starbase Documentation

Release 0.3.1

Artur Barseghyan <artur.barseghyan@gmail.com>

June 19, 2014

Contents

HBase Stargate (REST API) client wrapper for Python.

Read the official documentation of the Stargate.

CHAPTER 1

Description

starbase is (at the moment) a client implementation of the Apache HBase REST API (Stargate).

CHAPTER 2

What you have to know

Beware, that REST API is slow (not to blame on this library!). If you can operate with HBase directly better do so.

CHAPTER 3

Prerequisites

You need to have Hadoop, HBase, Thrift and Stargate running. If you want to make it easy for yourself, read my instructions on installing Cloudera manager (free) on Ubuntu 12.04 LTS here or here.

Once you have everything installed and running (by default Stargate runs on 127.0.0.1:8000), you should be able to run *src/starbase/client/test.py* without problems (UnitTest).

CHAPTER 4

Supported Python versions

- 2.6.8 and up
- 2.7
- 3.3

CHAPTER 5

Features

Project is still in development, thus not all the features of the API are available.

5.1 Features implemented

- Connect to Stargate.
- Show software version.
- Show cluster version.
- Show cluster status.
- List tables.
- Retrieve table schema.
- Retrieve table meta data.
- Get a list of tables' column families.
- Create a table.
- Delete a table.
- Alter table schema.
- Insert (PUT) data into a single row (single or multiple columns).
- Update (POST) data of a single row (single or multiple columns).
- Select (GET) a single row from table, optionally with selected columns only.
- Delete (DELETE) a single row by id.
- Batch insert (PUT).
- Batch update (POST).
- Basic HTTP auth is working. You could provide a login and a password to the connection.
- Retrive all rows in a table (table scanning).

5.2 Features in-development

- Table scanning.
- Syntax globbing.

CHAPTER 6

Installation

Install latest stable version from PyPI.

```
$ pip install starbase
```

Or latest stable version from github.

```
$ pip install -e git+https://github.com/barseghyanartur/starbase@stable#egg=starbase
```

CHAPTER 7

Usage and examples

Operating with API starts with making a connection instance.

7.1 Required imports

```
from starbase import Connection
```

7.2 Create a connection instance

Defaults to 127.0.0.1:8000. Specify `host` and `port` arguments when creating a connection instance, if your settings are different.

```
c = Connection()
```

With customisations, would look simlar to the following.

```
c = Connection(host='192.168.88.22', port=8001)
```

7.3 Show tables

Assuming that there are two existing tables named `table1` and `table2`, the following would be printed out.

```
c.tables()
```

Output.

```
['table1', 'table2']
```

7.4 Operating with table schema

Whenever you need to operate with a table (also, if you need to create one), you need to have a table instance created.

Create a table instance (note, that at this step no table is created).

```
t = c.table('table3')
```

7.4.1 Create a new table

Assuming that no table named `table3` yet exists in the database, create a table named `table3` with columns (column families) `column1`, `column2`, `column3` (this is the point where the table is actually created). In the example below, `column1`, `column2` and `column3` are column families (in short - columns). Columns are declared in the table schema.

```
t.create('column1', 'column2', 'column3')
```

Output.

```
201
```

7.4.2 Check if table exists

```
t.exists()
```

Output.

```
True
```

7.4.3 Show table columns (column families)

```
t.columns()
```

Output.

```
['column1', 'column2', 'column3']
```

7.4.4 Add columns to the table

Add columns given (`column4`, `column5`, `column6`, `column7`).

```
t.add_columns('column4', 'column5', 'column6', 'column7')
```

Output.

```
200
```

7.4.5 Drop columns from table

Drop columns given (`column6`, `column7`).

```
t.drop_columns('column6', 'column7')
```

Output.

```
201
```

7.4.6 Drop entire table schema

```
t.drop()
```

Output.

```
200
```

7.5 Operating with table data

7.5.1 Insert data into a single row

HBase is a key/value store. In HBase columns (also named column families) are part of declared table schema and have to be defined when a table is created. Columns have qualifiers, which are not declared in the table schema. Number of column qualifiers is not limited.

Within a single row, a value is mapped by a column family and a qualifier (in terms of key/value store concept). Value might be anything castable to string (JSON objects, data structures, XML, etc).

In the example below, `key11`, `key12`, `key21`, etc. - are the qualifiers. Obviously, `column1`, `column2` and `column3` are column families.

Column families must be composed of printable characters. Qualifiers can be made of any arbitrary bytes.

Table rows are identified by row keys - unique identifiers (UID or so called primary key). In the example below, `my-key-1` is the row key (UID).

recap all what's said above, HBase maps (row key, column family, column qualifier and timestamp) to a value.

```
t.insert(
    'my-key-1',
    {
        'column1': {'key11': 'value 11', 'key12': 'value 12',
                    'key13': 'value 13'},
        'column2': {'key21': 'value 21', 'key22': 'value 22'},
        'column3': {'key32': 'value 31', 'key32': 'value 32'}
    }
    )
```

Output.

```
200
```

Note, that you may also use the *native* way of naming the columns and cells (qualifiers). Result of the following would be equal to the result of the previous example.

```
t.insert(
    'my-key-1',
    {
        'column1:key11': 'value 11', 'column1:key12': 'value 12',
        'column1:key13': 'value 13',
        'column2:key21': 'value 21', 'column2:key22': 'value 22',
        'column3:key32': 'value 31', 'column3:key32': 'value 32'
    }
    )
```

Output.

```
200
```

7.5.2 Update row data

```
t.update(
    'my-key-1',
    {'column4': {'key41': 'value 41', 'key42': 'value 42'}}
    )
```

Output.

```
200
```

7.5.3 Remove row, row column or row cell data

Remove a row cell (qualifier) data. In the example below, the `my-key-1` is table row UID, `column4` is the column family and the `key41` is the qualifier. Note, that only qualifer data (for the row given) is being removed. All other possible qualifiers of the column `column4` will remain untouched.

```
t.remove('my-key-1', 'column4', 'key41')
```

Output.

```
200
```

Remove a row column (column family) data. Note, that at this point, the entire column data (data of all qualifiers for the row given) is being removed.

```
t.remove('my-key-1', 'column4')
```

Output.

```
200
```

Remove an entire row data. Note, that in this case, entire row data, along with all columns and qualifiers for the row given, is being removed.

```
t.remove('my-key-1')
```

Output.

```
200
```

7.5.4 Fetch table data

Fetch a single row data with all columns and qualifiers.

```
t.fetch('my-key-1')
```

Output.

```
{
    'column1': {'key11': 'value 11', 'key12': 'value 12', 'key13': 'value 13'},
    'column2': {'key21': 'value 21', 'key22': 'value 22'},
    'column3': {'key32': 'value 31', 'key32': 'value 32'}
}
```

Fetch a single row data with selected columns (limit to `column1` and `column2` columns and all their qualifiers).

```
t.fetch('my-key-1', ['column1', 'column2'])
```

Output.

```
{
    'column1': {'key11': 'value 11', 'key12': 'value 12', 'key13': 'value 13'},
    'column2': {'key21': 'value 21', 'key22': 'value 22'},
}
```

Narrow the result set even more (limit to qualifiers `key1` and `key2` of column `column1` and qualifier `key32` of column `column3`).

```
t.fetch('my-key-1', {'column1': ['key11', 'key13'], 'column3': ['key32']})
```

Output.

```
{
    'column1': {'key11': 'value 11', 'key13': 'value 13'},
    'column3': {'key32': 'value 32'}
}
```

Note, that you may also use the *native* way of naming the columns and cells (qualifiers). Example below does exactly the same as example above.

```
t.fetch('my-key-1', ['column1:key11', 'column1:key13', 'column3:key32'])
```

Output.

```
{
    'column1': {'key11': 'value 11', 'key13': 'value 13'},
    'column3': {'key32': 'value 32'}
}
```

If you set the *perfect_dict* argument to False, you'll get the *native* data structure.

```
t.fetch(
    'my-key-1',
    ['column1:key11', 'column1:key13', 'column3:key32'],
    perfect_dict=False
    )
```

Output.

```
{
    'column1:key11': 'value 11',
    'column1:key13': 'value 13',
    'column3:key32': 'value 32'
}
```

7.6 Batch operations with table data

Batch operations (insert and update) work similar to normal insert and update, but are done in a batch. You are advised to operate in batch as much as possible.

7.6.1 Batch insert

In the example below, we will insert 5000 records in a batch.

```
data = {
    'column1': {'key11': 'value 11', 'key12': 'value 12', 'key13': 'value 13'},
    'column2': {'key21': 'value 21', 'key22': 'value 22'},
}
b = t.batch()
if b:
    for i in range(0, 5000):
        b.insert('my-key-%s' % i, data)
    b.commit(finalize=True)
```

Output.

```
{'method': 'PUT', 'response': [200], 'url': 'table3/bXkta2V5LTA='}
```

7.6.2 Batch update

In the example below, we will update 5000 records in a batch.

```
data = {
    'column3': {'key31': 'value 31', 'key32': 'value 32'},
}
b = t.batch()
if b:
    for i in range(0, 5000):
        b.update('my-key-%s' % i, data)
    b.commit(finalize=True)
```

Output.

```
{'method': 'POST', 'response': [200], 'url': 'table3/bXkta2V5LTA='}
```

Note: The table *batch* method accepts an optional *size* argument (int). If set, an auto-commit is fired each the time the stack is `full`.

7.7 Table data search (row scanning)

Table scanning is in development (therefore, the scanning API will likely be changed). Result set returned is a generator.

7.7.1 Fetch all rows

```
t.fetch_all_rows()
```

Output.

```
<generator object results at 0x28e9190>
```

7.7.2 Fetch rows with a filter given

```
rf = '{"type": "RowFilter", "op": "EQUAL", "comparator": {"type": "RegexStringComparator", "value": '
t.fetch_all_rows(with_row_id=True, filter_string=rf)
```

Output.

```
<generator object results at 0x28e9190>
```

CHAPTER 8

More information on table operations

By default, prior further execution of the *fetch*, *insert*, *update*, *remove* (table row operations) methods, it's being checked whether the table exists or not. That's safe, but comes in cost of an extra (light though) HTTP request. If you're absolutely sure you want to avoid those checks, you can disable them. It's possible to disable each type of row operation, by setting the following properties of the table instance to False: `check_if_exists_on_row_fetch`, `check_if_exists_on_row_insert`, `check_if_exists_on_row_remove` and `check_if_exists_on_row_update`.

```
t.check_if_exists_on_row_fetch = False
t.fetch('row1')
```

It's also possible to disable them all at once, by calling the `disable_row_operation_if_exists_checks` method of the table instance.

```
t.disable_row_operation_if_exists_checks()
t.remove('row1')
```

Same goes for table scanner operations. Setting the value of `check_if_exists_on_scanner_operations` of a table instance to False, skips the checks for scanner operations.

```
t.check_if_exists_on_scanner_operations = False
t.fetch_all_rows(flat=True)
```

CHAPTER 9

Exception handling

Methods that accept *fail_silently* argument are listed per class below.

9.1 starbase.client.connection.Connection

- cluster_version
- cluster_status
- drop_table
- tables
- table_exists
- version

9.2 starbase.client.table.Table

- add_columns
- batch
- create
- drop
- drop_columns
- exists
- insert
- fetch
- fetch_all_rows
- regions
- remove
- schema
- update

9.3 starbase.client.table.Batch

- commit
- insert
- update

9.4 starbase.client.transport.HttpRequest

Class *starbase.client.table.Batch* accepts *fail_silently* as a constructor argument.

CHAPTER 10

More examples

10.1 Show software version

```
print connection.version
```

Output.

```
{u'JVM': u'Sun Microsystems Inc. 1.6.0_43-20.14-b01',
 u'Jersey': u'1.8',
 u'OS': u'Linux 3.5.0-30-generic amd64',
 u'REST': u'0.0.2',
 u'Server': u'jetty/6.1.26'}
```

10.2 Show cluster version

```
print connection.cluster_version
```

Output.

```
u'0.94.7'
```

10.3 Show cluster status

```
print connection.cluster_status
```

Output.

```
{u'DeadNodes': [],
 u'LiveNodes': [{u'Region': [{u'currentCompactedKVs': 0,
 ...
 u'regions': 3,
 u'requests': 0}
```

10.4 Show table schema

```
print table.schema()
```

Output.

```
{u'ColumnSchema': [{u'BLOCKCACHE': u'true',
   u'BLOCKSIZE': u'65536',
 ...
   u'IS_ROOT': u'false',
 u'name': u'messages'}
```

10.5 Print table metadata

```
print table.regions()
```

CHAPTER 11

License

GPL 2.0/LGPL 2.1

CHAPTER 12

Support

For any issues contact me at the e-mail given in the *Author* section.

CHAPTER 13

Author

Artur Barseghyan <artur.barseghyan@gmail.com>

CHAPTER 14

Documentation

Contents:

14.1 starbase Package

14.1.1 exceptions Module

exception starbase.exceptions.**BaseException**
Bases: exceptions.Exception

Base exception.

exception starbase.exceptions.**ImproperlyConfigured**
Bases: starbase.exceptions.BaseException

Exception raised when developer didn't configure the code properly.

exception starbase.exceptions.**InvalidArguments**
Bases: exceptions.ValueError, starbase.exceptions.BaseException

Exception raised when invalid arguments supplied.

exception starbase.exceptions.**ParseError**
Bases: starbase.exceptions.BaseException

Raised if the request or response contain malformed data.

exception starbase.exceptions.**DoesNotExist**
Bases: starbase.exceptions.DatabaseError

Does not exist.

exception starbase.exceptions.**DatabaseError**
Bases: starbase.exceptions.BaseException

General database error, as defined in PEP249 interface.

exception starbase.exceptions.**IntegrityError**
Bases: starbase.exceptions.DatabaseError

Integrity error, as defined in PEP249 interface.

14.1.2 Subpackages

client Package

connection Module

class `starbase.client.connection.`**`Connection`**(*host='127.0.0.1', port=8000, user='', password='', secure=False, content_type='json', perfect_dict=True*)

Bases: `object`

Connection instance.

Parameters

- **host** (*str*) – Stargate host.
- **port** (*int*) – Stargate port.
- **user** (*str*) – Stargate user. Use this if your stargate is protected with HTTP basic auth (to be used in combination with *password* argument).
- **password** (*str*) – Stargate password (see comment to *user*).
- **secure** (*bool*) – If set to True, HTTPS is used; otherwise - HTTP. Default value is False.
- **content_type** (*str*) – Content type for data wrapping when communicating with the Stargate. Possible options are: json.
- **perfect_dict** (*bool*) – Global setting. If set to True, generally data will be returned as perfect dict.

`cluster_status`

Storage cluster satus. Returns detailed status on the HBase cluster backing the Stargate instance.

Parameters **fail_silently** (*bool*) –

Return dict Dictionary with information on dead nodes, live nodes, average load, regions, etc.

`cluster_version`

Storage cluster version. Returns version information regarding the HBase cluster backing the Stargate instance.

Parameters **fail_silently** (*bool*) –

Return str HBase version.

`create_table`(*name, *columns*)

Creates the table and returns the instance created. If table already exists, returns None.

Parameters

- **name** (*str*) – Table name.
- ***columns** (*list|tuple*) –

Return starbase.client.table.Table

`drop_table`(*name, fail_silently=True*)

Drops the table.

Parameters **name** (*str*) – Table name.

Return int Status code.

table(*name*)
Initializes a table instance to work with.

Parameters **name** (*str*) – Table name. Example value 'test'.

Return stargate.base.Table

This method does not check if table exists. Use the following methods to perform the check:

- *starbase.client.Connection.table_exists* or
- *starbase.client.table.Table.exists*.

table_exists(*name*, *fail_silently=True*)
Checks if table exists.

Parameters

- **name** (*str*) – Table name.
- **fail_silently** (*bool*) –

Return bool

tables(*raw=False*, *fail_silently=True*)
Table list. Retrieves the list of available tables.

Parameters

- **raw** (*bool*) – If set to True raw result (JSON) is returned.
- **fail_silently** (*bool*) –

Return list Just a list of plain strings of table names, no Table instances.

version
Software version. Returns the software version.

Parameters **fail_silently** (*bool*) –

Return dict Dictionary with info on software versions (OS, Server, JVM, etc).

helpers Module

starbase.client.helpers.**build_json_data**(*row*, *columns*, *timestamp=None*, *encode_content=False*, *with_row_declaration=True*)
Builds JSON data for read-write purposes. Used in *starbase.client.Table._build_table_data*.

Parameters

- **row** (*str*) –
- **columns** (*dict*) –
- **timestamp** – Not yet used.
- **encode_content** (*bool*) –
- **with_row_declaration** (*bool*) –

Return dict

Subpackages

transport Package

transport Package

class `starbase.client.transport.`**`HttpRequest`**(*connection*, *url=''*, *data={}*, *decode_content=False*, *method='GET'*, *fail_silently=True*)

Bases: `object`

HTTP request.

Parameters

- **connection** (*starbase.client.connection.Connection*) –
- **url** (*str*) –
- **data** (*dict*) –
- **decode_content** (*bool*) – If set to True, response content is decoded.
- **method** (*str*) –
- **fail_silently** (*bool*) –

`get_response`()

Return starbase.client.transport.HttpResponse

class `starbase.client.transport.`**`HttpResponse`**(*content*, *raw*)

Bases: `object`

HTTP response.

Parameters

- **content** –
- **raw** (*bool*) –

`get_content`(*decode_content=False*, *keys_to_bypass_decoding=[]*, *keys_to_skip=[]*)

Gets response content.

Parameters

- **decode_content** (*bool*) – If set to True, content is decoded with default decoder, having the empty keys ignored.
- **keys_to_bypass_decoding** (*list|tuple|set*) – List of keys to bypass decoding.
- **keys_to_skip** (*list|tuple|set*) – List of keys to ignore (won't be in the resulted content).

Return str

`status_code`

Gets the HTTP code.

Return str

table Package

table Package

class `starbase.client.table.Table`(*connection*, *name*)

Bases: `object`

For HBase table operations.

Parameters

- **connection** (*stargate.base.Connection*) – Connection instance.
- **name** (*str*) – Table name.

FALSE_ROW_KEY = 'false-row-key'

add_columns(**columns*, ***kwargs*)

Add columns to existing table (POST). If not successful, returns appropriate HTTP error status code. If successful, returns HTTP 200 status.

Parameters

- **name** (*str*) – Table name.
- ***columns** (*list*) – List of columns (plain strings) to ADD.

Return int HTTP response status code (200 on success).

Example

In the example below we create a new table named *table1* with columns *column1* and *column2*. In the next step we add columns *column3* and *columns4* to it.

```
>>> from starbase import Connection
>>> connection = Connection()
>>> table = connection.table('table1')
>>> table.create('column1', 'column2')
>>> table.add_columns('column3', 'column4')
```

batch(*size=None*, *fail_silently=True*)

Returns a Batch instance. Returns None if table does not exist.

Parameters

- **size** (*int*) – Size of auto-commit. If not given, auto-commit is disabled.
- **fail_silently** (*bool*) –

Return starbase.client.table.batch.Batch

Example

Assuming that we have a table named *table1* with columns *column1* and *column2*.

```
>>> from starbase import Connection
>>> connection = Connection()
>>> table = connection.table('table1')
>>> batch = table.batch()
>>> batch.insert('row1', {'column1': {'id': '1', 'name': 'Some name'}, 'column2': {'id': '2'
>>> batch.insert('row2', {'column1': {'id': '12', 'name': 'Some name'}, 'column2': {'id': '2
>>> batch.insert('row3', {'column1': {'id': '13', 'name': 'Some name'}, 'column2': {'id': '2
>>> batch.commit(finalize=True)
```

check_if_exists_on_batch_operations

check_if_exists_on_schema_operations

columns()

Gets a plain list of column families of the table given.

Return list Just a list of plain strings of column family names.

Example

```
>>> from starbase import Connection
>>> connection = Connection()
>>> table = connection.table('table1')
>>> table.columns()
```

create(**columns*, ***kwargs*)

Creates a table schema. If not successful, returns appropriate HTTP error status code. If successful, returns HTTP 201 status.

Parameters ***columns** (*list*) – List of columns (plain strings).

Return int HTTP response status code (201 on success). Returns boolean False on failure.

Example

```
>>> from starbase import Connection
>>> connection = Connection()
>>> table = connection.table('table1')
>>> table.create('column1', 'column2')
```

disable_if_exists_checks()

Skips calling the *exists* method on any operation.

disable_row_operation_if_exists_checks()

Disables *exists* method on row operations.

drop(*fail_silently=True*)

Drops current table. If not successful, returns appropriate HTTP error status code. If successful, returns HTTP 200 status.

Parameters **fail_silently** (*bool*) –

Return int HTTP response status code (200 on success).

Example

In the example below we check if table named *table1* exists and if so - drop it.

```
>>> from starbase import Connection
>>> connection = Connection()
>>> table = connection.table('table1')
>>> if table.exists():
>>>     table.drop()
```

drop_columns(**columns*, ***kwargs*)

Removes/drops columns from table (PUT).If not successful, returns appropriate HTTP error status code. If successful, returns HTTP 201 status.

Parameters

- **name** (*str*) – Table name.
- ***columns** (*list*) – List of columns (plain strings) to REMOVE.

Return int HTTP response status code (201 on success).

Example

Assuming that we have a table named *table1* with columns *column1* and *column2*.

```
>>> from starbase import Connection
>>> connection = Connection()
>>> table = connection.table('table1')
>>> table.drop_columns('column1', 'column2')
```

enable_if_exists_checks()

Enables *exists* method on any operation. The opposite of *disable_if_exists_checks*.

enable_row_operation_if_exists_checks()

Enables *exists* method on row operations. The opposite of *disable_row_operation_if_exists_checks*.

exists(*fail_silently=True*)

Checks if table exists.

Parameters **fail_silently** (*bool*) –

Return **bool**

Example

```
>>> from starbase import Connection
>>> connection = Connection()
>>> table = connection.table('table1')
>>> table.exists()
```

fetch(*row*, *columns=None*, *timestamp=None*, *number_of_versions=None*, *raw=False*, *perfect_dict=None*, *fail_silently=True*)

Fetches a single row from table.

Parameters

- **row** (*str*) –
- **columns** (*list|set|tuple|dict*) –
- **timestamp** – Not yet used.
- **number_of_versions** (*int*) – If provided, multiple versions of the given record are returned.
- **perfect_dict** (*bool*) –
- **raw** (*bool*) –

Return **dict**

Example

In the example below we first create a table named *table1* with columns *column1*, *column2* and *column3*, then insert a row with *column1* and *column2* data, then update the same row with *column3* data and then fetch the data.

```
>>> from starbase import Connection
>>> connection = Connection()
>>> table = connection.table('table1')
>>> table.create('column1', 'column2', 'column3')
>>> table.insert('row1', {'column1': {'id': '1', 'name': 'Some name'}, 'column2': {'id': '2'
>>> table.update('row2', {'column3': {'gender': 'male', 'favourite_book': 'Steppenwolf', 'ac
```

Fetching entire *row1*.

```
>>> table.fetch('row1')
```

Fetching the row *row1* with data from *column1* and *column3* only.

```
>>> table.fetch('row1', ['column1', 'column3'])
```

Fetching the row *row1* with fields *gender* and *favourite_book* from *column3* and fild *age* of column *column2*.

```
>>> table.fetch('row1', {'column3': ['gender', 'favourite_book'], 'column2': ['age']})
```

fetch_all_rows (*with_row_id=False*, *raw=False*, *perfect_dict=None*, *flat=False*, *filter_string=None*, *scanner_config=''*, *fail_silently=True*)

Fetches all table rows.

Parameters

- **with_row_id** (*bool*) – If set to True, returned along with row id.
- **raw** (*bool*) – If set to True, raw response is returned.
- **perfect_dict** (*bool*) – If set to True, a perfect dict struture is used for output data.
- **filter_string** (*string*) – If set, applies the given filter string to the scanner.

Returns list

Example

```
>>> filter_string = '{"type": "RowFilter", "op": "EQUAL", "comparator": '
>>>                 '{"type": "RegexStringComparator", "value": "^row_1.+"}}'
>>> rows = self.table.fetch_all_rows(
>>>     with_row_id = True,
>>>     perfect_dict = perfect_dict,
>>>     filter_string = row_filter_string
>>>     )
```

insert (*row*, *columns*, *timestamp=None*, *fail_silently=True*)

Inserts a single row into a table.

Parameters

- **row** (*str*) –
- **tuple or set) columns** (*(list,*) –
- **timestamp** –

Return int HTTP status code (200 on success).

Example

In the example below we first create a table named *table1* and then insert two rows to it.

```
>>> from starbase import Connection
>>> connection = Connection()
>>> table = connection.table('table1')
>>> table.create('column1', 'column2', 'column3')
>>> table.insert('row1', {'column1': {'id': '1', 'name': 'Some name'}, 'column2': {'id': '2'
>>> table.insert('row2', {'column3': {'gender': 'male', 'favourite_book': 'Steppenwolf'}})
```

metadata (*fail_silently=True*)

Table metadata. Retrieves table region metadata.

Return dict

regions (*fail_silently=True*)
Table metadata. Retrieves table region metadata.

Return dict

remove (*row*, *column=None*, *qualifier=None*, *timestamp=None*, *fail_silently=True*)
Removes/delets a single row/column/qualifier from a table (depending on the depth given). If only row is given, the entire row is deleted. If row and column, only the column value is deleted (entirely for the row given). If qualifier is given as well, then only the qualifier value would be deleted.

Parameters

- **row** (*str*) –
- **column** (*str*) –
- **qualifier** (*str*) –

Return int HTTP status code.

Example

In the example below we first create a table named *table1* with columns *column1*, *column2* and *column3*. Then we insert a single row with multiple columns and then remove parts from that row.

```
>>> from starbase import Connection
>>> connection = Connection()
>>> table = connection.table('table1')
>>> table.create('column1', 'column2', 'column3')
>>> table.insert('row1', {'column1': {'id': '1', 'name': 'Some name'}, 'column2': {'id': '2'
>>> table.remove('row1', 'column2', 'id')
>>> table.remove('row1', 'column1')
>>> table.remove('row1')
```

schema (*fail_silently=True*)
Table schema. Retrieves table schema.

Return dict Dictionary with schema info (detailed information on column families).

Example

```
>>> from starbase import Connection
>>> connection = Connection()
>>> table = connection.table('table1')
>>> table.schema()
```

update (*row*, *columns*, *timestamp=None*, *fail_silently=True*)
Updates a single row in a table.

Parameters

- **row** (*str*) –
- **columns** (*dict*) –
- **timestamp** – Not yet used.

Return int HTTP response status code (200 on success).

Example

In the example below we first create a table named *table1* with columns *column1*, *column2* and *column3*. Then we insert a row with *column1* and *column2* data and then update the same row with *column3* data.

```
>>> from starbase import Connection
>>> connection = Connection()
>>> table = connection.table('table1')
>>> table.create('column1', 'column2', 'column3')
>>> table.insert('row1', {'column1': {'id': '1', 'name': 'Some name'}, 'column2': {'id': '2'
>>> table.update('row1', {'column3': {'gender': 'female', 'favourite_book': 'Solaris'}})
```

batch Module

class `starbase.client.table.batch.`**`Batch`**(*table*, *size=None*)

Bases: `object`

Table batch operations.

Parameters

- **table** (*starbase.client.table.Table*) –
- **size** (*int*) – Batch size. When set, auto commits stacked records when the stack reaches the `size` value.

`commit`(*finalize=False*, *fail_silently=True*)

Sends all queued items to Stargate.

Parameters

- **finalize** (*bool*) – If set to True, the batch is finalized, settings are cleared up and response is returned.
- **fail_silently** (*bool*) –

Return dict If *finalize* set to True, returns the returned value of method meth::*starbase.client.batch.Batch.finalize*.

`finalize`()

Finalize the batch operation. Clear all settings.

Return dict

`insert`(*row*, *columns*, *timestamp=None*, *fail_silently=True*)

`outgoing`()

Returns number of outgoing requests.

Return int

`update`(*row*, *columns*, *timestamp=None*, *encode_content=True*, *fail_silently=True*)

scanner Module

class `starbase.client.table.scanner.`**`Scanner`**(*table*, *url*, *batch_size=None*, *start_row=None*, *end_row=None*, *start_time=None*, *end_time=None*, *data={}*, *extra_headers={}*, *method=None*)

Bases: `object`

Table scanner operations.

`delete`()

Delete scanner.

`results`(*with_row_id=False*, *raw=False*, *perfect_dict=None*)

json_decoder Package

`json_decoder` Package

Recursively decodes values of entire dictionary (JSON) using *base64.decodestring*. Optionally ignores keys given in *keys_to_skip*. It's also possible to give a custom *decoder* instead of *base64.decodestring*.

`starbase.json_decoder.`**`json_decode`**(*json_data, keys_to_bypass_decoding=['timestamp'], keys_to_skip=[], decoder=<function decodestring at 0x7f9e68667578>*)

Recursively decodes values of entire dictionary (JSON) using *base64.decodestring*. Optionally ignores (does not include in the final dictionary) keys given in *keys_to_skip*.

NOTE: Whenever you can, give *decoder* callables, instead of strings (works faster).

NOTE: In HBase stargate *$* keys represent values of the columns.

Parameters

- **json_data** (*dict*) –
- **keys_to_bypass_decoding** (*list|tuple|set*) – Keys to bypass encoding/decoding. Example value ['timestamp',]
- **keys_to_skip** (*list|tuple|set*) – Keys to be excluded from final dict. Example value ['timestamp',]
- **decoder** (*str*) – Example value 'base64.decodestring' or base64.decodestring (assuming that *base64* module has already been imported).

Return dict

Example 1

```
>>> test_json_data = {
>>>     u'Row': [
>>>         {
>>>             u'Cell': [
>>>                 {u'$': u'NDQ=', u'column': u'Y29tcG9uZW50Omlk', u'timestamp': 1369030584274}
>>>                 {u'$': u'MQ==', u'column': u'bWFjaGluZTppZA==', u'timestamp': 1369030584274}
>>>                 {u'$': u'NTUx', u'column': u'c2Vuc29yOmlk', u'timestamp': 1369030584274},
>>>                 {u'$': u'NjQ2', u'column': u'c2Vuc29yOm1lYXN1cmVtZW50', u'timestamp': 136903
>>>                 {u'$': u'VGVtcA==', u'column': u'c2Vuc29yOnR5cGU=', u'timestamp': 1369030584
>>>                 {u'$': u'UGFzY2Fs', u'column': u'c2Vuc29yOnVuaXRfb2ZfbWVhc3VyZQ==', u'timest
>>>             ],
>>>             u'key': u'NDk1MzczYzMtNGVkZi00OWZkLTgwM2YtMDljYjIxYTYyN2Vh'
>>>         }
>>>     ]
>>> }
>>> json_decode(test_json_data, keys_to_skip=['timestamp'])
```

{

u'Row': [

{

u'Cell': [{u'column': 'component:id'}, {u'column': 'machine:id'}, {u'column': 'sensor:id'}, {u'column': 'sensor:measurement'}, {u'column': 'sensor:type'}, {u'column': 'sensor:unit_of_measure'}

], u'key': '495373c3-4edf-49fd-803f-09cb21a627ea'

}

]

}

Example 2

```
>>> # Assuming the 'test_json_data' is the same as in example 1
>>> json_decode(test_json_data, decoder='path.to.your.own.decoder')
```

Example 3

```
>>> # Assuming the 'test_json_data' is the same as in example 1
>>> json_decode(test_json_data)
```

{

u'Row': [

{

u'Cell': [{u'$': '44', u'column': 'component:id'}, {u'$': '1', u'column': 'machine:id'}, {u'$': '551', u'column': 'sensor:id'}, {u'$': '646', u'column': 'sensor:measurement'}, {u'$': 'Temp', u'column': 'sensor:type'}, {u'$': 'Pascal', u'column': 'sensor:unit_of_measure'}

], u'key': '495373c3-4edf-49fd-803f-09cb21a627ea'

}

]

}

CHAPTER 15

Indices and tables

- *genindex*
- *modindex*
- *search*

Python Module Index

S