but it may also be set manually if you want autopacking behavior without setting a comparator_type. Options include an instance of any class in pycassa.types, such as LongType().

super_column_name_class
Like column_name_class, but for super column names.

default_validation_class
The default data type of column values, which pycassa will use to determine how to pack and unpack them.

This is set automatically by inspecting the column family's default_validation_class, but it may also be set manually if you want autopacking behavior without setting a default_validation_class. Options include an instance of any class in pycassa.types, such as LongType().

column_validators
Like default_validation_class, but is a dict mapping individual columns to types.

key_validation_class
The data type of row keys, which pycassa will use to determine how to pack and unpack them.

This is set automatically by inspecting the column family's `key_validation_class` (which only exists in Cassandra 0.8 or greater), but may be set manually if you want the autopacking behavior without setting a `key_validation_class` or if you are using Cassandra 0.7. Options include an instance of any class in `pycassa.types`, such as `LongType()`.

dict_class = <class 'collections.OrderedDict'>
Results are returned as dictionaries. By default, python 2.7's `collections.OrderedDict` is used if available, otherwise `OrderedDict` is used so that order is maintained. A different class, such as `dict`, may be instead by used setting this.

buffer_size = 1024
When calling `get_range()` or `get_indexed_slices()`, the intermediate results need to be buffered if we are fetching many rows, otherwise performance may suffer and the Cassandra server may overallocate memory and fail. This is the size of that buffer in number of rows. The default is 1024.

column_buffer_size = 1024
The number of columns fetched at once for `xget()`

timestamp = <unbound method ColumnFamily.gm_timestamp>
Each `insert()` or `remove()` sends a timestamp with every column. This attribute is a function that is used to get this timestamp when needed. The default function is `gm_timestamp()`.

load_schema()
Loads the schema definition for this column family from Cassandra and updates comparator and validation classes if neccessary.

get(*key[, columns][, column_start][, column_finish][, column_reversed][, column_count][, include_timestamp][, super_column][, read_consistency_level]*)
Fetches all or part of the row with key *key*.

The columns fetched may be limited to a specified list of column names using *columns*.

Alternatively, you may fetch a slice of columns or super columns from a row using *column_start*, *column_finish*, and *column_count*. Setting these will cause columns or super columns to be fetched starting with *column_start*, continuing until *column_count* columns or super columns have been fetched or *column_finish* is reached. If *column_start* is left as the empty string, the slice will begin with the start of the row; leaving *column_finish* blank will cause the slice to extend to the end of the row. Note that *column_count* defaults to 100, so rows over this size will not be completely fetched by default.

If *column_reversed* is `True`, columns are fetched in reverse sorted order, beginning with *column_start*. In this case, if *column_start* is the empty string, the slice will begin with the end of the row.

You may fetch all or part of only a single super column by setting *super_column*. If this is set, *column_start*, *column_finish*, *column_count*, and *column_reversed* will apply to the subcolumns of *super_column*.

To include every column's timestamp in the result set, set *include_timestamp* to `True`. Results will include a `(value, timestamp)` tuple for each column.

If this is a standard column family, the return type is of the form `{column_name: column_value}`. If this is a super column family and *super_column* is not specified, the results are of the form `{super_column_name: {column_name, column_value}}`. If *super_column* is set, the super column name will be excluded and the results are of the form `{column_name: column_value}`.

multiget(*keys[, columns][, column_start][, column_finish][, column_reversed][, column_count][, include_timestamp][, super_column][, read_consistency_level][, buffer_size]*)
Fetch multiple rows from a Cassandra server.

keys should be a list of keys to fetch.

buffer_size is the number of rows from the total list to fetch at a time. If left as `None`, the ColumnFamily's `buffer_size` will be used.

All other parameters are the same as `get()`, except that a list of keys may be passed in.

Results will be returned in the form: `{key: {column_name: column_value}}`. If an OrderedDict is used, the rows will have the same order as *keys*.

xget(*key[, column_start][, column_finish][, column_reversed][, column_count][, include_timestamp][, read_consistency_level][, buffer_size]*)
Like `get()`, but creates a generator that pages over the columns automatically.

The number of columns fetched at once can be controlled with the *buffer_size* parameter. The default is `column_buffer_size`.

The generator returns *(name, value)* tuples.

get_count(*key[, super_column][, columns][, column_start][, column_finish][, super_column][, read_consistency_level][, column_reversed][, max_count]*)
Count the number of columns in the row with key *key*.

You may limit the columns or super columns counted to those in *columns*. Additionally, you may limit the columns or super columns counted to only those between *column_start* and *column_finish*.

You may also count only the number of subcolumns in a single super column using *super_column*. If this is set, *columns*, *column_start*, and *column_finish* only apply to the subcolumns of *super_column*.

To put an upper bound on the number of columns that are counted, set *max_count*.

multiget_count(*key[, super_column][, columns][, column_start][, column_finish][, super_column][, read_consistency_level][, buffer_size][, column_reversed][, max_count]*)
Perform a column count in parallel on a set of rows.

The parameters are the same as for `multiget()`, except that a list of keys may be used. A dictionary of the form `{key: int}` is returned.

buffer_size is the number of rows from the total list to count at a time. If left as `None`, the ColumnFamily's `buffer_size` will be used.

To put an upper bound on the number of columns that are counted, set *max_count*.

get_range(*[start][, finish][, columns][, column_start][, column_finish][, column_reversed][, column_count][, row_count][, include_timestamp][, super_column][, read_consistency_level][, buffer_size][, filter_empty]*)
Get an iterator over rows in a specified key range.

The key range begins with *start* and ends with *finish*. If left as empty strings, these extend to the beginning and end, respectively. Note that if RandomPartitioner is used, rows are stored in the order of the MD5 hash of their keys, so getting a lexicographical range of keys is not feasible.

The *row_count* parameter limits the total number of rows that may be returned. If left as `None`, the number of rows that may be returned is unlimted (this is the default).

When calling *get_range()*, the intermediate results need to be buffered if we are fetching many rows, otherwise the Cassandra server will overallocate memory and fail. *buffer_size* is the size of that buffer in number of rows. If left as `None`, the ColumnFamily's `buffer_size` attribute will be used.

When *filter_empty* is left as `True`, empty rows (including range ghosts) will be skipped and will not count towards *row_count*.

All other parameters are the same as those of `get()`.

A generator over `(key, {column_name: column_value})` is returned. To convert this to a list, use `list()` on the result.

get_indexed_slices(*index_clause[, columns][, column_start][, column_finish][, column_reversed][, column_count][, include_timestamp][, read_consistency_level][, buffer_size]*)
Similar to `get_range()`, but an `IndexClause` is used instead of a key range.

index_clause limits the keys that are returned based on expressions that compare the value of a column to a given value. At least one of the expressions in the `IndexClause` must be on an indexed column.

Note that Cassandra does not support secondary indexes or get_indexed_slices() for super column families.

See Also:

`create_index_clause()` and `create_index_expression()`

insert(*key, columns[, timestamp][, ttl][, write_consistency_level]*)
Insert or update columns in the row with key *key*.

columns should be a dictionary of columns or super columns to insert or update. If this is a standard column family, *columns* should look like `{column_name: column_value}`. If this is a super column family, *columns* should look like `{super_column_name: {sub_column_name: value}}`. If this is a counter column family, you may use integers as values and those will be used as counter adjustments.

A timestamp may be supplied for all inserted columns with *timestamp*.

ttl sets the "time to live" in number of seconds for the inserted columns. After this many seconds, Cassandra will mark the columns as deleted.

The timestamp Cassandra reports as being used for insert is returned.

batch_insert(*rows[, timestamp][, ttl][, write_consistency_level]*)
Like `insert()`, but multiple rows may be inserted at once.

The *rows* parameter should be of the form `{key: {column_name: column_value}}` if this is a standard column family or `{key: {super_column_name: {column_name: column_value}}}` if this is a super column family.

add(*key, column[, value][, super_column][, write_consistency_level]*)
Increment or decrement a counter.

value should be an integer, either positive or negative, to be added to a counter column. By default, *value* is 1. New in version 1.1.0: Available in Cassandra 0.8.0 and later.

remove(*key[, columns][, super_column][, write_consistency_level]*)
Remove a specified row or a set of columns within the row with key *key*.

A set of columns or super columns to delete may be specified using *columns*.

A single super column may be deleted by setting *super_column*. If *super_column* is specified, *columns* will apply to the subcolumns of *super_column*.

If *columns* and *super_column* are both `None`, the entire row is removed.

The timestamp used for the mutation is returned.

remove_counter(*key, column[, super_column][, write_consistency_level]*)
Remove a counter at the specified location.

Note that counters have limited support for deletes: if you remove a counter, you must wait to issue any following update until the delete has reached all the nodes and all of them have been fully compacted. New in version 1.1.0: Available in Cassandra 0.8.0 and later.

truncate()
Marks the entire ColumnFamily as deleted.

From the user's perspective, a successful call to `truncate` will result complete data deletion from this column family. Internally, however, disk space will not be immediatily released, as with all deletes in Cassandra, this one only marks the data as deleted.

The operation succeeds only if all hosts in the cluster at available and will throw an `UnavailableException` if some hosts are down.

batch(*self[, queue_size][, write_consistency_level]*)
Create batch mutator for doing multiple insert, update, and remove operations using as few roundtrips as possible.

The *queue_size* parameter sets the max number of mutations per request.

A `CfMutator` is returned.

pycassa.columnfamilymap – Maps Classes to Column Families

Provides a way to map an existing class of objects to a column family.

This can help to cut down boilerplate code related to converting objects to a row format and back again. ColumnFamilyMap is primarily useful when you have one "object" per row.

See Also:

`pycassa.types` for selecting data types for object attributes and infomation about creating custom data types.

class pycassa.columnfamilymap.**ColumnFamilyMap**(*cls*, *pool*, *column_family*[, *raw_columns*])
Maps an existing class to a column family. Class fields become columns, and instances of that class can be represented as rows in standard column families or super columns in super column families.

Instances of *cls* are returned from `get()`, `multiget()`, `get_range()` and `get_indexed_slices()`.

pool is a `ConnectionPool` that will be used in the same way a `ColumnFamily` uses one.

column_family is the name of a column family to tie to *cls*.

If *raw_columns* is `True`, all columns will be fetched into the *raw_columns* field in requests.

get(*key[, columns][, column_start][, column_finish][, column_count][, column_reversed][, super_column][, read_consistency_level]*)
Creates one or more instances of *cls* from the row with key *key*.

The fields that are retreived may be specified using *columns*, which should be a list of column names.

If the column family is a super column family, a list of *cls* instances will be returned, one for each super column. If the *super_column* parameter is not supplied, then *columns* specifies which super columns will be used to create instances of *cls*. If the *super_column* parameter *is* supplied, only one instance of *cls* will be returned; if *columns* is specified in this case, only those attributes listed in *columns* will be fetched.

All other parameters behave the same as in `ColumnFamily.get()`.

multiget(*keys[, columns][, column_start][, column_finish][, column_count][, column_reversed][, super_column][, read_consistency_level]*)
Like `get()`, but a list of keys may be specified.

The result of multiget will be a dictionary where the keys are the keys from the *keys* argument, minus any missing rows. The value for each key in the dictionary will be the same as if `get()` were called on that individual key.

get_range(*[start][, finish][, columns][, column_start][, column_finish][, column_reversed][, column_count][, row_count][, super_column][, read_consistency_level][, buffer_size]*)
Get an iterator over instances in a specified key range.

Like `multiget()`, whether a single instance or multiple instances are returned per-row when the column family is a super column family depends on what parameters are passed.

For an explanation of how `get_range()` works and a description of the parameters, see `ColumnFamily.get_range()`.

Example usage with a standard column family:

```
>>> pool = pycassa.ConnectionPool('Keyspace1')
>>> usercf =  pycassa.ColumnFamily(pool, 'Users')
>>> cfmap = pycassa.ColumnFamilyMap(MyClass, usercf)
>>> users = cfmap.get_range(row_count=2, columns=['name', 'age'])
>>> for key, user in users:
...     print user.name, user.age
Miles Davis 84
Winston Smith 42
```

get_indexed_slices(*index_clause[, columns][, column_start][, column_finish][, column_reversed][, column_count][, include_timestamp][, read_consistency_level][, buffer_size]*)
Fetches a list of instances that satisfy an index clause. Similar to `get_range()`, but uses an index clause instead of a key range.

See `ColumnFamily.get_indexed_slices()` for an explanation of the parameters.

insert(*instance[, columns][, write_consistency_level]*)
Insert or update stored instances.

instance should be an instance of *cls* to store.

The *columns* parameter allows to you specify which attributes of *instance* should be inserted or updated. If left as `None`, all attributes will be inserted.

batch_insert(*instances[, timestamp][, ttl][, write_consistency_level]*)
Insert or update stored instances.

instances should be a list containing instances of *cls* to store.

remove(*instance[, columns][, write_consistency_level]*)
Removes a stored instance.

The *columns* parameter is a list of columns that should be removed. If this is left as the default value of `None`, the entire stored instance will be removed.

pycassa.system_manager – Manage Schema Definitions

pycassa.system_manager.**SIMPLE_STRATEGY = 'SimpleStrategy'**
Replication strategy that simply chooses consecutive nodes in the ring for replicas

pycassa.system_manager.**NETWORK_TOPOLOGY_STRATEGY = 'NetworkTopologyStrategy'**
Replication strategy that puts a number of replicas in each datacenter

pycassa.system_manager.**OLD_NETWORK_TOPOLOGY_STRATEGY = 'OldNetworkTopologyStrategy'**
Original replication strategy for putting a number of replicas in each datacenter. This was originally called 'RackAwareStrategy'.

pycassa.system_manager.**KEYS_INDEX = 0**
A secondary index type where each indexed value receives its own row

class pycassa.system_manager.**SystemManager**(*server='localhost:9160'*, *credentials=None*, *framed_transport=True*, *timeout=30*, *socket_factory=<function default_socket_factory at 0x18d81b8>*, *transport_factory=<function default_transport_factory at 0x18fa2a8>*)

Lets you examine and modify schema definitions as well as get basic information about the cluster.

This class is mainly designed to be used manually in a python shell, not as part of a program, although it can be used that way.

All operations which modify a keyspace or column family definition will block until the cluster reports that all nodes have accepted the modification.

Example Usage:

```
>>> from pycassa.system_manager import *
>>> sys = SystemManager('192.168.10.2:9160')
>>> sys.create_keyspace('TestKeyspace', SIMPLE_STRATEGY, {'replication_factor': '1'})
>>> sys.create_column_family('TestKeyspace', 'TestCF', super=False,
...                          comparator_type=LONG_TYPE)
>>> sys.alter_column_family('TestKeyspace', 'TestCF', key_cache_size=42, gc_grace_seconds=1000)
>>> sys.drop_keyspace('TestKeyspace')
>>> sys.close()
```

close()
Closes the underlying connection

get_keyspace_column_families(*keyspace*, *use_dict_for_col_metadata=False*)
Returns a raw description of the keyspace, which is more useful for use in programs than `describe_keyspace()`.

If *use_dict_for_col_metadata* is `True`, the CfDef's column_metadata will be stored as a dictionary where the keys are column names instead of a list.

Returns a dictionary of the form `{column_family_name: CfDef}`

get_keyspace_properties(*keyspace*)
Gets a keyspace's properties.

Returns a `dict` with 'strategy_class' and 'strategy_options' as keys.

list_keyspaces()
Returns a list of all keyspace names.

describe_ring(*keyspace*)
Describes the Cassandra cluster

describe_cluster_name()
Gives the cluster name

describe_version()
Gives the server's API version

describe_schema_versions()
Lists what schema version each node has

describe_partitioner()
Gives the partitioner that the cluster is using

describe_snitch()
Gives the snitch that the cluster is using

create_keyspace(*name*, *replication_strategy='SimpleStrategy'*, *strategy_options=None*, *durable_writes=True*, ***ks_kwargs*)

Creates a new keyspace. Column families may be added to this keyspace after it is created using `create_column_family()`.

replication_strategy determines how replicas are chosen for this keyspace. The strategies that Cassandra provides by default are available as `SIMPLE_STRATEGY`, `NETWORK_TOPOLOGY_STRATEGY`, and `OLD_NETWORK_TOPOLOGY_STRATEGY`.

strategy_options is a dictionary of strategy options. For NetworkTopologyStrategy, the dictionary should look like `{'Datacenter1': '2', 'Datacenter2': '1'}`. This maps each datacenter (as defined in a Cassandra property file) to a replica count. For SimpleStrategy, you can specify the replication factor as follows: `{'replication_factor': '1'}`.

Example Usage:

```
>>> from pycassa.system_manager import *
>>> sys = SystemManager('192.168.10.2:9160')
>>> # Create a SimpleStrategy keyspace
>>> sys.create_keyspace('SimpleKS', SIMPLE_STRATEGY, {'replication_factor': '1'})
>>> # Create a NetworkTopologyStrategy keyspace
>>> sys.create_keyspace('NTS_KS', NETWORK_TOPOLOGY_STRATEGY, {'DC1': '2', 'DC2': '1'})
>>> sys.close()
```

alter_keyspace(*keyspace*, *replication_strategy=None*, *strategy_options=None*, *durable_writes=None*, ***ks_kwargs*)

Alters an existing keyspace.

> **Warning:** Don't use this unless you know what you are doing.

Parameters are the same as for `create_keyspace()`.

drop_keyspace(*keyspace*)

Drops a keyspace from the cluster.

create_column_family(*keyspace*, *name*, *column_validation_classes=None*, ***cf_kwargs*)

Creates a new column family in a given keyspace. If a value is not supplied for any of optional parameters, Cassandra will use a reasonable default value.

keyspace should be the name of the keyspace the column family will be created in. *name* gives the name of the column family.

alter_column_family(*keyspace*, *column_family*, *column_validation_classes=None*, ***cf_kwargs*)

Alters an existing column family.

Parameter meanings are the same as for `create_column_family()`.

drop_column_family(*keyspace*, *column_family*)

Drops a column family from the keyspace.

alter_column(*keyspace*, *column_family*, *column*, *value_type*)

Sets a data type for the value of a specific column.

value_type is a string that determines what type the column value will be. By default, `LONG_TYPE`, `INT_TYPE`, `ASCII_TYPE`, `UTF8_TYPE`, `TIME_UUID_TYPE`, `LEXICAL_UUID_TYPE` and `BYTES_TYPE` are provided. Custom types may be used as well by providing the class name; if the custom comparator class is not in `org.apache.cassandra.db.marshal`, the fully qualified class name must be given.

For super column families, this sets the subcolumn value type for any subcolumn named *column*, regardless of the super column name.

create_index(*keyspace*, *column_family*, *column*, *value_type*, *index_type=0*, *index_name=None*)
Creates an index on a column.

This allows efficient for index usage via `get_indexed_slices()`

column specifies what column to index, and *value_type* is a string that describes that column's value's data type; see `alter_column()` for a full description of *value_type*.

index_type determines how the index will be stored internally. Currently, `KEYS_INDEX` is the only option. *index_name* is an optional name for the index.

Example Usage:

```
>>> from pycassa.system_manager import *
>>> sys = SystemManager('192.168.2.10:9160')
>>> sys.create_index('Keyspace1', 'Standard1', 'birthdate', LONG_TYPE, index_name='bday_inde
>>> sys.close
```

drop_index(*keyspace*, *column_family*, *column*)
Drops an index on a column.

pycassa.index – Secondary Index Tools

Tools for using Cassandra's secondary indexes.

Example Usage:

```
>>> from pycassa.columnfamily import ColumnFamily
>>> from pycassa.pool import ConnectionPool
>>> from pycassa.index import *
>>>
>>> pool = ConnectionPool('Keyspace1')
>>> users = ColumnFamily(pool, 'Users')
>>> state_expr = create_index_expression('state', 'Utah')
>>> bday_expr = create_index_expression('birthdate', 1970, GT)
>>> clause = create_index_clause([state_expr, bday_expr], count=20)
>>> for key, user in users.get_indexed_slices(clause):
...     print user['name'] + ",", user['state'], user['birthdate']
John Smith, Utah 1971
Mike Scott, Utah 1980
Jeff Bird, Utah 1973
```

This gives you all of the rows (up to 20) which have a 'birthdate' value above 1970 and a state value of 'Utah'.

See Also:

`SystemManager` methods `create_index()` and `drop_index()`

pycassa.index.**EQ = 0**
Equality (==) operator for index expressions

pycassa.index.**GT = 2**
Greater-than (>) operator for index expressions

pycassa.index.**GTE = 1**
Greater-than-or-equal (>=) operator for index expressions

pycassa.index.**LT = 4**
Less-than (<) operator for index expressions

pycassa.index.**LTE = 3**
Less-than-or-equal (<=) operator for index expressions

static index.**create_index_expression**(*column_name*, *value*[, *op=EQ*])
Constructs an IndexExpression to use in an IndexClause

The expression will be applied to the column with name *column_name*. A match will only occur if the operator specified with *op* returns True when used on the actual column value and the *value* parameter.

The default operator is EQ, which tests for equality.

static index.**create_index_clause**(*expr_list[, start_key][, count]*)
Constructs an IndexClause for use with get_indexed_slices()

expr_list should be a list of IndexExpression objects that must be matched for a row to be returned. At least one of these expressions must be on an indexed column.

Cassandra will only return matching rows with keys after *start_key*. If this is the empty string, all rows will be considered. Keep in mind that this is not as meaningful unless an OrderPreservingPartitioner is used.

The number of rows to return is limited by *count*, which defaults to 100.

pycassa.batch – Batch Operations

The batch interface allows insert, update, and remove operations to be performed in batches. This allows a convenient mechanism for streaming updates or doing a large number of operations while reducing number of RPC roundtrips.

Batch mutator objects are synchronized and can be safely passed around threads.

```
>>> b = cf.batch(queue_size=10)
>>> b.insert('key1', {'col1':'value11', 'col2':'value21'})
>>> b.insert('key2', {'col1':'value12', 'col2':'value22'}, ttl=15)
>>> b.remove('key1', ['col2'])
>>> b.remove('key2')
>>> b.send()
```

One can use the *queue_size* argument to control how many mutations will be queued before an automatic send() is performed. This allows simple streaming of updates. If set to None, automatic checkpoints are disabled. Default is 100.

Supercolumns are supported:

```
>>> b = scf.batch()
>>> b.insert('key1', {'supercol1': {'colA':'value1a', 'colB':'value1b'}
...                   {'supercol2': {'colA':'value2a', 'colB':'value2b'}})
>>> b.remove('key1', ['colA'], 'supercol1')
>>> b.send()
```

You may also create a Mutator directly, allowing operations on multiple column families:

```
>>> b = Mutator(pool)
>>> b.insert(cf, 'key1', {'col1':'value1', 'col2':'value2'})
>>> b.insert(supercf, 'key1', {'subkey1': {'col1':'value1', 'col2':'value2'}})
>>> b.send()
```

Note: This interface does not implement atomic operations across column families. All the limitations of the *batch_mutate* Thrift API call applies. Remember, a mutation in Cassandra is always atomic per key per column family only.

Note: If a single operation in a batch fails, the whole batch fails.

In addition mutators can be used as context managers, where an implicit `send()` will be called upon exit.

```
>>> with cf.batch() as b:
...     b.insert('key1', {'col1':'value11', 'col2':'value21'})
...     b.insert('key2', {'col1':'value12', 'col2':'value22'})
```

Calls to `insert()` and `remove()` can also be chained:

```
>>> cf.batch().remove('foo').remove('bar').send()
```

class pycassa.batch.**Mutator**(*pool*, *queue_size=100*, *write_consistency_level=None*, *allow_retries=True*)

Batch update convenience mechanism.

Queues insert/update/remove operations and executes them when the queue is full or *send* is called explicitly.

pool is the `ConnectionPool` that will be used for operations.

After *queue_size* operations, `send()` will be executed automatically. Use 0 to disable automatic sends.

insert(*column_family, key, columns[, timestamp][, ttl]*)

Adds a single row insert to the batch.

column_family is the `ColumnFamily` that the insert will be executed on.

If this is used on a counter column family, integers may be used for column values, and they will be taken as counter adjustments.

remove(*column_family, key[, columns][, super_column][, timestamp]*)

Adds a single row remove to the batch.

column_family is the `ColumnFamily` that the remove will be executed on.

send([*write_consistency_level*])

Sends all operations currently in the batch and clears the batch.

class pycassa.batch.**CfMutator**(*column_family*, *queue_size=100*, *write_consistency_level=None*, *allow_retries=True*)

A `Mutator` that deals only with one column family.

column_family is the `ColumnFamily` that all operations will be executed on.

insert(*key, cols[, timestamp][, ttl]*)

Adds a single row insert to the batch.

remove(*key[, columns][, super_column][, timestamp]*)

Adds a single row remove to the batch.

pycassa.types – Data Type Descriptions

Data type definitions that are used when converting data to and from the binary format that the data will be stored in.

In addition to the default classes included here, you may also define custom types by creating a new class that extends `CassandraType`. For example, IntString, which stores an arbitrary integer as a string, may be defined as follows:

```
>>> class IntString(pycassa.types.CassandraType):
...
...     @staticmethod
...     def pack(intval):
...         return str(intval)
...
...     @staticmethod
```

```
...     def unpack(strval):
...         return int(strval)
```

class pycassa.types.**CassandraType**(*reversed=False*, *default=None*)
A data type that Cassandra is aware of and knows how to validate and sort. All of the other classes in this module are subclasses of this class.

If *reversed* is true and this is used as a column comparator, the columns will be sorted in reverse order.

The *default* parameter only applies to use of this with ColumnFamilyMap, where *default* is used if a row does not contain a column corresponding to this item.

class pycassa.types.**BytesType**
Stores data as a byte array

class pycassa.types.**AsciiType**
Stores data as ASCII text

class pycassa.types.**UTF8Type**
Stores data as UTF8 encoded text

class pycassa.types.**LongType**
Stores data as an 8 byte integer

class pycassa.types.**IntegerType**
Stores data as a variable-length integer. This is a more compact format for storing small integers than LongType, and the limits on the size of the integer are much higher. Changed in version 1.2.0: Prior to 1.2.0, this was always stored as a 4 byte integer.

class pycassa.types.**DoubleType**
Stores data as an 8 byte double

class pycassa.types.**FloatType**
Stores data as a 4 byte float

class pycassa.types.**DecimalType**
Stores an unlimited precision decimal number. *decimal.Decimal* objects are used by pycassa to represent these objects.

class pycassa.types.**DateType**
An 8 byte timestamp. This will be returned as a datetime.datetime instance by pycassa. Either datetime instances or timestamps will be accepted. Changed in version 1.7.0: Prior to 1.7.0, datetime objects were expected to be in local time. In 1.7.0 and beyond, naive datetimes are assumed to be in UTC and tz-aware objects will be automatically converted to UTC for storage in Cassandra.

class pycassa.types.**UUIDType**
Stores data as a type 1 or type 4 UUID

class pycassa.types.**TimeUUIDType**
Stores data as a version 1 UUID

class pycassa.types.**LexicalUUIDType**
Stores data as a non-version 1 UUID

class pycassa.types.**OldPycassaDateType**
This class can only read and write the DateType format used by pycassa versions 1.2.0 to 1.5.0.

This formats store the number of microseconds since the unix epoch, rather than the number of milliseconds, which is what cassandra-cli and other clients supporting DateType use. Changed in version 1.7.0: Prior to 1.7.0, datetime objects were expected to be in local time. In 1.7.0 and beyond, naive datetimes are assumed to be in UTC and tz-aware objects will be automatically converted to UTC for storage in Cassandra.

class pycassa.types.**IntermediateDateType**
This class is capable of reading either the DateType format by pycassa versions 1.2.0 to 1.5.0 or the correct format used in pycassa 1.5.1+. It will only write the new, correct format.

This type is a good choice when you are using DateType as the validator for non-indexed column values and you are in the process of converting from thee old format to the new format.

It almost certainly *should not be used* for row keys, column names (if you care about the sorting), or column values that have a secondary index on them. Changed in version 1.7.0: Prior to 1.7.0, datetime objects were expected to be in local time. In 1.7.0 and beyond, naive datetimes are assumed to be in UTC and tz-aware objects will be automatically converted to UTC for storage in Cassandra.

class pycassa.types.**CompositeType**(**components*)
A type composed of one or more components, each of which have their own type. When sorted, items are primarily sorted by their first component, secondarily by their second component, and so on.

Each of **components* should be an instance of a subclass of CassandraType.

See Also:

Composite Types

class pycassa.types.**DynamicCompositeType**(**aliases*)
A type composed of one or more components, each of which have their own type. When sorted, items are primarily sorted by their first component, secondarily by their second component, and so on.

Unlike CompositeType, DynamicCompositeType columns need not all be of the same structure. Each column can be composed of different component types.

Components are specified using a 2-tuple made up of a comparator type and value. Aliases for comparator types can optionally be specified with a dictionary during instantiation.

pycassa.util – Utilities

A combination of utilities used internally by pycassa and utilities available for use by others working with pycassa.

pycassa.util.**convert_time_to_uuid**(*time_arg*, *lowest_val=True*, *randomize=False*)
Converts a datetime or timestamp to a type 1 uuid.UUID.

This is to assist with getting a time slice of columns or creating columns when column names are TimeUUIDType. Note that this is done automatically in most cases if name packing and value packing are enabled.

Also, be careful not to rely on this when specifying a discrete set of columns to fetch, as the non-timestamp portions of the UUID will be generated randomly. This problem does not matter with slice arguments, however, as the non-timestamp portions can be set to their lowest or highest possible values.

Parameters

- **datetime** (datetime or timestamp) – The time to use for the timestamp portion of the UUID. Expected inputs to this would either be a datetime object or a timestamp with the same precision produced by time.time(). That is, sub-second precision should be below the decimal place.
- **lowest_val** (*bool*) – Whether the UUID produced should be the lowest possible value UUID with the same timestamp as datetime or the highest possible value.
- **randomize** (*bool*) – Whether the clock and node bits of the UUID should be randomly generated. The *lowest_val* argument will be ignored if this is true.

Return type uuid.UUID

Changed in version 1.7.0: Prior to 1.7.0, datetime objects were expected to be in local time. In 1.7.0 and beyond, naive datetimes are assumed to be in UTC and tz-aware objects will be automatically converted to UTC.

pycassa.util.**convert_uuid_to_time**(*uuid_arg*)
Converts a version 1 uuid.UUID to a timestamp with the same precision as time.time() returns. This is useful for examining the results of queries returning a v1 UUID.

Parameters **uuid_arg** – a version 1 UUID

Return type timestamp

class pycassa.util.**OrderedDict**(**args*, ***kwds*)
A dictionary which maintains the insertion order of keys.

A dictionary which maintains the insertion order of keys.

pycassa.logging.pycassa_logger – Pycassa Logging

Logging facilities for pycassa.

class pycassa.logging.pycassa_logger.**PycassaLogger**
The root logger for pycassa.

This uses a singleton-like pattern, so creating a new instance will always give you the same result. This means that you can adjust all of pycassa's logging by calling methods on any instance.

pycassa does *not* automatically add a handler to the logger, so logs will not be captured by default. You *must* add a logging.Handler() object to the root handler for logs to be captured. See the example usage below.

By default, the root logger name is 'pycassa' and the logging level is 'info'.

The available levels are:

- debug
- info
- warn
- error
- critical

Example Usage:

```
>>> import logging
>>> log = pycassa.PycassaLogger()
>>> log.set_logger_name('pycassa_library')
>>> log.set_logger_level('debug')
>>> log.get_logger().addHandler(logging.StreamHandler())
```

add_child_logger(*child_logger_name*, *name_change_callback*)
Adds a child logger to pycassa that will be updated when the logger name changes.

get_logger()
Returns the underlying logging.Logger instance.

get_logger_level()
Gets the logging level for all pycassa logging.

get_logger_name()
Gets the root logger name for pycassa.

set_logger_level(*level*)
Sets the logging level for all pycassa logging.

set_logger_name(*logger_name*)
Sets the root logger name for pycassa and all of its children loggers.

pycassa.logging.pool_stats_logger – Connection Pool Stats

class pycassa.logging.pool_stats_logger.**StatsLogger**
Basic stats logger that increment counts. You can plot these as *COUNTER* or *DERIVED* (RRD) or apply derivative (graphite) except for `opened`, which tracks the currently opened connections.

Usage:

```
>>> pool = ConnectionPool(...)
>>> stats_logger = StatsLogger()
>>> pool.add_listener(stats_logger)
>>>
>>> # use the pool for a while...
>>> import pprint
>>> pprint.pprint(pool.stats)
{'at_max': 0,
 'checked_in': 401,
 'checked_out': 403,
 'created': {'failure': 0, 'success': 0},
 'disposed': {'failure': 0, 'success': 0},
 'failed': 1,
 'list': 0,
 'opened': {'current': 2, 'max': 2},
 'recycled': 0}
```

Get your stats as `stats_logger.stats` and push them to your metrics system.

reset(**args*, ***kwargs*)
Reset all counters to 0

pycassa.contrib.stubs – Pycassa Stubs

A functional set of stubs to be used for unit testing.

Projects that use pycassa and need to run an automated unit test suite on a system like Jenkins can use these stubs to emulate interactions with Cassandra without spinning up a cluster locally.

class pycassa.contrib.stubs.**ColumnFamilyStub**(*pool=None*, *column_family=None*, *rows=None*)
Functional ColumnFamily stub object.

Acts very similar to a remote column family, supporting a basic version of the API. When instantiated, it registers itself with the supplied (stub) connection pool.

get(*key[, columns][, column_start][, column_finish][, include_timestamp]*)
Get a value from the column family stub.

multiget(*keys[, columns][, column_start][, column_finish][, include_timestamp]*)
Get multiple key values from the column family stub.

get_range(*[columns][, include_timestamp]*)
Currently just gets all values from the column family.

get_indexed_slices (*index_clause[, columns], include_timestamp]*)
Grabs rows that match a pycassa index clause.

See pycassa.index.create_index_clause() for creating such an index clause.

insert (*key*, *columns*[, *timestamp*])
Insert data to the column family stub.

remove (*key*[, *columns*])
Remove a key from the column family stub.

truncate ()
Clears all data from the column family stub.

batch (*self*)
Returns itself.

class pycassa.contrib.stubs.**ConnectionPoolStub**
Connection pool stub.

Notes created column families in self.column_families.

class pycassa.contrib.stubs.**SystemManagerStub**
Functional System Manager stub object.

Records when column families, columns, and indexes have been created. To see what has been recorded, look at self.column_families.

create_column_family (*keyspace*, *table_name*)
Create a column family and record its existence.

alter_column (*keyspace*, *table_name*, *column_name*, *column_type*)
Alter a column, recording its name and type.

create_index (*keyspace*, *table_name*, *column_name*, *column_type*)
Create an index, recording its name and type.

describe_schema_versions ()
Describes the schema based on a hash of the stub system state.

5.5 Changelog

5.5.1 Changes in Version 1.8.0

This release requires either Python 2.6 or 2.7. Python 2.4 and 2.5 are no longer supported. There are no concrete plans for Python 3 compatibility yet.

Features

- Add configurable socket_factory attribute and constructor parameter to ConnectionPool and SystemManager.
- Add SSL support via the new socket_factory attribute.
- Add support for DynamicCompositeType
- Add mock support through a new pycassa.contrib.stubs module

Bug Fixes

- Don't return closed connections to the pool. This was primarily a problem when operations failed after retrying up to the limit, resulting in a `MaximumRetryException` or `AllServersUnavailable`.
- Set keyspace for connection after logging in instead of before. This fixes authentication against Cassandra 1.2, which requires logging in prior to setting a keyspace.
- Specify correct UUID variant when creating v1 `uuid.UUID` objects from datetimes or timestamps
- Add 900ns to v1 `uuid.UUID` timestamps when the "max" TimeUUID for a specific datetime or timestamp is requested, such as a column slice end
- Also look at attributes of parent classes when creating columns from attributes in `ColumnFamilyMap`

Other

- Upgrade bundled Thrift-generated python to 19.35.0, generated with Thrift 0.9.0.

5.5.2 Changes in Version 1.7.2

This release fixes a minor bug and upgrades the bundled Cassandra Thrift client interface to 19.34.0, matching Cassandra 1.2.0-beta1. This doesn't affect any existing Thrift methods, only adds new ones (that aren't yet utilized by pycassa), so there should not be any breakage.

Bug Fixes

- Fix single-component composite packing
- Avoid cyclic imports during installation in setup.py

Other

- Travis CI integration

5.5.3 Changes in Version 1.7.1

This release has few changes, and should make for a smooth upgrade from 1.7.0.

Features

- Add support for DecimalType: `DecimalType`

Bug Fixes

- Fix bad slice ends when using `xget()` with composite columns and a *column_finish* parameter
- Fix bad documentation paths in debian packaging scripts

Other

- Add `__version__` and `__version_info__` attributes to the `pycassa` module

5.5.4 Changes in Version 1.7.0

This release has a few relatively large changes in it: a new connection pool stats collector, compatibility with Cassandra 0.7 through 1.1, and a change in timezone behavior for datetimes.

Before upgrading, take special care to make sure datetimes that you pass to pycassa (for TimeUUIDType or DateType data) are in UTC, and make sure your code expects to get UTC datetimes back in return.

Likewise, the SystemManager changes *should* be backwards compatible, but there may be minor differences, mostly in `create_column_family()` and `alter_column_family()`. Be sure to test any code that works programmatically with these.

Features

- Added `StatsLogger` for tracking `ConnectionPool` metrics
- Full Cassandra 1.1 compatibility in `SystemManager`. To support this, all column family or keyspace attributes that have existed since Cassandra 0.7 may be used as keyword arguments for `create_column_family()` and `alter_column_family()`. It is up to the user to know which attributes are available and valid for their version of Cassandra. As part of this change, the version-specific thrift-generated cassandra modules (`pycassa.cassandra.c07`, `pycassa.cassandra.c08`, and `pycassa.cassandra.c10`) have been replaced by `pycassa.cassandra`. A minor related change is that individual connections now now longer ask for the node's API version, and that information is no longer stored as an attribute of the `ConnectionWrapper`.

Bug Fixes

- Fix `xget()` paging for non-string comparators
- Add `batch_insert()` to `ColumnFamilyMap`
- Use *setattr* instead of directly updating the object's `__dict__` in `ColumnFamilyMap` to avoid breaking descriptors
- Fix single-column counter increments with `ColumnFamily.insert()`
- Include *AuthenticationException* and *AuthorizationException* in the `pycassa` module
- Support counters in `xget()`
- Sort column families in pycassaShell for display
- Raise `TypeError` when bad keyword arguments are used when creating a `ColumnFamily` object

Other

All `datetime` objects create by pycassa now use UTC as their timezone rather than the local timezone. Likewise, naive `datetime` objects that are passed to pycassa are now assumed to be in UTC time, but `tz_info` is respected if set.

Specifically, the types of data that you may need to make adjustments for when upgrading are TimeUUIDType and DateType (including OldPycassaDateType and IntermediateDateType).

5.5.5 Changes in Version 1.6.0

This release adds a few minor features and several important bug fixes.

The most important change to take note of if you are using composite comparators is the change to the default inclusive/exclusive behavior for slice ends.

Other than that, this should be a smooth upgrade from 1.5.x.

Features

- New script for easily building RPM packages
- Add request and parameter information to PoolListener callback
- Add `ColumnFamily.xget()`, a generator version of `get()` that automatically pages over columns in reasonably sized chunks
- Add support for Int32Type, a 4-byte signed integer format
- Add constants for the highest and lowest possible TimeUUID values to `pycassa.util`

Bug Fixes

- Various 2.4 syntax errors
- Raise `AllServersUnavailable` if `server_list` is empty
- Handle custom types inside of composites
- Don't erase `comment` when updating column families
- Match Cassandra's sorting of TimeUUIDType values when the timestamps tie. This could result in some columns being erroneously left off of the end of column slices when datetime objects or timestamps were used for `column_start` or `column_finish`
- Use gevent's queue in place of the stdlib version when gevent monkeypatching has been applied
- Avoid sub-microsecond loss of precision with TimeUUID timestamps when using `pycassa.util.convert_time_to_uuid()`
- Make default slice ends inclusive when using `CompositeType` comparator Previously, the end of the slice was exclusive by default (as was the start of the slice when `column_reversed` was `True`)

5.5.6 Changes in Version 1.5.1

This release only affects those of you using DateType data, which has been supported since pycassa 1.2.0. If you are using DateType, it is **very** important that you read this closely.

DateType data is internally stored as an 8 byte integer timestamp. Since version 1.2.0 of pycassa, the timestamp stored has counted the number of *microseconds* since the unix epoch. The actual format that Cassandra standardizes on is *milliseconds* since the epoch.

If you are only using pycassa, you probably won't have noticed any problems with this. However, if you try to use cassandra-cli, sstable2json, Hector, or any other client that supports DateType, DateType data written by pycassa will appear to be far in the future. Similarly, DateType data written by other clients will appear to be in the past when loaded by pycassa.

This release changes the default DateType behavior to comply with the standard, millisecond-based format. **If you use DateType, and you upgrade to this release without making any modifications, you will have problems.** Unfortunately, this is a bit of a tricky situation to resolve, but the appropriate actions to take are detailed below.

To temporarily continue using the old behavior, a new class has been created: `pycassa.types.OldPycassaDateType`. This will read and write DateType data exactly the same as pycassa 1.2.0 to 1.5.0 did.

If you want to convert your data to the new format, the other new class, `pycassa.types.IntermediateDateType`, may be useful. It can read either the new or old format correctly (unless you have used dates close to 1970 with the new format) and will write only the new format. The best case for using this is if you have DateType validated columns that don't have a secondary index on them.

To tell pycassa to use `OldPycassaDateType` or `IntermediateDateType`, use the `ColumnFamily` attributes that control types: `column_name_class`, `key_validation_class`, `column_validators`, and so on. Here's an example:

```
from pycassa.types import OldPycassaDateType, IntermediateDateType
from pycassa.column_family import ColumnFamily
from pycassa.pool import ConnectionPool

pool = ConnectionPool('MyKeyspace', ['192.168.1.1'])

# Our tweet timeline has a comparator_type of DateType
tweet_timeline_cf = ColumnFamily(pool, 'tweets')
tweet_timeline_cf.column_name_class = OldPycassaDateType()

# Our tweet timeline has a comparator_type of DateType
users_cf = ColumnFamily(pool, 'users')
users_cf.column_validators['join_date'] = IntermediateDateType()
```

If you're using DateType for the *key_validation_class*, column names, column values with a secondary index on them, or are using the DateType validated column as a non-indexed part of an index clause with *get_indexed_slices()* (eg. "where state = 'TX' and join_date > 2012"), you need to be more careful about the conversion process, and `IntermediateDateType` probably isn't a good choice.

In most of cases, if you want to switch to the new date format, a manual migration script to convert all existing DateType data to the new format will be needed. In particular, if you convert keys, column names, or indexed columns on a live data set, be very careful how you go about it. If you need any assistance or suggestions at all with migrating your data, please feel free to send an email to tyler@datastax.com; I would be glad to help.

5.5.7 Changes in Version 1.5.0

The main change to be aware of for this release is the new no-retry behavior for counter operations. If you have been maintaining a separate connection pool with retries disabled for usage with counters, you may discontinue that practice after upgrading.

Features

- By default, counter operations will not be retried automatically. This makes it easier to use a single connection pool without worrying about overcounting.

Bug Fixes

- Don't remove entire row when an empty list is supplied for the *columns* parameter of `remove()` or the batch remove methods.
- Add python-setuptools to debian build dependencies
- Batch `remove()` was not removing subcolumns when the specified supercolumn was 0 or other "falsey" values
- Don't request an extra row when reading fewer than *buffer_size* rows with `get_range()` or `get_indexed_slices()`.
- Remove *pool_type* from logs, which showed up as `None` in recent versions
- Logs were erroneously showing the same server for retries of failed operations even when the actual server being queried had changed

5.5.8 Changes in Version 1.4.0

This release is primarily a bugfix release with a couple of minor features and removed deprecated items.

Features

- Accept column_validation_classes when creating or altering column families with SystemManager
- Ignore UNREACHABLE nodes when waiting for schema version agreement

Bug Fixes

- Remove accidental print statement in SystemManager
- Raise TypeError when unexpected types are used for comparator or validator types when creating or altering a Column Family
- Fix packing of column values using column-specific validators during batch inserts when the column name is changed by packing
- Always return timestamps from inserts
- Fix NameError when timestamps are used where a DateType is expected
- Fix NameError in python 2.4 when unpacking DateType objects
- Handle reading composites with trailing components missing
- Upgrade ez_setup.py to fix broken setuptools link

Removed Deprecated Items

- `pycassa.connect()`
- `pycassa.connect_thread_local()`
- `ConnectionPool.status()`
- `ConnectionPool.recreate()`

5.5.9 Changes in Version 1.3.0

This release adds full compatibility with Cassandra 1.0 and removes support for schema manipulation in Cassandra 0.7.

In this release, schema manipulation should work with Cassandra 0.8 and 1.0, but not 0.7. The data API should continue to work with all three versions.

Bug Fixes

- Don't ignore *columns* parameter in `ColumnFamilyMap.insert()`
- Handle empty instance fields in `ColumnFamilyMap.insert()`
- Use the same default for *timeout* in `pycassa.connect()` as `ConnectionPool` uses
- Fix typo which caused a different exception to be thrown when an `AllServersUnavailable` exception was raised
- IPython 0.11 compatibility in pycassaShell
- Correct dependency declaration in `setup.py`
- Add UUIDType to supported types

Features

- The *filter_empty* parameter was added to `get_range()` with a default of `True`; this allows empty rows to be kept if desired

Deprecated

- `pycassa.connect()`
- `pycassa.connect_thread_local()`

5.5.10 Changes in Version 1.2.1

This is strictly a bug-fix release addressing a few issues created in 1.2.0.

Bug Fixes

- Correctly check for Counters in `ColumnFamily` when setting *default_validation_class*
- Pass kwargs in `ColumnFamilyMap` to `ColumnFamily`
- Avoid potential UnboundLocal in `ConnectionPool.execute()` when `get()` fails
- Fix ez_setup dependency/bundling so that package installations using easy_install or pip don't fail without ez_setup installed

5.5.11 Changes in Version 1.2.0

This should be a fairly smooth upgrade from pycassa 1.1. The primary changes that may introduce minor incompatibilities are the changes to `ColumnFamilyMap` and the automatic skipping of "ghost ranges" in `ColumnFamily.get_range()`.

Features

- Add `ConnectionPool.fill()`
- Add `FloatType`, `DoubleType`, `DateType`, and `BooleanType` support.
- Add `CompositeType` support for static composites. See *Composite Types* for more details.
- Add *timestamp*, *ttl* to `ColumnFamilyMap.insert()` params
- Support variable-length integers with `IntegerType`. This allows more space-efficient small integers as well as integers that exceed the size of a long.
- Make `ColumnFamilyMap` a subclass of `ColumnFamily` instead of using one as a component. This allows all of the normal adjustments normally done to a `ColumnFamily` to be done to a `ColumnFamilyMap` instead. See *Class Mapping with Column Family Map* for examples of using the new version.
- Expose the following `ConnectionPool` attributes, allowing them to be altered after creation: `max_overflow`, `pool_timeout`, `recycle`, `max_retries`, and `logging_name`. Previously, these were all supplied as constructor arguments. Now, the preferred way to set them is to alter the attributes after creation. (However, they may still be set in the constructor by using keyword arguments.)
- Automatically skip "ghost ranges" in `ColumnFamily.get_range()`. Rows without any columns will not be returned by the generator, and these rows will not count towards the supplied *row_count*.

Bug Fixes

- Add connections to `ConnectionPool` more readily when *prefill* is `False`. Before this change, if the ConnectionPool was created with `prefill=False`, connections would only be added to the pool when there was concurrent demand for connections. After this change, if `prefill=False` and `pool_size=N`, the first *N* operations will each result in a new connection being added to the pool.
- Close connection and adjust the `ConnectionPool`'s connection count after a `TApplicationException`. This exception generally indicates programmer error, so it's not extremely common.
- Handle typed keys that evaluate to `False`

Deprecated

- `ConnectionPool.recreate()`
- `ConnectionPool.status()`

Miscellaneous

- Better failure messages for `ConnectionPool` failures
- More efficient packing and unpacking

- More efficient multi-column inserts in `ColumnFamily.insert()` and `ColumnFamily.batch_insert()`
- Prefer Python 2.7's `collections.OrderedDict` over the bundled version when available

5.5.12 Changes in Version 1.1.1

Features

- Add `max_count` and `column_reversed` params to `get_count()`
- Add `max_count` and `column_reversed` params to `multiget_count()`

Bug Fixes

- Don't retry operations after a `TApplicationException`. This exception is reserved for programmatic errors (such as a bad API parameters), so retries are not needed.
- If the read_consistency_level kwarg was used in a `ColumnFamily` constructor, it would be ignored, resulting in a default read consistency level of `ONE`. This did not affect the read consistency level if it was specified in any other way, including per-method or by setting the `read_consistency_level` attribute.

5.5.13 Changes in Version 1.1.0

This release adds compatibility with Cassandra 0.8, including support for counters and key_validation_class. This release is backwards-compatible with Cassandra 0.7, and can support running against a mixed cluster of both Cassandra 0.7 and 0.8.

Changes related to Cassandra 0.8

- Addition of `COUNTER_COLUMN_TYPE` to `system_manager`.
- Several new column family attributes, including `key_validation_class`, `replicate_on_write`, `merge_shards_chance`, `row_cache_provider`, and `key_alias`.
- The new `ColumnFamily.add()` and `ColumnFamily.remove_counter()` methods.
- Support for counters in `pycassa.batch` and `ColumnFamily.batch_insert()`.
- Autopacking of keys based on `key_validation_class`.

Other Features

- `ColumnFamily.multiget()` now has a *buffer_size* parameter
- `ColumnFamily.multiget_count()` now returns rows in the order that the keys were passed in, similar to how `multiget()` behaves. It also uses the `dict_class` attribute for the containing class instead of always using a `dict`.
- Autpacking behavior is now more transparent and configurable, allowing the user to get functionality similar to the CLI's `assume` command, whereby items are packed and unpacked as though they were a certain data type, even if Cassandra does not use a matching comparator type or validation class. This behavior can be controlled through the following attributes:
  - `ColumnFamily.column_name_class`

  - ColumnFamily.super_column_name_class
  - ColumnFamily.key_validation_class
  - ColumnFamily.default_validation_class
  - ColumnFamily.column_validators
- A ColumnFamily may reload its schema to handle changes in validation classes with ColumnFamily.load_schema().

Bug Fixes

There were several related issues with overlow in ConnectionPool:

- Connection failures when a ConnectionPool was in a state of overflow would not result in adjustment of the overflow counter, eventually leading the ConnectionPool to refuse to create new connections.
- Settings of -1 for ConnectionPool.overflow erroneously caused overflow to be disabled.
- If overflow was enabled in conjunction with *prefill* being disabled, the effective overflow limit was raised to max_overflow + pool_size.

Other

- Overflow is now disabled by default in ConnectionPool.
- ColumnFamilyMap now sets the underlying ColumnFamily's autopack_names and autopack_values attributes to False upon construction.
- Documentation and tests will no longer be included in the packaged tarballs.

Removed Deprecated Items

The following deprecated items have been removed:

- ColumnFamilyMap.get_count()
- The *instance* parameter from ColumnFamilyMap.get_indexed_slices()
- The Int64 Column type.
- SystemManager.get_keyspace_description()

Deprecated

Athough not technically deprecated, most ColumnFamily constructor arguments should instead be set by setting the corresponding attribute on the ColumnFamily after construction. However, all previous constructor arguments will continue to be supported if passed as keyword arguments.

5.5.14 Changes in Version 1.0.8

- Pack IndexExpression values in get_indexed_slices() that are supplied through the IndexClause instead of just the *instance* parameter.

- Column names and values which use Cassandra's IntegerType are unpacked as though they are in a BigInteger-like format. This is (backwards) compatible with the format that pycassa uses to pack IntegerType data. This fixes an incompatibility with the format that cassandra-cli and other clients use to pack IntegerType data.
- Restore Python 2.5 compatibility that was broken through out of order keyword arguments in `ConnectionWrapper`.
- Pack *column_start* and *column_finish* arguments in `ColumnFamily *get*()` methods when the *super_column* parameter is used.
- Issue a `DeprecationWarning` when a method, parameter, or class that has been deprecated is used. Most of these have been deprecated for several releases, but no warnings were issued until now.
- Deprecations are now split into separate sections for each release in the changelog.

Deprecated

- The *instance* parameter of `ColumnFamilyMap.get_indexed_slices()`

5.5.15 Changes in Version 1.0.7

- Catch KeyError in `pycassa.columnfamily.ColumnFamily.multiget()` empty row removal. If the same non-existent key was passed multiple times, a `KeyError` was raised when trying to remove it from the OrderedDictionary after the first removal. The `KeyError` is caught and ignored now.
- Handle connection failures during retries. When a connection fails, it tries to create a new connection to replace itself. Exceptions during this process were not properly handled; they are now handled and count towards the retry count for the current operation.
- Close connection when a `MaximumRetryException` is raised. Normally a connection is closed when an operation it is performing fails, but this was not happening for the final failure that triggers the `MaximumRetryException`.

5.5.16 Changes in Version 1.0.6

- Add `EOFError` to the list of exceptions that cause a connection swap and retry
- Improved autopacking efficiency for AsciiType, UTF8Type, and BytesType
- Preserve sub-second timestamp precision in datetime arguments for insertion or slice bounds where a TimeUUID is expected. Previously, precision below a second was lost.
- In a `MaximumRetryException`'s message, include details about the last `Exception` that caused the `MaximumRetryException` to be raised
- `pycassa.pool.ConnectionPool.status()` now always reports a non-negative overflow; 0 is now used when there is not currently any overflow
- Created `pycassa.types.Long` as a replacement for `pycassa.types.Int64`. `Long` uses big-endian encoding, which is compatible with Cassandra's LongType, while `Int64` used little-endian encoding.

Deprecated

- `pycassa.types.Int64` has been deprecated in favor of `pycassa.types.Long`

5.5.17 Changes in Version 1.0.5

- Assume port 9160 if only a hostname is given
- Remove super_column param from pycassa.columnfamily.ColumnFamily.get_indexed_slices()
- Enable failover on functions that previously lacked it
- Increase base backoff time to 0.01 seconds
- Add a timeout paremeter to pycassa.system_manager.SystemManger
- Return timestamp on single-column inserts

5.5.18 Changes in Version 1.0.4

- Fixed threadlocal issues that broke multithreading
- Fix bug in pycassa.columnfamily.ColumnFamily.remove() when a super_column argument is supplied
- Fix minor PoolLogger logging bugs
- Added pycassa.system_manager.SystemManager.describe_partitioner()
- Added pycassa.system_manager.SystemManager.describe_snitch()
- Added pycassa.system_manager.SystemManager.get_keyspace_properties()
- Moved pycassa.system_manager.SystemManager.describe_keyspace() and pycassa.system_manager.SystemManager.describe_column_family() to pycassaShell describe_keyspace() and describe_column_family()

Deprecated

- Renamed pycassa.system_manager.SystemManager.get_keyspace_description() to pycassa.system_manager.SystemManager.get_keyspace_column_families() and deprecated the previous name

5.5.19 Changes in Version 1.0.3

- Fixed supercolumn slice bug in get()
- pycassaShell now runs scripts with execfile to allow for multiline statements
- 2.4 compatability fixes

5.5.20 Changes in Version 1.0.2

- Failover handles a greater set of potential failures
- pycassaShell now loads/reloads pycassa.columnfamily.ColumnFamily instances when the underlying column family is created or updated
- Added an option to pycassaShell to run a script after startup
- Added pycassa.system_manager.SystemManager.list_keyspaces()

5.5.21 Changes in Version 1.0.1

- Allow pycassaShell to be run without specifying a keyspace
- Added `pycassa.system_manager.SystemManager.describe_schema_versions()`

5.5.22 Changes in Version 1.0.0

- Created the `SystemManager` class to allow for keyspace, column family, and index creation, modification, and deletion. These operations are no longer provided by a Connection class.
- Updated pycassaShell to use the SystemManager class
- Improved retry behavior, including exponential backoff and proper resetting of the retry attempt counter
- Condensed connection pooling classes into only `pycassa.pool.ConnectionPool` to provide a simpler API
- Changed `pycassa.connection.connect()` to return a connection pool
- Use more performant Thrift API methods for `insert()` and `get()` where possible
- Bundled `OrderedDict` and set it as the default dictionary class for column families
- Provide better `TypeError` feedback when columns are the wrong type
- Use Thrift API 19.4.0

Deprecated

- `ColumnFamilyMap.get_count()` has been deprecated. Use `ColumnFamily.get_count()` instead.

5.5.23 Changes in Version 0.5.4

- Allow for more backward and forward compatibility
- Mark a server as being down more quickly in `Connection`

5.5.24 Changes in Version 0.5.3

- Added `PooledColumnFamily`, which makes it easy to use connection pooling automatically with a ColumnFamily.

5.5.25 Changes in Version 0.5.2

- Support for adding/updating/dropping Keyspaces and CFs in `pycassa.connection.Connection`
- `get_range()` optimization and more configurable batch size
- batch `get_indexed_slices()` similar to `ColumnFamily.get_range()`
- Reorganized pycassa logging
- More efficient packing of data types
- Fix error condition that results in infinite recursion
- Limit pooling retries to only appropriate exceptions

- Use Thrift API 19.3.0

5.5.26 Changes in Version 0.5.1

- Automatically detect if a column family is a standard column family or a super column family
- `multiget_count()` support
- Allow preservation of key order in `multiget()` if an ordered dictionary is used
- Convert timestamps to v1 UUIDs where appropriate
- pycassaShell documentation
- Use Thrift API 17.1.0

5.5.27 Changes in Version 0.5.0

- Connection Pooling support: `pycassa.pool`
- Started moving logging to `pycassa.logger`
- Use Thrift API 14.0.0

5.5.28 Changes in Version 0.4.3

- Autopack on CF's default_validation_class
- Use Thrift API 13.0.0

5.5.29 Changes in Version 0.4.2

- Added batch mutations interface: `pycassa.batch`
- Made bundled thrift-gen code a subpackage of pycassa
- Don't attempt to reencode already encoded UTF8 strings

5.5.30 Changes in Version 0.4.1

- Added `batch_insert()`
- Redifined `insert()` in terms of `batch_insert()`
- Fixed UTF8 autopacking
- Convert datetime slice args to uuids when appropriate
- Changed how thrift-gen code is bundled
- Assert that the major version of the thrift API is the same on the client and on the server
- Use Thrift API 12.0.0

5.5.31 Changes in Version 0.4.0

- Added pycassaShell, a simple interactive shell
- Converted the test config from xml to yaml
- Fixed overflow error on `get_count()`
- Only insert columns which exist in the model object
- Make ColumnFamilyMap not ignore the ColumnFamily's dict_class
- Specify keyspace as argument to `connect()`
- Add support for framed transport and default to using it
- Added autopacking for column names and values
- Added support for secondary indexes with `get_indexed_slices()` and `pycassa.index`
- Added `truncate()`
- Use Thrift API 11.0.0

5.6 Assorted Cassandra and pycassa Functionality

These sections document how to make use of various features offered by either Cassandra or pycassa.

5.6.1 Secondary Indexes

Cassandra supports secondary indexes, which allow you to efficiently get only rows which match a certain expression.

Here's a description of secondary indexes and how to use them.

In order to use `get_indexed_slices()` to get data from an indexed column family using the indexed column, we need to create an `IndexClause` which contains a list of `IndexExpression` objects. The `IndexExpression` objects inside the clause are ANDed together, meaning every expression must match for a row to be returned.

Suppose we have a 'Users' column family with one row per user, and we want to get all of the users from Utah with a birthdate after 1970. We can make use of the `pycassa.index` module to make this easier:

```
>>> from pycassa.pool import ConnectionPool
>>> from pycassa.columnfamily import ColumnFamily
>>> from pycassa.index import *
>>> pool = ConnectionPool('Keyspace1')
>>> users = ColumnFamily(pool, 'Users')
>>> state_expr = create_index_expression('state', 'Utah')
>>> bday_expr = create_index_expression('birthdate', 1970, GT)
>>> clause = create_index_clause([state_expr, bday_expr], count=20)
>>> for key, user in users.get_indexed_slices(clause):
...     print user['name'] + ",", user['state'], user['birthdate']
John Smith, Utah 1971
Mike Scott, Utah 1980
Jeff Bird, Utah 1973
```

Although at least one `IndexExpression` in the clause must be on an indexed column, you may also have other expressions which are on non-indexed columns.

5.6.2 Super Columns

Cassandra allows you to group columns in "super columns". You would create a super column family through `cassandra-cli` in the following way:

```
[default@keyspace1] create column family Super1 with column_type=Super;
632cf985-645e-11e0-ad9e-e700f669bcfc
Waiting for schema agreement...
... schemas agree across the cluster
```

To use a super column in **pycassa**, you only need to add an extra level to the dictionary:

```
>>> col_fam = pycassa.ColumnFamily(pool, 'Super1')
>>> col_fam.insert('row_key', {'supercol_name': {'col_name': 'col_val'}})
1354491238721345
>>> col_fam.get('row_key')
{'supercol_name': {'col_name': 'col_val'}}
```

The *super_column* parameter for `get()`-like methods allows you to be selective about what subcolumns you get from a single super column.

```
>>> col_fam = pycassa.ColumnFamily(pool, 'Letters')
>>> col_fam.insert('row_key', {'lowercase': {'a': '1', 'b': '2', 'c': '3'}})
1354491239132744
>>> col_fam.get('row_key', super_column='lowercase')
{'supercol1': {'a': '1', 'b': '2', 'c': '3'}}
>>> col_fam.get('row_key', super_column='lowercase', columns=['a', 'b'])
{'supercol1': {'a': '1', 'b': '2'}}
>>> col_fam.get('row_key', super_column='lowercase', column_start='b')
{'supercol1': {'b': '1', 'c': '2'}}
>>> col_fam.get('row_key', super_column='lowercase', column_finish='b', column_reversed=True)
{'supercol1': {'c': '2', 'b': '1'}}
```

5.6.3 Composite Types

pycassa currently supports static CompositeTypes. DynamicCompositeType support is planned.

Creating a CompositeType Column Family

When creating a column family, you can specify a CompositeType comparator or validator using `CompositeType` in conjunction with the other types in `pycassa.types`.

```
>>> from pycassa.types import *
>>> from pycassa.system_manager import *
>>>
>>> sys = SystemManager()
>>> comparator = CompositeType(LongType(reversed=True), AsciiType())
>>> sys.create_column_family("Keyspace1", "CF1", comparator_type=comparator)
```

This example creates a column family with column names that have two components. The first component is a `LongType`, sorted in reverse order; the second is a normally sorted `AsciiType`.

You may put an aribtrary number of components in a `CompositeType`, and each component may be reversed or not.

Insert CompositeType Data

When inserting data, where a `CompositeType` is expected, you should supply a tuple which includes all of the components.

Continuing the example from above:

```
>>> cf = ColumnFamily(pool, "CF1")
>>> cf.insert("key", {(1234, "abc"): "colval"})
```

When dealing with composite keys or column values, supply tuples in exactly the same manner.

Fetching CompositeType Data

`CompositeType` data is also returned in a tuple format.

```
>>> cf.get("key")
{(1234, "abc"): "colval"}
```

When fetching a slice of columns, slice ends are specified using tuples as well. However, you are only required to supply at least the first component of the `CompositeType`; elements may be left off of the end of the tuple in order to slice columns based on only the first or first few components.

For example, suppose our *comparator_type* is `CompositeType(LongType, AsciiType, LongType)`. Valid slice ends would include `(1, )`, `(1, "a")`, and `(1, "a", 2011)`.

If you supply a slice start and a slice end that only specify the first component, you will get back all columns where the first component falls in that range, regardless of what the value of the other components is.

When slicing columns, the second component is only compared to the second component of the slice start if the first component of the column name matches the first component of the slice start. Likewise with the slice end, the second component will only be checked if the first components match. In essence, components after the first only serve as "tie-breakers" at the slice ends, and have no effect in the "middle" of the slice. Keep in mind the sorted order of the columns within Cassandra, and that when you get a slice of columns, you can only get a contiguous slice, not separate chunks out of the row.

Inclusive or Exclusive Slice Ends

By default, slice ends are inclusive on the final component you supply for that slice end. This means that if you give a `column_finish` of `(123, "b")`, then columns named `(123, "a", 2011)`, `(123, "b", 0)`, and `(123, "b" 123098123012)` would all be returned.

With composite types, you have the option to make the slice start and finish exlusive. To do so, replace the final component in your slice end with a tuple like `(value, False)`. (Think of the `False` as being short for `inclusive=False`. You can also explicitly specify `True`, but this is redundant.) Now, if you gave a `column_finish` of `(123, ("b", False))`, you would only get back `(123, "a", 2011)`. The same principle applies for `column_start`.

5.6.4 Class Mapping with Column Family Map

You can map existing classes to column families using `ColumnFamilyMap`.

To specify the fields to be persisted, use any of the subclasses of `pycassa.types.CassandraType` available in `pycassa.types`.

```
>>> from pycassa.types import *
>>> class User(object):
...     key = LexicalUUIDType()
...     name = Utf8Type()
...     age = IntegerType()
...     height = FloatType()
...     score = DoubleType(default=0.0)
...     joined = DateType()
```

The defaults will be filled in whenever you retrieve instances from the Cassandra server and the column doesn't exist. If you want to add a column in the future, you can simply add the relevant attribute to the class and the default value will be used when you get old instances.

```
>>> from pycassa.pool import ConnectionPool
>>> from pycassa.columnfamilymap import ColumnFamilyMap
>>>
>>> pool = ConnectionPool('Keyspace1')
>>> cfmap = ColumnFamilyMap(User, pool, 'users')
```

All the functions are exactly the same as for `ColumnFamily`, except that they return instances of the supplied class when possible.

```
>>> from datetime import datetime
>>> import uuid
>>>
>>> key = uuid.uuid4()
>>>
>>> user = User()
>>> user.key = key
>>> user.name = 'John'
>>> user.age = 18
>>> user.height = 5.9
>>> user.joined = datetime.now()
>>> cfmap.insert(user)
1261395560186855

>>> user = cfmap.get(key)
>>> user.name
"John"
>>> user.age
18

>>> users = cfmap.multiget([key1, key2])
>>> print users[0].name
"John"
>>> for user in cfmap.get_range():
...    print user.name
"John"
"Bob"
"Alex"

>>> cfmap.remove(user)
1261395603906864
>>> cfmap.get(user.key)
Traceback (most recent call last):
...
cassandra.ttypes.NotFoundException: NotFoundException()
```

5.6.5 Version 1 UUIDs (TimeUUIDType)

Version 1 UUIDs are frequently used for timelines instead of timestamps. Normally, this makes it difficult to get a slice of columns for some time range or to create a column name or value for some specific time.

To make this easier, if a `datetime` object or a timestamp with the same precision as the output of `time.time()` is passed where a TimeUUID is expected, **pycassa** will convert that into a `uuid.UUID` with an equivalent timestamp component.

Suppose we have something like Twissandra's public timeline but with TimeUUIDs for column names. If we want to get all tweets that happened yesterday, we can do:

```
>>> import datetime
>>> line = pycassa.ColumnFamily(pool, 'Userline')
>>> today = datetime.datetime.utcnow()
>>> yesterday = today - datetime.timedelta(days=1)
>>> tweets = line.get('__PUBLIC__', column_start=yesterday, column_finish=today)
```

Now, suppose there was a tweet that was supposed to be posted on December 11th at 8:02:15, but it was dropped and now we need to put it in the public timeline. There's no need to generate a UUID, we can just pass another datetime object instead:

```
>>> from datetime import datetime
>>> line = pycassa.ColumnFamily(pool, 'Userline')
>>> time = datetime(2010, 12, 11, 8, 2, 15)
>>> line.insert('__PUBLIC__', {time: 'some tweet stuff here'})
```

One limitation of this is that you can't ask for one specific column with a TimeUUID name by passing a `datetime` through something like the *columns* parameter for `get()`; this is because there is no way to know the non-timestamp components of the UUID ahead of time. Instead, simply pass the same `datetime` object for both *column_start* and *column_finish* and you'll get one or more columns for that exact moment in time.

Note that Python does not sort UUIDs the same way that Cassandra does. When Cassandra sorts V1 UUIDs it first compares the time component, and then the raw bytes of the UUID. Python on the other hand just sorts the raw bytes. If you need to sort UUIDs in Python the same way Cassandra does you will want to use something like this:

```
>>> import uuid, random
>>> uuids = [uuid.uuid1() for _ xrange(10)]
>>> random.shuffle(uuids)
>>> improperly_sorted = sorted(uuids)
>>> properly_sorted = sorted(uuids, key=lambda k: (k.time, k.bytes))
```

5.6.6 pycassaShell

pycassaShell is an interactive Cassandra python shell. It is useful for exploring Cassandra, especially for those who are just beginning.

Requirements

Python 2.6 or later is required.

Make sure you have **pycassa** installed as shown in *Installing*.

It is **strongly** recommended that you have IPython, an enhanced interactive python shell, installed. This gives you tab completion, colors, and working arrow keys!

On Debian based systems, this can be installed by:

```
apt-get install ipython
```

Alternatively, if `easy_install` is available:

```
easy_install ipython
```

Usage

```
pycassaShell -k KEYSPACE [OPTIONS]
```

The available options are:

- `-H`, `--host` - The hostname to connect to. Defaults to 'localhost'
- `-p`, `--port` - The port to connect to. Defaults to 9160.
- `-u`, `--user` - If authentication or authorization are enabled, this username is used.
- `-P`, `--passwd` - If authentication or authorization are enabled, this password is used.
- `-S`, `--streaming` - Use a streaming transport. Works with Cassandra 0.6.x and below.
- `-F`, `--framed` - Use a framed transport. Works with Cassandra 0.7.x. This is the default.

When pycassaShell starts, it creates a `ColumnFamily` for every existing column family and prints the names of the objects. You can use these to easily insert and retrieve data from Cassandra.

```
>>> STANDARD1.insert('key', {'colname': 'val'})
1286048238391943
>>> STANDARD1.get('key')
{'colname': 'val'}
```

If you are interested in the keyspace and column family definitions, **pycassa** provides several methods that can be used with `SYSTEM_MANAGER`:

- `create_keyspace()`
- `alter_keyspace()`
- `drop_keyspace()`
- `create_column_family()`
- `alter_column_family()`
- `drop_column_family()`
- `alter_column()`
- `create_index()`
- `drop_index()`

Example usage:

```
>>> describe_keyspace('Keyspace1')

Name:                                Keyspace1

Replication Strategy:                SimpleStrategy
Replication Factor:                  1

Column Families:
   Indexed1
```

```
   Standard2
   Standard1
   Super1

>>> describe_column_family('Keyspace1', 'Indexed1')

Name:                               Indexed1
Description:
Column Type:                        Standard

Comparator Type:                    BytesType
Default Validation Class:           BytesType

Cache Sizes
  Row Cache:                        Disabled
  Key Cache:                        200000 keys

Read Repair Chance:                 100.0%

GC Grace Seconds:                   864000

Compaction Thresholds
  Min:                              4
  Max:                              32

Memtable Flush After Thresholds
  Throughput:                       63 MiB
  Operations:                       295312 operations
  Time:                             60 minutes

Cache Save Periods
  Row Cache:                        Disabled
  Key Cache:                        3600 seconds

Column Metadata
  - Name:                           birthdate
    Value Type:                     LongType
    Index Type:                     KEYS
    Index Name:                     None

>>> SYSTEM_MANAGER.create_keyspace('Keyspace1', strategy_options={"replication_factor": "1"})
>>> SYSTEM_MANAGER.create_column_family('Keyspace1', 'Users', comparator_type=INT_TYPE)
>>> SYSTEM_MANAGER.alter_column_family('Keyspace1', 'Users', key_cache_size=100)
>>> SYSTEM_MANAGER.create_index('Keyspace1', 'Users', 'birthdate', LONG_TYPE, index_name='bday_index'
>>> SYSTEM_MANAGER.drop_keyspace('Keyspace1')
```

5.7 Using pycassa with Other Tools

5.7.1 Using with Celery

Celery is an asynchronous task queue/job queue based on distributed message passing.

Usage in a Worker

Workers in celery may be created by spawning new processes or threads from the celeryd process. The multiprocessing module is used to spawn new worker processes, while eventlet is used to spawn new worker green threads.

`multiprocessing`

The `ConnectionPool` class is not `multiprocessing`-safe. Because celery evaluates globals prior to spawning new worker processes, a global `ConnectionPool` will be shared among multiple processes. This is inherently unsafe and will result in race conditions.

Instead of having celery spawn multiple child processes, it is recommended that you set CELERYD_CONCURRENCY to 1 and start multiple separate celery processes. The process argument `--pool=solo` may also be used when starting the celery processes.

See Also:

Using with multiprocessing

`eventlet`

Because the `ConnectionPool` class uses concurrency primitives from the `threading` module, you can use `eventlet` worker threads after monkey patching the standard library. Specifically, the `threading` and `socket` modules must monkey-patched.

Be aware that you may need to install dnspython in order to connect to your nodes.

See Also:

Using with Eventlet

Usage as a Broker Backend

pycassa is not currently a broker backend option.

5.7.2 Using with Eventlet

Because the `ConnectionPool` class uses concurrency primitives from the `threading` module, you can use `eventlet` green threads after monkey patching the standard library. Specifically, the `threading` and `socket` modules must monkey-patched.

Be aware that you may need to install dnspython in order to connect to your nodes.

5.7.3 Using with `multiprocessing`

The `ConnectionPool` class is not `multiprocessing`-safe. If you're using pycassa with multiprocessing, be sure to create one `ConnectionPool` per process. Creating a `ConnectionPool` before forking and sharing it among processes is inherently unsafe and will result in race conditions.

5.8 Development

5.8.1 New thrift API

pycassa includes Cassandra's Python Thrift API in *pycassa.cassandra*. Since Cassandra 1.1.0, the generated Thrift definitions are fully backwards compatible, allowing you to use attributes that have been deprecated or removed in recent versions of Cassandra. So, even though the code is generated from a Cassandra 1.1.0 definition, you can use the resulting code with 0.7 and still have full access to attributes that were removed after 0.7, such as the memtable flush thresholds.

The following explains the procedure using Mac OS Lion as an example. Other Linux and BSD versions should work in similar ways. Of course you need to have a supported Java JDK installed, the Apple provided JDK is fine. This approach doesn't install any of the tools globally but keeps them isolated. As such we are using *virtualenv <http://pypi.python.org/pypi/virtualenv>*.

First you need some prerequisites, installed via macports or some other package management system:

```
sudo port install boost libevent
```

Download and unpack thrift (http://thrift.apache.org/download/):

```
wget http://apache.osuosl.org/thrift/0.8.0/thrift-0.8.0.tar.gz
tar xzf thrift-0.8.0.tar.gz
```

Create a virtualenv and tell thrift to install into it:

```
cd thrift-0.8.0
virtualenv-2.7 .
export PY_PREFIX=$PWD
```

Configure and build thrift with the minimal functionality we need:

```
./configure --prefix=$PWD --disable-static --with-boost=/opt/local \
    --with-libevent=/opt/local --without-csharp --without-cpp \
    --without-java --without-erlang --without-perl --without-php \
    --without-php_extension --without-ruby --without-haskell --without-go
make
make install
```

You can test the successful install:

```
bin/thrift -version
bin/python -c "from thrift.protocol import fastbinary; print(fastbinary)"
```

Next up is Cassandra. Clone the Git repository:

```
cd ..
git clone http://git-wip-us.apache.org/repos/asf/cassandra.git
cd cassandra
```

We will build the Thrift API for the 1.1.1 release, so checkout the tag (instructions simplified from build.xml *gen-thrift-py*):

```
git checkout cassandra-1.1.1
cd interface
../../thrift-0.8.0/bin/thrift --gen py:new_style -o thrift/ \
    cassandra.thrift
```

We are only interested in the generated Python modules:

```
ls thrift/gen-py/cassandra/*.py
```

These should replace the python files in *pycassa/cassandra*, allowing you to use the latest Thrift methods and object definitions, such as CfDef (which controls what attributes you may set when creating or updating a column family). Don't forget to review the documentation.

Make sure you run the tests, especially if adjusting the default protocol version or introducing backwards incompatible API changes.

5.8.2 References

- http://thrift.apache.org/docs/install/
- http://wiki.apache.org/cassandra/HowToContribute
- http://wiki.apache.org/cassandra/InstallThrift

PYTHON MODULE INDEX

p