clickhouse-driver Documentation

Release 0.0.16

clickhouse-driver authors

May 31, 2019

Contents

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 provider two protocols for communication: HTTP protocol and Native (TCP) protocol.

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.

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, Python 2.7, and PyPy.

1.1.2 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.

1.1.3 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]
```

1.1.4 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*.

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)]
```

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
```

(continues on next page)

(continued from previous page)

```
... )
>>>
>>> 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}]
... )
>>> client.execute(
...     'INSERT INTO test (x) VALUES',
...     [[200]]
... )
>>> client.execute(
...     'INSERT INTO test (x) VALUES',
...     ((x, ) for x in range(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) '
```

(continues on next page)

(continued from previous page)

```
...     'SELECT * FROM system.numbers LIMIT %(limit)s',
...     {'limit': 5}
... )
[]
```

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

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 Asynchronous behavior

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.

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* 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 install in order by enable compression suport, 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
... )
```

1.3.8 Reading query profile info

Last query's profile info can be examined. *rows_before_limit* examine example:

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

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': {
...         '': {
...             '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: {b328ad33-60e8-4012-
↪b4cc-97f44a7b28f2} [ 25 ] <Debug> executeQuery: (from 127.0.0.1:57762)␣
↪SELECT 1
2018-12-14 10:24:53,874 INFO     clickhouse_driver.log: {b328ad33-60e8-4012-
↪b4cc-97f44a7b28f2} [ 25 ] <Debug> executeQuery: Query pipeline:
Expression
 Expression
  One

2018-12-14 10:24:53,875 INFO     clickhouse_driver.log: {b328ad33-60e8-4012-
↪b4cc-97f44a7b28f2} [ 25 ] <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: {b328ad33-60e8-4012-
↪b4cc-97f44a7b28f2} [ 25 ] <Debug> MemoryTracker: Peak memory usage (for␣
↪query): 40.23 KiB.
[(1,)]
```

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

INSERT types: `int`, `long`.

SELECT type: `int`.

1.4.2 Float32/64

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

SELECT type: `float`.

1.4.3 Date

INSERT types: `date`.

SELECT type: `date`.

1.4.4 DateTime('timezone')

Timezone support is new in version 0.0.11.

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.

1.4.5 String/FixedString(N)

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

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

String column is encoded/decoded using UTF-8 encoding.

String column can be returned without decoding. 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 UTF-8 encoding, it will return as `bytes`.

During INSERT, if `strings_as_bytes` setting is not specified and string cannot be encoded with `UTF-8`, 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}]
... )
>>> client.execute('SELECT * FROM test')
[('foo',), ('bar',), ('foo',)]
```

For Python 2.7 enum34 package is used.

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: `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]}]
```

(continues on next page)

(continued from previous page)

```
... )
>>> 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`.

1.4.11 IPv4/IPv6

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')}
... ])
>>> 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) '
```

(continues on next page)

(continued from previous page)

```
...     '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"}
... ])
>>> 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'),)]
>>>
```

For Python 2.7 ipaddress package is used.

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. Defaults to `None` (no additional settings). See all available settings in ClickHouse docs.

Driver's settings:

- insert_block_size – chunk size to split rows for `INSERT`. Defaults to `1048576`.
- strings_as_bytes – turns off string column encoding/decoding.

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 will be returned in column-oriented form. Defaults to `False` (row-like form).

Returns

- `None` for INSERT queries.
- 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*)

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`.

Returns *ProgressQueryResult* proxy.

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*)

Represents connection between client and ClickHouse server.

Parameters

- **host** – host with running ClickHouse server.
- **port** – port ClickHouse server is bound to. Defaults to `9000`.
- **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.

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**(*packet_generator*, *with_column_types=False*, *columnar=False*)
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).

CHAPTER 3

Additional Notes

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

3.1 Changelog

Changelog is available in github repo.

3.2 License

ClickHouse Python Driver is distributed under the MIT license.

3.3 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