clickhouse-driver Documentation

Release 0.2.2

clickhouse-driver authors

Sep 24, 2021

Contents

Release 0.2.2.

Welcome to clickhouse-driver's documentation. Get started with *Installation* and then get an overview with the *Quick-start* where common queries are described.

CHAPTER 1

User's Guide

This part of the documentation focuses on step-by-step instructions for development with clickhouse-driver.

Clickhouse-driver is designed to communicate with ClickHouse server from Python over native protocol.

ClickHouse server provides two protocols for communication:

- HTTP protocol (port 8123 by default);
- Native (TCP) protocol (port 9000 by default).

Each protocol has own advantages and disadvantages. Here we focus on advantages of native protocol:

- Native protocol is more configurable by various settings.
- Binary data transfer is more compact than text data.
- Building python types from binary data is more effective than from text data.
- LZ4 compression is faster than gzip. Gzip compression is used in HTTP protocol.
- Query profile info is available over native protocol. We can read rows before limit metric for example.

Once again: clickhouse-driver uses native protocol (port 9000).

There is an asynchronous wrapper for clickhouse-driver: aioch. It's available here.

1.1 Installation

1.1.1 Python Version

Clickhouse-driver supports Python 3.4 and newer and PyPy.

1.1.2 Build Dependencies

Starting from version *0.1.0* for building from source *gcc*, python and linux headers are required.

Example for *python:alpine* docker image:

```
apk add gcc musl-dev
```

By default there are wheels for Linux, Mac OS X and Windows.

Packages for Linux and Mac OS X are available for python: 3.4 – 3.9.

Packages for Windows are available for python: 3.5 – 3.9.

1.1.3 Dependencies

These distributions will be installed automatically when installing clickhouse-driver.

- pytz library for timezone calculations.
- enum34 backported Python 3.4 Enum.

Optional dependencies

These distributions will not be installed automatically. Clickhouse-driver will detect and use them if you install them.

- clickhouse-cityhash provides CityHash algorithm of specific version, see *CityHash algorithm notes*.
- lz4 enables LZ4/LZ4HC compression support.
- zstd enables ZSTD compression support.

1.1.4 Installation from PyPI

The package can be installed using `pip`:

```
pip install clickhouse-driver
```

You can install extras packages if you need compression support. Example of LZ4 compression requirements installation:

```
pip install clickhouse-driver[lz4]
```

You also can specify multiple extras by using comma. Install LZ4 and ZSTD requirements:

```
pip install clickhouse-driver[lz4,zstd]
```

1.1.5 NumPy support

You can install additional packages (NumPy and Pandas) if you need NumPy support:

```
pip install clickhouse-driver[numpy]
```

NumPy supported versions are limited by `numpy` package python support.

1.1.6 Installation from github

Development version can be installed directly from github:

```
pip install git+https://github.com/mymarilyn/clickhouse-driver@master
↪#egg=clickhouse-driver
```

1.2 Quickstart

This page gives a good introduction to clickhouse-driver. It assumes you already have clickhouse-driver installed. If you do not, head over to the *Installation* section.

A minimal working example looks like this:

```
>>> from clickhouse_driver import Client
>>>
>>> client = Client(host='localhost')
>>>
>>> client.execute('SHOW DATABASES')
[('default',)]
```

This code will show all tables from `'default'` database.

There are two conceptual types of queries:

- Read only queries: SELECT, SHOW, etc.
- Read and write queries: INSERT.

Every query should be executed by calling one of the client's execute methods: *execute*, *execute_with_progress*, *execute_iter method*.

- SELECT queries can use *execute*, *execute_with_progress*, *execute_iter* methods.
- INSERT queries can use only *execute* method.

1.2.1 Selecting data

Simple select query looks like:

```
>>> client.execute('SELECT * FROM system.numbers LIMIT 5')
[(0,), (1,), (2,), (3,), (4,)]
```

Of course queries can and should be parameterized to avoid SQL injections:

```
>>> from datetime import date
>>> client.execute(
...     'SELECT %(date)s, %(a)s + %(b)s',
...     {'date': date.today(), 'a': 1, 'b': 2}
... )
[('2018-10-21', 3)]
```

Percent symbols in inlined constants should be doubled if you mix constants with `%` symbol and `%(x)s` parameters.

```
>>> client.execute(
...     "SELECT 'test' like '%%es%%', %(x)s",
...     {'x': 1}
... )
```

Customisation `SELECT` output with `FORMAT` clause is not supported.

1.2.2 Selecting data with progress statistics

You can get query progress statistics by using *execute_with_progress*. It can be useful for cancelling long queries.

```
>>> from datetime import datetime
>>>
>>> progress = client.execute_with_progress(
...     'LONG AND COMPLICATED QUERY'
... )
>>>
>>> timeout = 20
>>> started_at = datetime.now()
>>>
>>> for num_rows, total_rows in progress:
...     if total_rows:
...         done = float(num_rows) / total_rows
...     else:
...         done = total_rows
...
...     now = datetime.now()
...     elapsed = (now - started_at).total_seconds()
...     # Cancel query if it takes more than 20 seconds
...     # to process 50% of rows.
...     if elapsed > timeout and done < 0.5:
...         client.cancel()
...         break
... else:
...     rv = progress.get_result()
...     print(rv)
...
```

1.2.3 Streaming results

When you are dealing with large datasets block by block results streaming may be useful:

```
>>> settings = {'max_block_size': 100000}
>>> rows_gen = client.execute_iter(
...     'QUERY WITH MANY ROWS', settings=settings
... )
>>>
>>> for row in rows_gen:
...     print(row)
...
```

1.2.4 Inserting data

Insert queries in Native protocol are a little bit tricky because of ClickHouse's columnar nature. And because we're using Python.

INSERT query consists of two parts: query statement and query values. Query values are split into chunks called blocks. Each block is sent in binary columnar form.

As data in each block is sent in binary we should not serialize into string by using substitution `%(a)s` and then deserialize it back into Python types.

This INSERT would be extremely slow if executed with thousands rows of data:

```
>>> client.execute(
...     'INSERT INTO test (x) VALUES (%(a)s), (%(b)s), ...',
...     {'a': 1, 'b': 2, ...}
... )
```

To insert data efficiently, provide data separately, and end your statement with a `VALUES` clause:

```
>>> client.execute(
...     'INSERT INTO test (x) VALUES',
...     [{'x': 1}, {'x': 2}, {'x': 3}, {'x': 100}]
... )
4
>>> client.execute(
...     'INSERT INTO test (x) VALUES',
...     [[200]]
... )
1
>>> client.execute(
...     'INSERT INTO test (x) VALUES',
...     ((x, ) for x in range(5))
... )
5
```

You can use any iterable yielding lists, tuples or dicts.

If data is not passed, connection will be terminated after a timeout.

```
>>> client.execute('INSERT INTO test (x) VALUES')  # will hang
```

The following WILL NOT work:

```
>>> client.execute(
...     'INSERT INTO test (x) VALUES (%(a)s), (%(b)s)',
...     {'a': 1, 'b': 2}
... )
```

Of course for `INSERT ... SELECT` queries data is not needed:

```
>>> client.execute(
...     'INSERT INTO test (x) '
...     'SELECT * FROM system.numbers LIMIT %(limit)s',
...     {'limit': 5}
... )
[]
```

ClickHouse will execute this query like a usual `SELECT` query.

Inserting data in different formats with `FORMAT` clause is not supported.

See *Inserting data from CSV file* if you need to data in custom format.

1.2.5 DDL

DDL queries can be executed in the same way SELECT queries are executed:

```
>>> client.execute('DROP TABLE IF EXISTS test')
[]
>>> client.execute('CREATE TABLE test (x Int32) ENGINE = Memory')
[]
```

1.2.6 Async and multithreading

Every ClickHouse query is assigned an identifier to enable request execution tracking. However, ClickHouse native protocol is synchronous: all incoming queries are executed consecutively. Clickhouse-driver does not yet implement a connection pool.

To utilize ClickHouse's asynchronous capability you should either use multiple Client instances or implement a queue.

The same thing is applied to multithreading. Queries from different threads can't use one Client instance with single connection. You should use different clients for different threads.

However, if you are using DB API for communication with the server each cursor create its own Client instance. This makes communication thread-safe.

1.3 Features

- Compression support:
  - LZ4/LZ4HC
  - ZSTD
- TLS support (since server version 1.1.54304).

1.3.1 External data for query processing

You can pass external data alongside with query:

```
>>> tables = [{
...     'name': 'ext',
...     'structure': [('x', 'Int32'), ('y', 'Array(Int32)')],
...     'data': [
...         {'x': 100, 'y': [2, 4, 6, 8]},
...         {'x': 500, 'y': [1, 3, 5, 7]},
...     ]
... }]
>>> client.execute(
...     'SELECT sum(x) FROM ext', external_tables=tables
... )
[(600,)]
```

1.3.2 Settings

There are a lot of ClickHouse server settings. Settings can be specified during Client initialization:

```
# Set max number threads for all queries execution.
>>> settings = {'max_threads': 2}
>>> client = Client('localhost', settings=settings)
```

Each setting can be overridden in an `execute`, `execute_with_progress` and `execute_iter` statement:

```
# Set lower priority to query and limit max number threads
# to execute the request.
>>> settings = {'max_threads': 2, 'priority': 10}
>>> client.execute('SHOW TABLES', settings=settings)
[('first_table',)]
```

1.3.3 Compression

Native protocol supports two types of compression: LZ4 and ZSTD. When compression is enabled compressed data should be hashed using CityHash algorithm. Additional packages should be installed in order by enable compression support, see *Installation from PyPI*. Enabled client-side compression can save network traffic.

Client with compression support can be constructed as follows:

```
>>> from clickhouse_driver import Client
>>> client_with_lz4 = Client('localhost', compression=True)
>>> client_with_lz4 = Client('localhost', compression='lz4')
>>> client_with_zstd = Client('localhost', compression='zstd')
```

CityHash algorithm notes

Unfortunately ClickHouse server comes with built-in old version of CityHash algorithm (1.0.2). That's why we can't use original CityHash package. An older version is published separately at PyPI.

1.3.4 Secure connection

```
>>> from clickhouse_driver import Client
>>>
>>> client = Client('localhost', secure=True)
>>> # Using self-signed certificate.
... self_signed_client = Client(
...     'localhost', secure=True,
...     ca_certs='/etc/clickhouse-server/server.crt'
... )
>>> # Disable verification.
... no_verifyed_client = Client(
...     'localhost', secure=True, verify=False
... )
>>>
>>> # Example of secured client with Let's Encrypt certificate.
... import certifi
>>>
>>> client = Client(
```

(continues on next page)

(continued from previous page)

```
...     'remote-host', secure=True, ca_certs=certifi.where()
... )
```

1.3.5 Specifying query id

You can manually set query identificator for each query. UUID for example:

```
>>> from uuid import uuid4
>>>
>>> query_id = str(uuid4())
>>> print(query_id)
bbd7dea3-eb63-4a21-b727-f55b420a7223
>>> client.execute(
...     'SELECT * FROM system.processes', query_id=query_id
... )
[(1, 'default', 'bbd7dea3-eb63-4a21-b727-f55b420a7223', '127.0.0.1', 57664,
↪'default', 'bbd7dea3-eb63-4a21-b727-f55b420a7223', '127.0.0.1', 57664, 1,
↪'klebedev', 'klebedev-ThinkPad-T460', 'ClickHouse python-driver', 18, 10,␣
↪3, 54406, 0, '', '', 0.004916541, 0, 0, 0, 0, 0, 0, 0, 'SELECT * FROM␣
↪system.processes', (25,), ('Query', 'SelectQuery',
↪'NetworkReceiveElapsedMicroseconds', 'ContextLock',
↪'RWLockAcquiredReadLocks'), (1, 1, 54, 9, 1), ('use_uncompressed_cache',
↪'load_balancing', 'max_memory_usage'), ('0', 'random', '10000000000'))]
```

You can cancel query with specific id by sending another query with the same query id if option replace_running_query is set to 1.

Query results are fetched by the same instance of Client that emitted query.

1.3.6 Retrieving results in columnar form

Columnar form sometimes can be more useful.

```
>>> client.execute('SELECT arrayJoin(range(3))', columnar=True)
[(0, 1, 2)]
```

1.3.7 Data types checking on INSERT

Data types check is disabled for performance on `INSERT` queries. You can turn it on by *types_check* option:

```
>>> client.execute(
...     'INSERT INTO test (x) VALUES', [('abc', )],
...     types_check=True
... )
1
```

1.3.8 Query execution statistics

Client stores statistics about last query execution. It can be obtained by accessing *last_query* attribute. Statistics is sent from ClickHouse server and calculated on client side. *last_query* contains info about:

- profile: rows before limit

```
>>> client.execute('SELECT arrayJoin(range(100)) LIMIT 3')
[(0,), (1,), (2,)]
>>> client.last_query.profile_info.rows_before_limit
100
```

- **progress:**
  - processed rows;
  - processed bytes;
  - total rows;
  - written rows (*new in version 0.1.3*);
  - written bytes (*new in version 0.1.3*);

```
>>> client.execute('SELECT max(number) FROM numbers(10)')
[(9,)]
>>> client.last_query.progress.rows
10
>>> client.last_query.progress.bytes
80
>>> client.last_query.progress.total_rows
10
```

- elapsed time:

```
>>> client.execute('SELECT sleep(1)')
[(0,)]
>>> client.last_query.elapsed
1.0060372352600098
```

1.3.9 Receiving server logs

Query logs can be received from server by using *send_logs_level* setting:

```
>>> from logging.config import dictConfig
>>> # Simple logging configuration.
... dictConfig({
...     'version': 1,
...     'disable_existing_loggers': False,
...     'formatters': {
...         'standard': {
...             'format': '%(asctime)s %(levelname)-8s %(name)s: %(message)s'
...         },
...     },
...     'handlers': {
...         'default': {
...             'level': 'INFO',
...             'formatter': 'standard',
...             'class': 'logging.StreamHandler',
...         },
...     },
...     'loggers': {
...         '': {
```

(continues on next page)

(continued from previous page)

```
...             'handlers': ['default'],
...             'level': 'INFO',
...             'propagate': True
...         },
...     }
... })
>>>
>>> settings = {'send_logs_level': 'debug'}
>>> client.execute('SELECT 1', settings=settings)
2018-12-14 10:24:53,873 INFO     clickhouse_driver.log: [ klebedev-ThinkPad-
↪T460 ] [ 25 ] {b328ad33-60e8-4012-b4cc-97f44a7b28f2} <Debug> executeQuery:␣
↪(from 127.0.0.1:57762) SELECT 1
2018-12-14 10:24:53,874 INFO     clickhouse_driver.log: [ klebedev-ThinkPad-
↪T460 ] [ 25 ] {b328ad33-60e8-4012-b4cc-97f44a7b28f2} <Debug> executeQuery:␣
↪Query pipeline:
Expression
 Expression
  One

2018-12-14 10:24:53,875 INFO     clickhouse_driver.log: [ klebedev-ThinkPad-
↪T460 ] [ 25 ] {b328ad33-60e8-4012-b4cc-97f44a7b28f2} <Information>␣
↪executeQuery: Read 1 rows, 1.00 B in 0.004 sec., 262 rows/sec., 262.32 B/
↪sec.
2018-12-14 10:24:53,875 INFO     clickhouse_driver.log: [ klebedev-ThinkPad-
↪T460 ] [ 25 ] {b328ad33-60e8-4012-b4cc-97f44a7b28f2} <Debug>␣
↪MemoryTracker: Peak memory usage (for query): 40.23 KiB.
[(1,)]
```

1.3.10 Multiple hosts

New in version 0.1.3.

Additional connection points can be defined by using *alt_hosts*. If main connection point is unavailable driver will use next one from *alt_hosts*.

This option is good for ClickHouse cluster with multiple replicas.

```
>>> from clickhouse_driver import Client
>>> client = Client('host1', alt_hosts='host2:1234,host3,host4:5678')
```

In example above on every *new* connection driver will use following sequence of hosts if previous host is unavailable:

- host1:9000;
- host2:1234;
- host3:9000;
- host4:5678.

All queries within established connection will be sent to the same host.

1.3.11 Python DB API 2.0

New in version 0.1.3.

This driver is also implements DB API 2.0 specification. It can be useful for various integrations.

Threads may share the module and connections.

Parameters are expected in Python extended format codes, e.g. ... *WHERE name=%(name)s*.

```
>>> from clickhouse_driver import connect
>>> conn = connect('clickhouse://localhost')
>>> cursor = conn.cursor()
>>>
>>> cursor.execute('SHOW TABLES')
>>> cursor.fetchall()
[('test',)]
>>> cursor.execute('DROP TABLE IF EXISTS test')
>>> cursor.fetchall()
[]
>>> cursor.execute('CREATE TABLE test (x Int32) ENGINE = Memory')
>>> cursor.fetchall()
[]
>>> cursor.executemany(
...     'INSERT INTO test (x) VALUES',
...     [{'x': 100}]
... )
>>> cursor.rowcount
1
>>> cursor.executemany('INSERT INTO test (x) VALUES', [[200]])
>>> cursor.rowcount
1
>>> cursor.execute(
...     'INSERT INTO test (x) '
...     'SELECT * FROM system.numbers LIMIT %(limit)s',
...     {'limit': 3}
... )
>>> cursor.rowcount
0
>>> cursor.execute('SELECT sum(x) FROM test')
>>> cursor.fetchall()
[(303,)]
```

ClickHouse native protocol is synchronous: when you emit query in connection you must read whole server response before sending next query through this connection. To make DB API thread-safe each cursor should use it's own connection to the server. In Under the hood *Cursor* is wrapper around pure *Client*.

Connection class is just wrapper for handling multiple cursors (clients) and do not initiate actual connections to the ClickHouse server.

There are some non-standard ClickHouse-related *Cursor methods* for: external data, settings, etc.

For automatic disposal Connection and Cursor instances can be used as context managers:

```
>>> with connect('clickhouse://localhost') as conn:
>>>     with conn.cursor() as cursor:
>>>         cursor.execute('SHOW TABLES')
>>>         print(cursor.fetchall())
```

1.3.12 NumPy/Pandas support

New in version 0.1.6.

Starting from version 0.1.6 package can SELECT and INSERT columns as NumPy arrays. Additional packages are required for *NumPy support*.

```
>>> client = Client('localhost', settings={'use_numpy': True}):
>>> client.execute(
...     'SELECT * FROM system.numbers LIMIT 10000',
...     columnar=True
... )
[array([   0,    1,    2, ..., 9997, 9998, 9999], dtype=uint64)]
```

Supported types:

- Float32/64
- [U]Int8/16/32/64
- Date/DateTime('timezone')/DateTime64('timezone')
- String/FixedString(N)
- LowCardinality(T)
- Nullable(T)

Direct loading into NumPy arrays increases performance and lowers memory requirements on large amounts of rows.

Direct loading into pandas DataFrame is also supported by using *query_dataframe*:

```
>>> client = Client('localhost', settings={'use_numpy': True})
>>> client.query_dataframe('
...     'SELECT number AS x, (number + 100) AS y '
...     'FROM system.numbers LIMIT 10000'
... )
         x      y
0        0    100
1        1    101
2        2    102
3        3    103
4        4    104
...    ...    ...
9995  9995  10095
9996  9996  10096
9997  9997  10097
9998  9998  10098
9999  9999  10099

[10000 rows x 2 columns]
```

Writing pandas DataFrame is also supported with *insert_dataframe*:

```
>>> client = Client('localhost', settings={'use_numpy': True})
>>> client.execute(
...     'CREATE TABLE test (x Int64, y Int64) Engine = Memory'
... )
>>> []
>>> df = client.query_dataframe(
...     'SELECT number AS x, (number + 100) AS y '
...     'FROM system.numbers LIMIT 10000'
... )
>>> client.insert_dataframe('INSERT INTO test VALUES', df)
>>> 10000
```

Starting from version 0.2.2 nullable columns are also supported. Keep in mind that nullable columns have `object` dtype. For convenience `np.nan` and `None` is supported as `NULL` values for inserting. But only `None` is returned

after selecting for NULL values.

```
>>> client = Client('localhost', settings={'use_numpy': True})
>>> client.execute(
...     'CREATE TABLE test ('
...     'a Nullable(Int64),
...     'b Nullable(Float64),
...     'c Nullable(String)'
...     ') Engine = Memory'
... )
>>> []
>>> df = pd.DataFrame({
...      'a': [1, None, None],
...      'b': [1.0, None, np.nan],
...      'c': ['a', None, np.nan],
... }, dtype=object)
>>> client.insert_dataframe('INSERT INTO test VALUES', df)
3
>>> client.query_dataframe('SELECT * FROM test')
      a     b     c
0     1     1     a
1  None  None  None
2  None   NaN  None
```

It's important to specify *dtype* during dataframe creation:

```
>>> bad_df = pd.DataFrame({
...      'a': [1, None, None],
...      'b': [1.0, None, np.nan],
...      'c': ['a', None, np.nan],
... })
>>> bad_df
     a    b     c
0  1.0  1.0     a
1  NaN  NaN  None
2  NaN  NaN   NaN
>>> good_df = pd.DataFrame({
...      'a': [1, None, None],
...      'b': [1.0, None, np.nan],
...      'c': ['a', None, np.nan],
... }, dtype=object)
>>> good_df
      a     b     c
0     1     1     a
1  None  None  None
2  None   NaN   NaN
```

As you can see float column `b` in `bad_df` has two `NaN` values. But `NaN` and `None` is not the same for float point numbers. `NaN` is `float('nan')` where `None` is representing `NULL`.

1.3.13 Automatic disposal

New in version 0.2.2.

Each Client instance can be used as a context manager:

```
>>> with Client('localhost') as client:
>>>     client.execute('SELECT 1')
```

Upon exit, any established connection to the ClickHouse server will be closed automatically.

1.4 Supported types

Each ClickHouse type is deserialized to a corresponding Python type when SELECT queries are prepared. When serializing INSERT queries, clickhouse-driver accepts a broader range of Python types. The following ClickHouse types are supported by clickhouse-driver:

1.4.1 [U]Int8/16/32/64/128/256

INSERT types: `int`, `long`.

SELECT type: `int`.

1.4.2 Float32/64

INSERT types: `float`, `int`, `long`.

SELECT type: `float`.

1.4.3 Date/Date32

Date32 support is new in version 0.2.2.

INSERT types: `date`, `datetime`.

SELECT type: `date`.

1.4.4 DateTime('timezone')/DateTime64('timezone')

Timezone support is new in version 0.0.11. DateTime64 support is new in version 0.1.3.

INSERT types: `datetime`, `int`, `long`.

Integers are interpreted as seconds without timezone (UNIX timestamps). Integers can be used when insertion of datetime column is a bottleneck.

SELECT type: `datetime`.

Setting use_client_time_zone is taken into consideration.

You can cast DateTime column to integers if you are facing performance issues when selecting large amount of rows.

Due to Python's current limitations minimal DateTime64 resolution is one microsecond.

1.4.5 String/FixedString(N)

INSERT types: `str`/`basestring`, `bytes`. See note below.

SELECT type: `str`/`basestring`, `bytes`. See note below.

String column is encoded/decoded with encoding specified by `strings_encoding` setting. Default encoding is UTF-8.

You can specify custom encoding:

```
>>> settings = {'strings_encoding': 'cp1251'}
>>> rows = client.execute(
...     'SELECT * FROM table_with_strings',
...     settings=settings
... )
```

Encoding is applied to all string fields in query.

String columns can be returned without any decoding. In this case return values are *bytes*:

```
>>> settings = {'strings_as_bytes': True}
>>> rows = client.execute(
...     'SELECT * FROM table_with_strings',
...     settings=settings
... )
```

If a column has FixedString type, upon returning from SELECT it may contain trailing zeroes in accordance with ClickHouse's storage format. Trailing zeroes are stripped by driver for convenience.

During SELECT, if a string cannot be decoded with specified encoding, it will return as `bytes`.

During INSERT, if `strings_as_bytes` setting is not specified and string cannot be encoded with encoding, a `UnicodeEncodeError` will be raised.

1.4.6 Enum8/16

INSERT types: `Enum`, `int`, `long`, `str`/`basestring`.

SELECT type: `str`/`basestring`.

```
>>> from enum import IntEnum
>>>
>>> class MyEnum(IntEnum):
...     foo = 1
...     bar = 2
...
>>> client.execute('DROP TABLE IF EXISTS test')
[]
>>> client.execute('''
...     CREATE TABLE test
...     (
...         x Enum8('foo' = 1, 'bar' = 2)
...     ) ENGINE = Memory
... ''')
[]
>>> client.execute(
...     'INSERT INTO test (x) VALUES',
...     [{'x': MyEnum.foo}, {'x': 'bar'}, {'x': 1}]
```

(continues on next page)

(continued from previous page)

```
... )
3
>>> client.execute('SELECT * FROM test')
[('foo',), ('bar',), ('foo',)]
```

Currently clickhouse-driver can't handle empty enum value due to Python's *Enum* mechanics. Enum member name must be not empty. See issue and workaround.

1.4.7 Array(T)

INSERT types: `list`, `tuple`.

SELECT type: `list`.

Versions before 0.1.4: SELECT type: `tuple`.

```
>>> client.execute('DROP TABLE IF EXISTS test')
[]
>>> client.execute(
...     'CREATE TABLE test (x Array(Int32)) '
...     'ENGINE = Memory'
... )
[]
>>> client.execute(
...     'INSERT INTO test (x) VALUES',
...     [{'x': [10, 20, 30]}, {'x': [11, 21, 31]}]
... )
2
>>> client.execute('SELECT * FROM test')
[((10, 20, 30),), ((11, 21, 31),)]
```

1.4.8 Nullable(T)

INSERT types: `NoneType`, `T`.

SELECT type: `NoneType`, `T`.

1.4.9 UUID

INSERT types: `str`/`basestring`, `UUID`.

SELECT type: `UUID`.

1.4.10 Decimal

New in version 0.0.16.

INSERT types: `Decimal`, `float`, `int`, `long`.

SELECT type: `Decimal`.

Supported subtypes:

- Decimal(P, S).

- Decimal32(S).
- Decimal64(S).
- Decimal128(S).
- Decimal256(S). *New in version 0.2.1.*

1.4.11 IPv4/IPv6

New in version 0.0.19.

INSERT types: `IPv4Address`/`IPv6Address`, `int`, `long`, `str`/`basestring`.

SELECT type: `IPv4Address`/`IPv6Address`.

```
>>> from ipaddress import IPv4Address, IPv6Address
>>>
>>> client.execute('DROP TABLE IF EXISTS test')
[]
>>> client.execute(
...     'CREATE TABLE test (x IPv4) '
...     'ENGINE = Memory'
... )
[]
>>> client.execute(
...     'INSERT INTO test (x) VALUES', [
...     {'x': '192.168.253.42'},
...     {'x': 167772161},
...     {'x': IPv4Address('192.168.253.42')}
... ])
3
>>> client.execute('SELECT * FROM test')
[(IPv4Address('192.168.253.42'),), (IPv4Address('10.0.0.1'),), (IPv4Address(
↪'192.168.253.42'),)]
>>>
>>> client.execute('DROP TABLE IF EXISTS test')
[]
>>> client.execute(
...     'CREATE TABLE test (x IPv6) '
...     'ENGINE = Memory'
... )
[]
>>> client.execute(
...     'INSERT INTO test (x) VALUES', [
...     {'x': '79f4:e698:45de:a59b:2765:28e3:8d3a:35ae'},
...     {'x': IPv6Address('12ff:0000:0000:0000:0000:0000:0000:0001')},
...     {'x': b"y\xf4\xe6\x98E\xde\xa5\x9b'e(\xe3\x8d:5\xae"}
... ])
3
>>> client.execute('SELECT * FROM test')
[(IPv6Address('79f4:e698:45de:a59b:2765:28e3:8d3a:35ae'),), (IPv6Address(
↪'12ff::1'),), (IPv6Address('79f4:e698:45de:a59b:2765:28e3:8d3a:35ae'),)]
>>>
```

1.4.12 LowCardinality(T)

New in version 0.0.20.

INSERT types: `T`.

SELECT type: `T`.

1.4.13 SimpleAggregateFunction(F, T)

New in version 0.0.21.

INSERT types: `T`.

SELECT type: `T`.

AggregateFunctions for *AggregatingMergeTree* Engine are not supported.

1.4.14 Tuple(T1, T2, ...)

New in version 0.1.4.

INSERT types: `list`, `tuple`.

SELECT type: `tuple`.

1.4.15 Nested

Nested type is represented by sequence of arrays. In example below actual columns for are `col.name` and `col.version`.

```
:) CREATE TABLE test_nested (col Nested(name String, version UInt16)) Engine␣
↪= Memory;

CREATE TABLE test_nested
(
    `col` Nested(
    name String,
    version UInt16)
)
ENGINE = Memory

Ok.

0 rows in set. Elapsed: 0.005 sec.

:) DESCRIBE TABLE test_nested FORMAT TSV;

DESCRIBE TABLE test_nested
FORMAT TSV

col.name  Array(String)
col.version       Array(UInt16)

2 rows in set. Elapsed: 0.004 sec.
```

Inserting data into nested column in `clickhouse-client`:

```
:) INSERT INTO test_nested VALUES (['a', 'b', 'c'], [100, 200, 300]);

INSERT INTO test_nested VALUES

Ok.

1 rows in set. Elapsed: 0.003 sec.
```

Inserting data into nested column with `clickhouse-driver`:

```
client.execute('INSERT INTO test_nested VALUES', [
    (['a', 'b', 'c'], [100, 200, 300]),
])
```

1.4.16 Map(key, value)

New in version 0.2.1.

INSERT types: `dict`.

SELECT type: `dict`.

1.5 Performance

This section compares clickhouse-driver performance over Native interface with TSV and JSONEachRow formats available over HTTP interface.

clickhouse-driver returns already parsed row items in Python data types. Driver performs all transformation for you.

When you read data over HTTP you may need to cast strings into Python types.

1.5.1 Test data

Sample data for testing is taken from ClickHouse docs.

Create database and table:

```
DROP DATABASE IF EXISTS perftest;

CREATE DATABASE perftest;

CREATE TABLE perftest.ontime (
  Year UInt16,
  Quarter UInt8,
  Month UInt8,
  DayofMonth UInt8,
  DayOfWeek UInt8,
  FlightDate Date,
  UniqueCarrier FixedString(7),
  AirlineID Int32,
  Carrier FixedString(2),
  TailNum String,
  FlightNum String,
  OriginAirportID Int32,
```

(continues on next page)

(continued from previous page)

```
    OriginAirportSeqID Int32,
    OriginCityMarketID Int32,
    Origin FixedString(5),
    OriginCityName String,
    OriginState FixedString(2),
    OriginStateFips String,
    OriginStateName String,
    OriginWac Int32,
    DestAirportID Int32,
    DestAirportSeqID Int32,
    DestCityMarketID Int32,
    Dest FixedString(5),
    DestCityName String,
    DestState FixedString(2),
    DestStateFips String,
    DestStateName String,
    DestWac Int32,
    CRSDepTime Int32,
    DepTime Int32,
    DepDelay Int32,
    DepDelayMinutes Int32,
    DepDel15 Int32,
    DepartureDelayGroups String,
    DepTimeBlk String,
    TaxiOut Int32,
    WheelsOff Int32,
    WheelsOn Int32,
    TaxiIn Int32,
    CRSArrTime Int32,
    ArrTime Int32,
    ArrDelay Int32,
    ArrDelayMinutes Int32,
    ArrDel15 Int32,
    ArrivalDelayGroups Int32,
    ArrTimeBlk String,
    Cancelled UInt8,
    CancellationCode FixedString(1),
    Diverted UInt8,
    CRSElapsedTime Int32,
    ActualElapsedTime Int32,
    AirTime Int32,
    Flights Int32,
    Distance Int32,
    DistanceGroup UInt8,
    CarrierDelay Int32,
    WeatherDelay Int32,
    NASDelay Int32,
    SecurityDelay Int32,
    LateAircraftDelay Int32,
    FirstDepTime String,
    TotalAddGTime String,
    LongestAddGTime String,
    DivAirportLandings String,
    DivReachedDest String,
    DivActualElapsedTime String,
    DivArrDelay String,
    DivDistance String,
```

(continues on next page)

(continued from previous page)

```
  Div1Airport String,
  Div1AirportID Int32,
  Div1AirportSeqID Int32,
  Div1WheelsOn String,
  Div1TotalGTime String,
  Div1LongestGTime String,
  Div1WheelsOff String,
  Div1TailNum String,
  Div2Airport String,
  Div2AirportID Int32,
  Div2AirportSeqID Int32,
  Div2WheelsOn String,
  Div2TotalGTime String,
  Div2LongestGTime String,
  Div2WheelsOff String,
  Div2TailNum String,
  Div3Airport String,
  Div3AirportID Int32,
  Div3AirportSeqID Int32,
  Div3WheelsOn String,
  Div3TotalGTime String,
  Div3LongestGTime String,
  Div3WheelsOff String,
  Div3TailNum String,
  Div4Airport String,
  Div4AirportID Int32,
  Div4AirportSeqID Int32,
  Div4WheelsOn String,
  Div4TotalGTime String,
  Div4LongestGTime String,
  Div4WheelsOff String,
  Div4TailNum String,
  Div5Airport String,
  Div5AirportID Int32,
  Div5AirportSeqID Int32,
  Div5WheelsOn String,
  Div5TotalGTime String,
  Div5LongestGTime String,
  Div5WheelsOff String,
  Div5TailNum String
) ENGINE = MergeTree
PARTITION BY Year
ORDER BY (Carrier, FlightDate)
SETTINGS index_granularity = 8192;
```

Download some data for 2017 year:

```
for s in `seq 2017 2017`
do
for m in `seq 1 12`
do
wget https://transtats.bts.gov/PREZIP/On_Time_Reporting_Carrier_On_Time_Performance_
↪1987_present_${s}_${m}.zip
done
done
```

Insert data into ClickHouse:

```
for i in *.zip; do echo $i; unzip -cq $i '*.csv' | sed 's/\.00//g' | clickhouse-
↪client --query="INSERT INTO perftest.ontime FORMAT CSVWithNames"; done
```

1.5.2 Required packages

```
pip install clickhouse-driver requests
```

For fast json parsing we'll use `ujson` package:

```
pip install ujson
```

Installed packages:

```
$ pip freeze
certifi==2020.4.5.1
chardet==3.0.4
clickhouse-driver==0.1.3
idna==2.9
pytz==2019.3
requests==2.23.0
tzlocal==2.0.0
ujson==2.0.3
urllib3==1.25.9
```

1.5.3 Versions

Machine: Linux ThinkPad-T460 4.4.0-177-generic #207-Ubuntu SMP Mon Mar 16 01:16:10 UTC 2020 x86_64 x86_64 x86_64 GNU/Linux

Python: CPython 3.6.5 (default, May 30 2019, 14:48:31) [GCC 5.4.0 20160609]

1.5.4 Benchmarking

Let's pick number of rows for testing with `clickhouse-client`.

```
SELECT count() FROM ontime WHERE FlightDate < '2017-01-04'

45202
```

```
SELECT count() FROM ontime WHERE FlightDate < '2017-01-10'

131848
```

```
SELECT count() FROM ontime WHERE FlightDate < '2017-01-16'

217015
```

```
SELECT count() FROM ontime WHERE FlightDate < '2017-02-01'

450017
```

```
SELECT count() FROM ontime WHERE FlightDate < '2017-02-18'

697813
```

Scripts below can be benchmarked with following one-liner:

```
for d in 2017-01-04 2017-01-10 2017-01-16 2017-02-01 2017-02-18; do /usr/bin/time -f "
↪%e s / %M kB" python script.py $d; done
```

Time will measure:

- elapsed real (wall clock) time used by the process, in seconds;
- maximum resident set size of the process during its lifetime, in kilobytes.

Plain text without parsing

Let's take get plain text response from ClickHouse server as baseline.

Fetching not parsed data with pure requests (1)

```
import sys
import requests

query = "SELECT * FROM perftest.ontime WHERE FlightDate < '{}' FORMAT {}".format(sys.
↪argv[1], sys.argv[2])
data = requests.get('http://localhost:8123/', params={'query': query})
```

Parsed rows

Line split into elements will be consider as "parsed" for TSV format (2)

```
import sys
import requests

query = "SELECT * FROM perftest.ontime WHERE FlightDate < '{}' FORMAT TSV".format(sys.
↪argv[1])
resp = requests.get('http://localhost:8123/', stream=True, params={'query': query})

data = [line.decode('utf-8').split('\t') for line in resp.iter_lines(chunk_
↪size=10000)]
```

Now we cast each element to it's data type (2.5)

```
from datetime import date
import sys
import requests


def get_python_type(ch_type):
  if ch_type.startswith('Int') or ch_type.startswith('UInt'):
    return int

  elif ch_type == 'String' or ch_type.startswith('FixedString'):
    return None
```

(continues on next page)

(continued from previous page)

```
  elif ch_type == 'Date':
    return lambda value: date(*[int(x) for x  in value.split('-')])

  raise ValueError(f'Unsupported type: "{ch_type}"')


resp = requests.get('http://localhost:8123', params={'query': 'describe table␣
↪perftest.ontime FORMAT TSV'})
ch_types = [x.split('\t')[1] for x in resp.text.split('\n') if x]
python_types = [get_python_type(x) for x in ch_types]

query = "SELECT * FROM perftest.ontime WHERE FlightDate < '{}' FORMAT TSV".format(sys.
↪argv[1])
resp = requests.get('http://localhost:8123/', stream=True, params={'query': query})

data = []

for line in resp.iter_lines(chunk_size=10000):
   data.append([cls(x) if cls else x for x, cls in zip(line.decode('utf-8').split('\t
↪'), python_types)])
```

JSONEachRow format can be loaded with json loads (3)

```
import sys
import requests
from ujson import loads

query = "SELECT * FROM perftest.ontime WHERE FlightDate < '{}' FORMAT JSONEachRow".
↪format(sys.argv[1])
resp = requests.get('http://localhost:8123/', stream=True, params={'query': query})

data = [list(loads(line).values()) for line in resp.iter_lines(chunk_size=10000)]
```

Get fully parsed rows with `clickhouse-driver` in Native format (4)

```
import sys
from clickhouse_driver import Client

query = "SELECT * FROM perftest.ontime WHERE FlightDate < '{}'".format(sys.argv[1])
client = Client.from_url('clickhouse://localhost')

data = client.execute(query)
```

Iteration over rows

Iteration over TSV (5)

```
import sys
import requests

query = "SELECT * FROM perftest.ontime WHERE FlightDate < '{}' FORMAT TSV".format(sys.
↪argv[1])
resp = requests.get('http://localhost:8123/', stream=True, params={'query': query})
```

(continues on next page)

(continued from previous page)

```
for line in resp.iter_lines(chunk_size=10000):
  line = line.decode('utf-8').split('\t')
```

Now we cast each element to it's data type (5.5)

```
from datetime import date
import sys
import requests


def get_python_type(ch_type):
  if ch_type.startswith('Int') or ch_type.startswith('UInt'):
    return int

  elif ch_type == 'String' or ch_type.startswith('FixedString'):
    return None

  elif ch_type == 'Date':
    return lambda value: date(*[int(x) for x  in value.split('-')])

  raise ValueError(f'Unsupported type: "{ch_type}"')


resp = requests.get('http://localhost:8123', params={'query': 'describe table␣
↪perftest.ontime FORMAT TSV'})
ch_types = [x.split('\t')[1] for x in resp.text.split('\n') if x]
python_types = [get_python_type(x) for x in ch_types]

query = "SELECT * FROM perftest.ontime WHERE FlightDate < '{}' FORMAT TSV".format(sys.
↪argv[1])
resp = requests.get('http://localhost:8123/', stream=True, params={'query': query})

for line in resp.iter_lines(chunk_size=10000):
   line = [cls(x) if cls else x for x, cls in zip(line.decode('utf-8').split('\t'),␣
↪python_types)]
```

Iteration over JSONEachRow (6)

```
import sys
import requests
from ujson import loads

query = "SELECT * FROM perftest.ontime WHERE FlightDate < '{}' FORMAT JSONEachRow".
↪format(sys.argv[1])
resp = requests.get('http://localhost:8123/', stream=True, params={'query': query})

for line in resp.iter_lines(chunk_size=10000):
  line = list(loads(line).values())
```

Iteration over rows with `clickhouse-driver` in Native format (7)

```
import sys
from clickhouse_driver import Client

query = "SELECT * FROM perftest.ontime WHERE FlightDate < '{}'".format(sys.argv[1])
client = Client.from_url('clickhouse://localhost')
```

(continues on next page)

(continued from previous page)

```
for row in client.execute_iter(query):
  pass
```

Iteration over string rows

OK, but what if we need only string columns?

Iteration over TSV (8)

```
import sys
import requests

cols = [
    'UniqueCarrier', 'Carrier', 'TailNum', 'FlightNum', 'Origin', 'OriginCityName',
↪'OriginState',
    'OriginStateFips', 'OriginStateName', 'Dest', 'DestCityName', 'DestState',
↪'DestStateFips',
    'DestStateName', 'DepartureDelayGroups', 'DepTimeBlk', 'ArrTimeBlk',
↪'CancellationCode',
    'FirstDepTime', 'TotalAddGTime', 'LongestAddGTime', 'DivAirportLandings',
↪'DivReachedDest',
    'DivActualElapsedTime', 'DivArrDelay', 'DivDistance', 'Div1Airport', 'Div1WheelsOn
↪', 'Div1TotalGTime',
    'Div1LongestGTime', 'Div1WheelsOff', 'Div1TailNum', 'Div2Airport', 'Div2WheelsOn',
↪ 'Div2TotalGTime',
    'Div2LongestGTime', 'Div2WheelsOff', 'Div2TailNum', 'Div3Airport', 'Div3WheelsOn',
↪ 'Div3TotalGTime',
    'Div3LongestGTime', 'Div3WheelsOff', 'Div3TailNum', 'Div4Airport', 'Div4WheelsOn',
↪ 'Div4TotalGTime',
    'Div4LongestGTime', 'Div4WheelsOff', 'Div4TailNum', 'Div5Airport', 'Div5WheelsOn',
↪ 'Div5TotalGTime',
    'Div5LongestGTime', 'Div5WheelsOff', 'Div5TailNum'
]

query = "SELECT {} FROM perftest.ontime WHERE FlightDate < '{}' FORMAT TSV".format(',
↪'.join(cols), sys.argv[1])
resp = requests.get('http://localhost:8123/', stream=True, params={'query': query})

for line in resp.iter_lines(chunk_size=10000):
  line = line.decode('utf-8').split('\t')
```

Iteration over JSONEachRow (9)

```
import sys
import requests
from ujson import loads

cols = [
    'UniqueCarrier', 'Carrier', 'TailNum', 'FlightNum', 'Origin', 'OriginCityName',
↪'OriginState',
    'OriginStateFips', 'OriginStateName', 'Dest', 'DestCityName', 'DestState',
↪'DestStateFips',
    'DestStateName', 'DepartureDelayGroups', 'DepTimeBlk', 'ArrTimeBlk',
↪'CancellationCode',
    'FirstDepTime', 'TotalAddGTime', 'LongestAddGTime', 'DivAirportLandings',
↪'DivReachedDest',
```

(continues on next page)

(continued from previous page)

```
    'DivActualElapsedTime', 'DivArrDelay', 'DivDistance', 'Div1Airport', 'Div1WheelsOn
↪', 'Div1TotalGTime',
    'Div1LongestGTime', 'Div1WheelsOff', 'Div1TailNum', 'Div2Airport', 'Div2WheelsOn',
↪ 'Div2TotalGTime',
    'Div2LongestGTime', 'Div2WheelsOff', 'Div2TailNum', 'Div3Airport', 'Div3WheelsOn',
↪ 'Div3TotalGTime',
    'Div3LongestGTime', 'Div3WheelsOff', 'Div3TailNum', 'Div4Airport', 'Div4WheelsOn',
↪ 'Div4TotalGTime',
    'Div4LongestGTime', 'Div4WheelsOff', 'Div4TailNum', 'Div5Airport', 'Div5WheelsOn',
↪ 'Div5TotalGTime',
    'Div5LongestGTime', 'Div5WheelsOff', 'Div5TailNum'
]

query = "SELECT {} FROM perftest.ontime WHERE FlightDate < '{}' FORMAT JSONEachRow".
↪format(', '.join(cols), sys.argv[1])
resp = requests.get('http://localhost:8123/', stream=True, params={'query': query})

for line in resp.iter_lines(chunk_size=10000):
  line = list(loads(line).values())
```

Iteration over string rows with `clickhouse-driver` in Native format (10)

```
import sys
from clickhouse_driver import Client

cols = [
    'UniqueCarrier', 'Carrier', 'TailNum', 'FlightNum', 'Origin', 'OriginCityName',
↪'OriginState',
    'OriginStateFips', 'OriginStateName', 'Dest', 'DestCityName', 'DestState',
↪'DestStateFips',
    'DestStateName', 'DepartureDelayGroups', 'DepTimeBlk', 'ArrTimeBlk',
↪'CancellationCode',
    'FirstDepTime', 'TotalAddGTime', 'LongestAddGTime', 'DivAirportLandings',
↪'DivReachedDest',
    'DivActualElapsedTime', 'DivArrDelay', 'DivDistance', 'Div1Airport', 'Div1WheelsOn
↪', 'Div1TotalGTime',
    'Div1LongestGTime', 'Div1WheelsOff', 'Div1TailNum', 'Div2Airport', 'Div2WheelsOn',
↪ 'Div2TotalGTime',
    'Div2LongestGTime', 'Div2WheelsOff', 'Div2TailNum', 'Div3Airport', 'Div3WheelsOn',
↪ 'Div3TotalGTime',
    'Div3LongestGTime', 'Div3WheelsOff', 'Div3TailNum', 'Div4Airport', 'Div4WheelsOn',
↪ 'Div4TotalGTime',
    'Div4LongestGTime', 'Div4WheelsOff', 'Div4TailNum', 'Div5Airport', 'Div5WheelsOn',
↪ 'Div5TotalGTime',
    'Div5LongestGTime', 'Div5WheelsOff', 'Div5TailNum'
]

query = "SELECT {} FROM perftest.ontime WHERE FlightDate < '{}'".format(', '.
↪join(cols), sys.argv[1])
client = Client.from_url('clickhouse://localhost')

for row in client.execute_iter(query):
  pass
```

Iteration over int rows

Iteration over TSV (11)

```
import sys
import requests

cols = [
    'Year', 'Quarter', 'Month', 'DayofMonth', 'DayOfWeek', 'AirlineID',
↪'OriginAirportID', 'OriginAirportSeqID',
    'OriginCityMarketID', 'OriginWac', 'DestAirportID', 'DestAirportSeqID',
↪'DestCityMarketID', 'DestWac',
    'CRSDepTime', 'DepTime', 'DepDelay', 'DepDelayMinutes', 'DepDel15', 'TaxiOut',
↪'WheelsOff', 'WheelsOn',
    'TaxiIn', 'CRSArrTime', 'ArrTime', 'ArrDelay', 'ArrDelayMinutes', 'ArrDel15',
↪'ArrivalDelayGroups',
    'Cancelled', 'Diverted', 'CRSElapsedTime', 'ActualElapsedTime', 'AirTime',
↪'Flights', 'Distance',
    'DistanceGroup', 'CarrierDelay', 'WeatherDelay', 'NASDelay', 'SecurityDelay',
↪'LateAircraftDelay',
    'Div1AirportID', 'Div1AirportSeqID', 'Div2AirportID', 'Div2AirportSeqID',
↪'Div3AirportID',
    'Div3AirportSeqID', 'Div4AirportID', 'Div4AirportSeqID', 'Div5AirportID',
↪'Div5AirportSeqID'
]

query = "SELECT {} FROM perftest.ontime WHERE FlightDate < '{}' FORMAT TSV".format(',
↪'.join(cols), sys.argv[1])
resp = requests.get('http://localhost:8123/', stream=True, params={'query': query})

for line in resp.iter_lines(chunk_size=10000):
  line = [int(x) for x in line.split(b'\t')]
```

Iteration over JSONEachRow (12)

```
import sys
import requests
from ujson import loads

cols = [
    'Year', 'Quarter', 'Month', 'DayofMonth', 'DayOfWeek', 'AirlineID',
↪'OriginAirportID', 'OriginAirportSeqID',
    'OriginCityMarketID', 'OriginWac', 'DestAirportID', 'DestAirportSeqID',
↪'DestCityMarketID', 'DestWac',
    'CRSDepTime', 'DepTime', 'DepDelay', 'DepDelayMinutes', 'DepDel15', 'TaxiOut',
↪'WheelsOff', 'WheelsOn',
    'TaxiIn', 'CRSArrTime', 'ArrTime', 'ArrDelay', 'ArrDelayMinutes', 'ArrDel15',
↪'ArrivalDelayGroups',
    'Cancelled', 'Diverted', 'CRSElapsedTime', 'ActualElapsedTime', 'AirTime',
↪'Flights', 'Distance',
    'DistanceGroup', 'CarrierDelay', 'WeatherDelay', 'NASDelay', 'SecurityDelay',
↪'LateAircraftDelay',
    'Div1AirportID', 'Div1AirportSeqID', 'Div2AirportID', 'Div2AirportSeqID',
↪'Div3AirportID',
    'Div3AirportSeqID', 'Div4AirportID', 'Div4AirportSeqID', 'Div5AirportID',
↪'Div5AirportSeqID'
]
```

(continues on next page)

(continued from previous page)

```
query = "SELECT {} FROM perftest.ontime WHERE FlightDate < '{}' FORMAT JSONEachRow".
↪format(', '.join(cols), sys.argv[1])
resp = requests.get('http://localhost:8123/', stream=True, params={'query': query})

for line in resp.iter_lines(chunk_size=10000):
  line = list(loads(line).values())
```

Iteration over int rows with `clickhouse-driver` in Native format (13)

```
import sys
from clickhouse_driver import Client

cols = [
    'Year', 'Quarter', 'Month', 'DayofMonth', 'DayOfWeek', 'AirlineID',
↪'OriginAirportID', 'OriginAirportSeqID',
    'OriginCityMarketID', 'OriginWac', 'DestAirportID', 'DestAirportSeqID',
↪'DestCityMarketID', 'DestWac',
    'CRSDepTime', 'DepTime', 'DepDelay', 'DepDelayMinutes', 'DepDel15', 'TaxiOut',
↪'WheelsOff', 'WheelsOn',
    'TaxiIn', 'CRSArrTime', 'ArrTime', 'ArrDelay', 'ArrDelayMinutes', 'ArrDel15',
↪'ArrivalDelayGroups',
    'Cancelled', 'Diverted', 'CRSElapsedTime', 'ActualElapsedTime', 'AirTime',
↪'Flights', 'Distance',
    'DistanceGroup', 'CarrierDelay', 'WeatherDelay', 'NASDelay', 'SecurityDelay',
↪'LateAircraftDelay',
    'Div1AirportID', 'Div1AirportSeqID', 'Div2AirportID', 'Div2AirportSeqID',
↪'Div3AirportID',
    'Div3AirportSeqID', 'Div4AirportID', 'Div4AirportSeqID', 'Div5AirportID',
↪'Div5AirportSeqID'
]

query = "SELECT {} FROM perftest.ontime WHERE FlightDate < '{}'".format(', '.
↪join(cols), sys.argv[1])
client = Client.from_url('clickhouse://localhost')

for row in client.execute_iter(query):
  pass
```

1.5.5 Results

This table contains memory and timing benchmark results of snippets above.

JSON in table is shorthand for JSONEachRow.

	Rows				
	50k	131k	217k	450k	697k
Plain text without parsing: timing					
Naive requests.get TSV (1)	0.40 s	0.67 s	0.95 s	1.67 s	2.52
Naive requests.get JSON (1)	0.61 s	1.23 s	2.09 s	3.52 s	5.20
Plain text without parsing: memory					
Naive requests.get TSV (1)	49 MB	107 MB	165 MB	322 MB	488
Naive requests.get JSON (1)	206 MB	564 MB	916 MB	1.83 GB	2.83
Parsed rows: timing					

Continued on next pa

Table 1 – continued from previous page

	Rows				
	50k	131k	217k	450k	697k
requests.get TSV (2)	0.81 s	1.81 s	3.09 s	7.22 s	11.8
requests.get TSV with cast (2.5)	1.78 s	4.58 s	7.42 s	16.12 s	25.5
requests.get JSON (3)	2.14 s	5.65 s	9.20 s	20.43 s	31.7
clickhouse-driver Native (4)	0.73 s	1.40 s	2.08 s	4.03 s	6.20
Parsed rows: memory					
requests.get TSV (2)	171 MB	462 MB	753 MB	1.51 GB	2.33
requests.get TSV with cast (2.5)	135 MB	356 MB	576 MB	1.15 GB	1.78
requests.get JSON (3)	139 MB	366 MB	591 MB	1.18 GB	1.82
clickhouse-driver Native (4)	135 MB	337 MB	535 MB	1.05 GB	1.62
Iteration over rows: timing					
requests.get TSV (5)	0.49 s	0.99 s	1.34 s	2.58 s	4.00
requests.get TSV with cast (5.5)	1.38 s	3.38 s	5.40 s	10.89 s	16.5
requests.get JSON (6)	1.89 s	4.73 s	7.63 s	15.63 s	24.6
clickhouse-driver Native (7)	0.62 s	1.28 s	1.93 s	3.68 s	5.54
Iteration over rows: memory					
requests.get TSV (5)	19 MB	19 MB	19 MB	19 MB	19 M
requests.get TSV with cast (5.5)	19 MB	19 MB	19 MB	19 MB	19 M
requests.get JSON (6)	20 MB	20 MB	20 MB	20 MB	20 M
clickhouse-driver Native (7)	56 MB	70 MB	71 MB	71 MB	71 M
Iteration over string rows: timing					
requests.get TSV (8)	0.40 s	0.67 s	0.80 s	1.55 s	2.18
requests.get JSON (9)	1.14 s	2.64 s	4.22 s	8.48 s	12.9
clickhouse-driver Native (10)	0.46 s	0.91 s	1.35 s	2.49 s	3.67
Iteration over string rows: memory					
requests.get TSV (8)	19 MB	19 MB	19 MB	19 MB	19 M
requests.get JSON (9)	20 MB	20 MB	20 MB	20 MB	20 M
clickhouse-driver Native (10)	46 MB	56 MB	57 MB	57 MB	57 M
Iteration over int rows: timing					
requests.get TSV (11)	0.84 s	2.06 s	3.22 s	6.27 s	10.0
requests.get JSON (12)	0.95 s	2.15 s	3.55 s	6.93 s	10.8
clickhouse-driver Native (13)	0.43 s	0.61 s	0.86 s	1.53 s	2.27
Iteration over int rows: memory					
requests.get TSV (11)	19 MB	19 MB	19 MB	19 MB	19 M
requests.get JSON (12)	20 MB	20 MB	20 MB	20 MB	20 M
clickhouse-driver Native (13)	41 MB	48 MB	48 MB	48 MB	49 M

1.5.6 Conclusion

If you need to get significant number of rows from ClickHouse server **as text** then TSV format is your choice. See **Iteration over string rows** results.

But if you need to manipulate over python data types then you should take a look on drivers with Native format. For most data types driver uses binary `pack()` / `unpack()` for serialization / deserialization. Which is obviously faster than `cls() for x in lst`. See (2.5) and (5.5).

It doesn't matter which interface to use if you manipulate small amount of rows.

1.6 Miscellaneous

1.6.1 Client configuring from URL

New in version 0.1.1.

Client can be configured from the given URL:

```
>>> from clickhouse_driver import Client
>>> client = Client.from_url(
...     'clickhouse://login:password@host:port/database'
... )
```

Port 9000 is default for schema `clickhouse`, port 9440 is default for schema `clickhouses`.

Connection to default database:

```
>>> client = Client.from_url('clickhouse://localhost')
```

Querystring arguments will be passed along to the *`Connection()`* class's initializer:

```
>>> client = Client.from_url(
...     'clickhouse://localhost/database?send_logs_level=trace&'
...     'client_name=myclient&'
...     'compression=lz4'
... )
```

If parameter doesn't match Connection's init signature will be treated as settings parameter.

1.6.2 Inserting data from CSV file

Let's assume you have following data in CSV file.

```
$ cat /tmp/data.csv
time,order,qty
2019-08-01 15:23:14,New order1,5
2019-08-05 09:14:45,New order2,3
2019-08-13 12:20:32,New order3,7
```

Data can be inserted into ClickHouse in the following way:

```
>>> from csv import DictReader
>>> from datetime import datetime
>>>
>>> from clickhouse_driver import Client
>>>
>>>
>>> def iter_csv(filename):
...     converters = {
...         'qty': int,
...         'time': lambda x: datetime.strptime(x, '%Y-%m-%d %H:%M:%S')
...     }
...
...     with open(filename, 'r') as f:
...         reader = DictReader(f)
...         for line in reader:
```

(continues on next page)

(continued from previous page)

```
...             yield {k: (converters[k](v) if k in converters else v) for k,
↪ v in line.items()}
...
>>> client = Client('localhost')
>>>
>>> client.execute(
...     'CREATE TABLE IF NOT EXISTS data_csv '
...     '('
...         'time DateTime, '
...         'order String, '
...         'qty Int32'
...     ') Engine = Memory'
... )
>>> []
>>> client.execute('INSERT INTO data_csv VALUES', iter_csv('/tmp/data.csv'))
3
```

Table can be populated with json file in the similar way.

1.6.3 Adding missed settings

It's hard to keep package settings in consistent state with ClickHouse server's. Some settings can be missed if your server is old. But, if setting is *supported by your server* and missed in the package it can be added by simple monkey pathing. Just look into ClickHouse server source and pick corresponding setting type from package or write your own type.

```
>>> from clickhouse_driver.settings.available import settings as available_
↪settings, SettingBool
>>> from clickhouse_driver import Client
>>>
>>> available_settings['allow_suspicious_low_cardinality_types'] =␣
↪SettingBool
>>>
>>> client = Client('localhost', settings={'allow_suspicious_low_cardinality_
↪types': True})
>>> client.execute('CREATE TABLE test (x LowCardinality(Int32)) Engine = Null
↪')
[]
```

New in version 0.1.5.

Modern ClickHouse servers (20.*+) use text serialization for settings instead of binary serialization. You don't have to add missed settings manually into available. Just specify new settings and it will work.

```
>>> client = Client('localhost', settings={'brand_new_setting': 42})
>>> client.execute('SELECT 1')
```

1.7 Unsupported server versions

Following versions are not supported by this package:

- 20.1.*. Due to keeping alias type name to metadata.

However you can use these versions for your own risk.

CHAPTER 2

API Reference

If you are looking for information on a specific function, class or method, this part of the documentation is for you.

2.1 API

This part of the documentation covers basic classes of the driver: Client, Connection and others.

2.1.1 Client

class clickhouse_driver.**Client**(**args*, ***kwargs*)

Client for communication with the ClickHouse server. Single connection is established per each connected instance of the client.

Parameters

- **settings** – Dictionary of settings that passed to every query (except for the client settings, see below). Defaults to `None` (no additional settings). See all available settings in ClickHouse docs.
- ****kwargs** – All other args are passed to the *`Connection`* constructor.

The following keys when passed in `settings` are used for configuring the client itself:

- `insert_block_size` – chunk size to split rows for `INSERT`. Defaults to `1048576`.
- `strings_as_bytes` – turns off string column encoding/decoding.
- `strings_encoding` – specifies string encoding. UTF-8 by default.
- **use_numpy – Use numpy for columns reading. New in version** *0.2.0*.
- **opentelemetry_traceparent – OpenTelemetry traceparent header as** described by W3C Trace Context recommendation. New in version *0.2.2*.
- **opentelemetry_tracestate – OpenTelemetry tracestate header as** described by W3C Trace Context recommendation. New in version *0.2.2*.

disconnect()
Disconnects from the server.

execute(*query*, *params=None*, *with_column_types=False*, *external_tables=None*, *query_id=None*, *settings=None*, *types_check=False*, *columnar=False*)
Executes query.

Establishes new connection if it wasn't established yet. After query execution connection remains intact for next queries. If connection can't be reused it will be closed and new connection will be created.

Parameters

- **query** – query that will be send to server.
- **params** – substitution parameters for SELECT queries and data for INSERT queries. Data for INSERT can be *list*, *tuple* or `GeneratorType`. Defaults to `None` (no parameters or data).
- **with_column_types** – if specified column names and types will be returned alongside with result. Defaults to `False`.
- **external_tables** – external tables to send. Defaults to `None` (no external tables).
- **query_id** – the query identifier. If no query id specified ClickHouse server will generate it.
- **settings** – dictionary of query settings. Defaults to `None` (no additional settings).
- **types_check** – enables type checking of data for INSERT queries. Causes additional overhead. Defaults to `False`.
- **columnar** – if specified the result of the SELECT query will be returned in column-oriented form. It also allows to INSERT data in columnar form. Defaults to `False` (row-like form).

Returns

- number of inserted rows for INSERT queries with data. Returning rows count from INSERT FROM SELECT is not supported.
- if *with_column_types=False*: *list* of *tuples* with rows/columns.
- **if *with_column_types=True*: *tuple* of 2 elements:**
  - The first element is *list* of *tuples* with rows/columns.
  - The second element information is about columns: names and types.

execute_iter(*query*, *params=None*, *with_column_types=False*, *external_tables=None*, *query_id=None*, *settings=None*, *types_check=False*)
New in version 0.0.14.

Executes SELECT query with results streaming. See, *Streaming results*.

Parameters

- **query** – query that will be send to server.
- **params** – substitution parameters for SELECT queries and data for INSERT queries. Data for INSERT can be *list*, *tuple* or `GeneratorType`. Defaults to `None` (no parameters or data).
- **with_column_types** – if specified column names and types will be returned alongside with result. Defaults to `False`.

- **external_tables** – external tables to send. Defaults to `None` (no external tables).
- **query_id** – the query identifier. If no query id specified ClickHouse server will generate it.
- **settings** – dictionary of query settings. Defaults to `None` (no additional settings).
- **types_check** – enables type checking of data for INSERT queries. Causes additional overhead. Defaults to `False`.

Returns *IterQueryResult* proxy.

execute_with_progress(*query*, *params=None*, *with_column_types=False*, *external_tables=None*, *query_id=None*, *settings=None*, *types_check=False*, *columnar=False*)

Executes SELECT query with progress information. See, *Selecting data with progress statistics*.

Parameters

- **query** – query that will be send to server.
- **params** – substitution parameters for SELECT queries and data for INSERT queries. Data for INSERT can be *list*, *tuple* or `GeneratorType`. Defaults to `None` (no parameters or data).
- **with_column_types** – if specified column names and types will be returned alongside with result. Defaults to `False`.
- **external_tables** – external tables to send. Defaults to `None` (no external tables).
- **query_id** – the query identifier. If no query id specified ClickHouse server will generate it.
- **settings** – dictionary of query settings. Defaults to `None` (no additional settings).
- **types_check** – enables type checking of data for INSERT queries. Causes additional overhead. Defaults to `False`.
- **columnar** – if specified the result will be returned in column-oriented form. Defaults to `False` (row-like form).

Returns *ProgressQueryResult* proxy.

classmethod from_url(*url*)

Return a client configured from the given URL.

For example:

```
clickhouse://[user:password]@localhost:9000/default
clickhouses://[user:password]@localhost:9440/default
```

Three URL schemes are supported: clickhouse:// creates a normal TCP socket connection clickhouses:// creates a SSL wrapped TCP socket connection

Any additional querystring arguments will be passed along to the Connection class's initializer.

insert_dataframe(*query*, *dataframe*, *external_tables=None*, *query_id=None*, *settings=None*)

New in version 0.2.0.

Inserts pandas DataFrame with specified query.

Parameters

- **query** – query that will be send to server.
- **dataframe** – pandas DataFrame.
- **external_tables** – external tables to send. Defaults to `None` (no external tables).
- **query_id** – the query identifier. If no query id specified ClickHouse server will generate it.
- **settings** – dictionary of query settings. Defaults to `None` (no additional settings).

Returns number of inserted rows.

query_dataframe(*query*, *params=None*, *external_tables=None*, *query_id=None*, *settings=None*)

New in version 0.2.0.

Queries DataFrame with specified SELECT query.

Parameters

- **query** – query that will be send to server.
- **params** – substitution parameters. Defaults to `None` (no parameters or data).
- **external_tables** – external tables to send. Defaults to `None` (no external tables).
- **query_id** – the query identifier. If no query id specified ClickHouse server will generate it.
- **settings** – dictionary of query settings. Defaults to `None` (no additional settings).

Returns pandas DataFrame.

2.1.2 Connection

class clickhouse_driver.connection.**Connection**(*host*, *port=None*, *database='default'*, *user='default'*, *password=''*, *client_name='python-driver'*, *connect_timeout=10*, *send_receive_timeout=300*, *sync_request_timeout=5*, *compress_block_size=1048576*, *compression=False*, *secure=False*, *verify=True*, *ssl_version=None*, *ca_certs=None*, *ciphers=None*, *keyfile=None*, *certfile=None*, *alt_hosts=None*, *settings_is_important=False*)

Represents connection between client and ClickHouse server.

Parameters

- **host** – host with running ClickHouse server.
- **port** – port ClickHouse server is bound to. Defaults to `9000` if connection is not secured and to `9440` if connection is secured.
- **database** – database connect to. Defaults to `'default'`.
- **user** – database user. Defaults to `'default'`.
- **password** – user's password. Defaults to `''` (no password).

- **client_name** – this name will appear in server logs. Defaults to `'python-driver'`.
- **connect_timeout** – timeout for establishing connection. Defaults to `10` seconds.
- **send_receive_timeout** – timeout for sending and receiving data. Defaults to `300` seconds.
- **sync_request_timeout** – timeout for server ping. Defaults to `5` seconds.
- **compress_block_size** – size of compressed block to send. Defaults to `1048576`.
- **compression** – specifies whether or not use compression. Defaults to `False`. Possible choices:
  - `True` is equivalent to `'lz4'`.
  - `'lz4'`.
  - `'lz4hc'` high-compression variant of `'lz4'`.
  - `'zstd'`.
- **secure** – establish secure connection. Defaults to `False`.
- **verify** – specifies whether a certificate is required and whether it will be validated after connection. Defaults to `True`.
- **ssl_version** – see `ssl.wrap_socket()` docs.
- **ca_certs** – see `ssl.wrap_socket()` docs.
- **ciphers** – see `ssl.wrap_socket()` docs.
- **keyfile** – see `ssl.wrap_socket()` docs.
- **certfile** – see `ssl.wrap_socket()` docs.
- **alt_hosts** – list of alternative hosts for connection. Example: alt_hosts=host1:port1,host2:port2.
- **settings_is_important** – `False` means unknown settings will be ignored, `True` means that the query will fail with UNKNOWN_SETTING error. Defaults to `False`.

disconnect()

Closes connection between server and client. Frees resources: e.g. closes socket.

2.1.3 QueryResult

class clickhouse_driver.result.**QueryResult**(*packet_generator*, *with_column_types=False*, *columnar=False*)

Stores query result from multiple blocks.

get_result()

Returns stored query result.

2.1.4 ProgressQueryResult

class clickhouse_driver.result.**ProgressQueryResult**(**args*, ***kwargs*)

Stores query result and progress information from multiple blocks. Provides iteration over query progress.

get_result()

Returns stored query result.

2.1.5 IterQueryResult

class clickhouse_driver.result.**IterQueryResult**(*packet_generator*, *with_column_types=False*)

Provides iteration over returned data by chunks (streaming by chunks).

2.2 DB API 2.0

This part of the documentation covers driver DB API.

clickhouse_driver.dbapi.**connect**(*dsn=None*, *host=None*, *user='default'*, *password=''*, *port=9000*, *database='default'*, ***kwargs*)

Create a new database connection.

The connection can be specified via DSN:

```
conn = connect("clickhouse://localhost/test?param1=value1&...")
```

or using database and credentials arguments:

```
conn = connect(database="test", user="default",
password="default", host="localhost", **kwargs)
```

The basic connection parameters are:

- *host*: host with running ClickHouse server.
- *port*: port ClickHouse server is bound to.
- *database*: database connect to.
- *user*: database user.
- *password*: user's password.

See defaults in *Connection* constructor.

DSN or host is required.

Any other keyword parameter will be passed to the underlying Connection class.

Returns a new connection.

exception clickhouse_driver.dbapi.**Warning**

with_traceback()

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

exception clickhouse_driver.dbapi.**Error**

with_traceback()

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

exception clickhouse_driver.dbapi.**DataError**

with_traceback()

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

exception clickhouse_driver.dbapi.**DatabaseError**

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

exception clickhouse_driver.dbapi.**ProgrammingError**

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

exception clickhouse_driver.dbapi.**IntegrityError**

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

exception clickhouse_driver.dbapi.**InterfaceError**

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

exception clickhouse_driver.dbapi.**InternalError**

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

exception clickhouse_driver.dbapi.**NotSupportedError**

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

exception clickhouse_driver.dbapi.**OperationalError**

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

2.2.1 Connection

class clickhouse_driver.dbapi.connection.**Connection**(*dsn=None*, *host=None*, *user='default'*, *password=''*, *port=9000*, *database='default'*, ***kwargs*)

Creates new Connection for accessing ClickHouse database.

Connection is just wrapper for handling multiple cursors (clients) and do not initiate actual connections to the ClickHouse server.

See parameters description in *Connection*.

close()
Close the connection now. The connection will be unusable from this point forward; an *Error* (or subclass) exception will be raised if any operation is attempted with the connection. The same applies to all cursor objects trying to use the connection.

commit()
Do nothing since ClickHouse has no transactions.

cursor()

Returns a new Cursor Object using the connection.

rollback()
Do nothing since ClickHouse has no transactions.

2.2.2 Cursor

class clickhouse_driver.dbapi.cursor.**Cursor**(*client*, *connection*)

close()
Close the cursor now. The cursor will be unusable from this point forward; an *Error* (or subclass) exception will be raised if any operation is attempted with the cursor.

columns_with_types

Returns list of column names with corresponding types of the last .execute*(). E.g. [('x', 'UInt64')].

execute(*operation*, *parameters=None*)
Prepare and execute a database operation (query or command).

Parameters

- **operation** – query or command to execute.
- **parameters** – sequence or mapping that will be bound to variables in the operation.

Returns None

executemany(*operation*, *seq_of_parameters*)
Prepare a database operation (query or command) and then execute it against all parameter sequences found in the sequence *seq_of_parameters*.

Parameters

- **operation** – query or command to execute.
- **seq_of_parameters** – sequences or mappings for execution.

Returns None

fetchall()
Fetch all (remaining) rows of a query result, returning them as a sequence of sequences (e.g. a list of tuples).

Returns list of fetched rows.

fetchmany(*size=None*)
Fetch the next set of rows of a query result, returning a sequence of sequences (e.g. a list of tuples). An empty sequence is returned when no more rows are available.

Parameters **size** – amount of rows to return.

Returns list of fetched rows or empty list.

fetchone()
Fetch the next row of a query result set, returning a single sequence, or None when no more data is available.

Returns the next row of a query result set or None.

rowcount

Returns the number of rows that the last .execute*() produced.

set_external_table(*name*, *structure*, *data*)
Adds external table to cursor context.

If the same table is specified more than once the last one is used.

Parameters

- **name** – name of external table
- **structure** – list of tuples (name, type) that defines table structure. Example [(x, 'Int32')].
- **data** – sequence of rows of tuples or dicts for transmission.

Returns None

set_query_id(*query_id*)
Specifies the query identifier for cursor.

Parameters **query_id** – the query identifier.

Returns None

set_settings(*settings*)
Specifies settings for cursor.

Parameters **settings** – dictionary of query settings

Returns None

set_stream_results(*stream_results*, *max_row_buffer*)
Toggles results streaming from server. Driver will consume block-by-block of *max_row_buffer* size and yield row-by-row from each block.

Parameters

- **stream_results** – enable or disable results streaming.
- **max_row_buffer** – specifies the maximum number of rows to buffer at a time.

Returns None

set_types_check(*types_check*)
Toggles type checking for sequence of INSERT parameters. Disabled by default.

Parameters **types_check** – new types check value.

Returns None

CHAPTER 3

Additional Notes

Legal information, changelog and contributing are here for the interested.

3.1 Development

3.1.1 Test configuration

In `setup.cfg` you can find ClickHouse server port, credentials, logging level and another options than can be tuned during local testing.

3.1.2 Running tests locally

Install desired Python version with system package manager/pyenv/another manager.

ClickHouse on host machine

Install desired versions of `clickhouse-server` and `clickhouse-client` on your machine.

Run tests:

```
python setup.py test
```

ClickHouse in docker

Create container desired version of `clickhouse-server`:

```
docker run --rm -e "TZ=Europe/Moscow" -p 127.0.0.1:9000:9000 --name test-
↪clickhouse-server yandex/clickhouse-server:$VERSION
```

Create container with the same version of `clickhouse-client`:

```
docker run --rm --entrypoint "/bin/sh" --name test-clickhouse-client --link␣
↪test-clickhouse-server:clickhouse-server yandex/clickhouse-client:$VERSION␣
↪-c 'while :; do sleep 1; done'
```

Create `clickhouse-client` script on your host machine:

```
echo -e '#!/bin/bash\n\ndocker exec -e "`env | grep ^TZ=`" test-clickhouse-
↪client clickhouse-client "$@"' | sudo tee /usr/local/bin/clickhouse-client␣
↪> /dev/null
sudo chmod +x /usr/local/bin/clickhouse-client
```

After it container `test-clickhouse-client` will communicate with `test-clickhouse-server` transparently from host machine.

Set `host=clickhouse-server` in `setup.cfg`.

Add entry in hosts file:

```
echo '127.0.0.1 clickhouse-server' | sudo tee -a /etc/hosts > /dev/null
```

Set `TZ=UTC` and run tests:

```
export TZ=UTC
python setup.py test
```

3.2 Changelog

Changelog is available in github repo.

3.3 License

ClickHouse Python Driver is distributed under the MIT license.

3.4 How to Contribute

1. Check for open issues or open a fresh issue to start a discussion around a feature idea or a bug.
2. Fork the repository on GitHub to start making your changes to the **master** branch (or branch off of it).
3. Write a test which shows that the bug was fixed or that the feature works as expected.
4. Send a pull request and bug the maintainer until it gets merged and published.

Python Module Index

C

Index

Q

R

S

W