
aioinflux Documentation

Release 0.5.1

Gustavo Bezerra

Jan 21, 2019

Contents:

1	Installation	3
1.1	Dependencies	3
2	User Guide	5
2.1	TL;DR	6
2.2	Client modes	6
2.3	Writing data	7
2.3.1	Writing dictionary-like objects	7
2.3.2	Writing DataFrames	8
2.3.3	Writing user-defined class objects	8
2.4	Querying data	11
2.4.1	Output formats	11
2.4.2	Retrieving DataFrames	12
2.4.3	Chunked responses	12
2.4.4	Iterating responses	13
2.4.5	Query patterns	14
2.5	Other functionality	14
2.5.1	Authentication	14
2.5.2	Unix domain sockets	14
2.5.3	HTTPS/SSL	14
2.5.4	Database selection	15
2.5.5	Debugging	15
3	Implementation details	17
4	API Reference	19
4.1	Client Interface	19
4.1.1	Result iteration	21
4.2	Serialization	22
4.2.1	Mapping	22
4.2.2	Dataframe	22
4.2.3	User-defined classes	22
5	Contributing	23
6	Alternatives	25

7 Indices and tables	27
Python Module Index	29

Asynchronous Python client for [InfluxDB](#). Built on top of [aiohttp](#) and [asyncio](#). Aioinflux is an alternative to the official InfluxDB Python client.

Aioinflux supports interacting with InfluxDB in a non-blocking way by using [aiohttp](#). It also supports writing and querying of [Pandas](#) dataframes, among other handy functionality.

CHAPTER 1

Installation

To install the latest release:

```
$ pip install aioinflux
$ pip install aioinflux[pandas] # For DataFrame parsing support
```

The library is still in beta, so you may also want to install the latest version from the development branch:

```
$ pip install git+https://github.com/plugaai/aioinflux@dev
```

1.1 Dependencies

Aioinflux supports Python 3.6+ **ONLY**. For older Python versions please use the [official Python client](#). However, there is [some discussion](#) regarding Pypy/Python 3.5 support.

The main third-party library dependency is `aiohttp`, for all HTTP request handling. and `pandas` for `DataFrame` reading/writing support.

There are currently no plans to support other HTTP libraries besides `aiohttp`. If `aiohttp` + `asyncio` is not your soup, see [Alternatives](#).

- *TL;DR*
- *Client modes*
- *Writing data*
 - *Writing dictionary-like objects*
 - *Writing DataFrames*
 - *Writing user-defined class objects*
 - * *User-defined class schema/type annotations*
 - * *@lineprotocol options*
 - * *Performance*
- *Querying data*
 - *Output formats*
 - *Retrieving DataFrames*
 - *Chunked responses*
 - *Iterating responses*
 - *Query patterns*
- *Other functionality*
 - *Authentication*
 - *Unix domain sockets*
 - *HTTPS/SSL*
 - *Database selection*

2.1 TL;DR

This sums most of what you can do with aioinflux:

```
import asyncio
from aioinflux import InfluxDBClient

point = {
    'time': '2009-11-10T23:00:00Z',
    'measurement': 'cpu_load_short',
    'tags': {'host': 'server01',
            'region': 'us-west'},
    'fields': {'value': 0.64}
}

async def main():
    async with InfluxDBClient(db='testdb') as client:
        await client.create_database(db='testdb')
        await client.write(point)
        resp = await client.query('SELECT value FROM cpu_load_short')
        print(resp)

asyncio.get_event_loop().run_until_complete(main())
```

2.2 Client modes

Despite the library's name, *InfluxDBClient* can also run in non-async mode (a.k.a blocking) mode. It can be useful for debugging and exploratory data analysis.

The running mode for can be switched on-the-fly by changing the `mode` attribute:

```
client = InfluxDBClient(mode='blocking')
client.mode = 'async'
```

The `blocking` mode is implemented through a decorator that automatically runs coroutines on the event loop as soon as they are generated. Usage is almost the same as in the `async` mode, but without the need of using `await` and being able to run from outside of a coroutine function:

```
client = InfluxDBClient(db='testdb', mode='blocking')
client.ping()
client.write(point)
client.query('SELECT value FROM cpu_load_short')
```

Note: The need for the `blocking` mode has been somewhat supplanted by the new async REPL available with the release of IPython 7.0. See [this blog post](#) for details.

If you are having issues running `blocking` mode with recent Python/IPython versions, see [this issue](#) for other possible workarounds.

2.3 Writing data

To write data to InfluxDB, use `InfluxDBClient`'s `write()` method. Successful writes will return `True`. In case some error occurs `InfluxDBWriteError` exception will be raised.

Input data to `write()` can be:

1. A mapping (e.g. dict) containing the keys: `measurement`, `time`, `tags`, `fields`
2. A `pandas.DataFrame` with a `DatetimeIndex`
3. A user defined class decorated w/ `lineprotocol()` (**recommended**, see *below*)
4. A string (`str` or `bytes`) properly formatted in InfluxDB's line protocol
5. An iterable of one of the above

Input data in formats 1-3 are serialized into the `line protocol` before being written to InfluxDB. `str` or `bytes` are assumed to already be in line protocol format and are inserted into InfluxDB as they are. All functionality regarding JSON parsing (InfluxDB's only output format) and serialization to line protocol (InfluxDB's only input format) is located in the `serialization` subpackage.

Beware that serialization is not highly optimized (C extensions / cythonization PRs are welcome!) and may become a bottleneck depending on your application's performance requirements. It is, however, reasonably (3-10x) **faster** than InfluxDB's official Python client.

2.3.1 Writing dictionary-like objects

Warning: This is the same format as the one used by InfluxDB's official Python client and is implemented in Aioinflux for compatibility purposes only. Using dictionaries to write data to InfluxDB is slower and more error-prone than the other methods provided by Aioinflux and therefore **discouraged**.

Aioinflux accepts any dictionary-like object (mapping) as input. The dictionary must contain the following keys:

- 1) **measurement:** Optional. Must be a string-like object. If omitted, must be specified when calling `write()` by passing a `measurement` argument.
- 2) **time:** Optional. The value can be `datetime.datetime`, date-like string (e.g., `2017-01-01`, `2009-11-10T23:00:00Z`) or anything else that can be parsed by `pandas.Timestamp`. See [Pandas documentation](#) for details. If Pandas is not available, `ciso8601` is used instead for date-like string parsing.
- 3) **tags:** Optional. This must contain another mapping of field names and values. Both tag keys and values should be strings.
- 4) **fields:** Mandatory. This must contain another mapping of field names and values. Field keys should be strings. Field values can be `float`, `int`, `str`, `bool` or `None` or any its subclasses. Attempting to use Numpy types will cause errors as `np.int64`, `np.float64`, etc are not subclasses of Python's built-in numeric types. Use dataframes for writing data using Numpy types.

Any keys other than the above will be ignored when writing data to InfluxDB.

A typical dictionary-like point would look something like the following:

```
{ 'time': '2009-11-10T23:00:00Z',
  'measurement': 'cpu_load_short',
  'tags': { 'host': 'server01', 'region': 'us-west' },
  'fields': { 'value1': 0.64, 'value2': True, 'value3': 10 } }
```

Note: Timestamps and timezones

Working with timezones in computing tends to be quite messy. To avoid such problems, the [broadly agreed](#) upon idea is to store timestamps in UTC. This is how both InfluxDB and Pandas treat timestamps internally.

Pandas and many other libraries also assume all input timestamps are in UTC unless otherwise explicitly noted. Aioinflux does the same and assumes any timezone-unaware `datetime.datetime` object or datetime-like strings is in UTC. Aioinflux does not raise any warnings when timezone-unaware input is passed and silently assumes it to be in UTC.

2.3.2 Writing DataFrames

Aioinflux also accepts Pandas dataframes as input. The only requirements for the dataframe is that the index **must** be of type `DatetimeIndex`. Also, any column whose dtype is object will be converted to a string representation.

A typical dataframe input should look something like the following:

		LUY	BEM	AJW	tag
2017-06-24	08:45:17.929097+00:00	2.545409	5.173134	5.532397	B
2017-06-24	10:15:17.929097+00:00	-0.306673	-1.132941	-2.130625	E
2017-06-24	11:45:17.929097+00:00	0.894738	-0.561979	-1.487940	B
2017-06-24	13:15:17.929097+00:00	-1.799512	-1.722805	-2.308823	D
2017-06-24	14:45:17.929097+00:00	0.390137	-0.016709	-0.667895	E

The measurement name must be specified with the `measurement` argument when calling `write()`. Columns that should be treated as tags must be specified by passing a sequence as the `tag_columns` argument. Additional tags (not present in the actual dataframe) can also be passed using arbitrary keyword arguments.

Example:

```
client = InfluxDBClient(db='testdb', mode='blocking')
client.write(df, measurement='prices', tag_columns=['tag'], asset_class='equities')
```

In the example above, `df` is the dataframe we are trying to write to InfluxDB and `measurement` is the measurement we are writing to.

`tag_columns` is in an optional iterable telling which of the dataframe columns should be parsed as tag values. If `tag_columns` is not explicitly passed, all columns in the dataframe whose dtype is not `DatetimeIndex` will be treated as InfluxDB field values.

Any other keyword arguments passed to `write()` are treated as extra tags which will be attached to the data being written to InfluxDB. Any string which is a valid [InfluxDB identifier](#) and valid [Python identifier](#) can be used as an extra tag key (with the exception of the strings `data`, `measurement` and `tag_columns`).

See [API reference](#) for details.

2.3.3 Writing user-defined class objects

Changed in version 0.5.0.

Aioinflux can add write any arbitrary user-defined class to InfluxDB through the use of the `lineprotocol()` decorator. This decorator monkey-patches an existing class and adds a `to_lineprotocol` method, which is used internally by InfluxDB to serialize the class data into a InfluxDB-compatible format. In order to generate `to_lineprotocol`, a typed schema must be defined using [type hints](#) in the form of type annotations or a schema dictionary.

This is the fastest and least error-prone method of writing data into InfluxDB provided by Aioinflux.

We recommend using `lineprotocol()` with `NamedTuple`:

```
from aioinflux import *
from typing import NamedTuple

@lineprotocol
class Trade(NamedTuple):
    timestamp: TIMEINT
    instrument: TAGENUM
    source: TAG
    side: TAG
    price: FLOAT
    size: INT
    trade_id: STR
```

Alternatively, the functional form of `namedtuple()` can also be used:

```
from collections import namedtuple

schema = dict(
    timestamp=TIMEINT,
    instrument=TAG,
    source=TAG,
    side=TAG,
    price=FLOAT,
    size=INT,
    trade_id=STR,
)

# Create class
Trade = namedtuple('Trade', schema.keys())

# Monkey-patch existing class and add ``to_lineprotocol``
Trade = lineprotocol(Trade, schema=schema)
```

Dataclasses (or any other user-defined class) can be used as well:

```
from dataclasses import dataclass

@lineprotocol
@dataclass
class Trade:
    timestamp: TIMEINT
    instrument: TAGENUM
    source: TAG
    side: TAG
    price: FLOAT
    size: INT
    trade_id: STR
```

If you want to preserve type annotations for another use, you can pass your serialization schema as a dictionary as well:

```
@lineprotocol(schema=dict(timestamp=TIMEINT, value=FLOAT))
@dataclass
class MyTypedClass:
```

(continues on next page)

(continued from previous page)

```

    timestamp: int
    value: float

print(MyTypedClass.__annotations__)
# {'timestamp': <class 'int'>, 'value': <class 'float'>}

MyTypedClass(1547710904202826000, 2.1).to_lineprotocol()
# b'MyTypedClass value=2.1 1547710904202826000'
```

The modified class will have a dynamically generated `to_lineprotocol` method which generates a line protocol representation of the data contained by the object:

```

trade = Trade(
    timestamp=1540184368785116000,
    instrument='AAPL',
    source='NASDAQ',
    side='BUY',
    price=219.23,
    size=100,
    trade_id='34a1e085-3122-429c-9662-7ce82039d287'
)

trade.to_lineprotocol()
# b'Trade,instrument=AAPL,source=NASDAQ,side=BUY price=219.23,size=100i,trade_id=
↪ "34a1e085-3122-429c-9662-7ce82039d287" 1540184368785116000'
```

Calling `to_lineprotocol` by the end-user is not necessary but may be useful for debugging.

`to_lineprotocol` is automatically used by `write()` when present.

```

client = InfluxDBClient()
await client.write(trade) # True
```

User-defined class schema/type annotations

In Aioinflux, InfluxDB types (and derived types) are represented by `TypeVar` defined in `usertype`. All schema values (type annotations) MUST be one of those types. The types available are based on the native types of InfluxDB (see the [InfluxDB docs](#) for details), with some extra types to help the serialization to line protocol and/or allow more flexible usage (such as the use of `Enum` objects).

Type	Description
MEASUREMENT	Optional. If missing, the measurement becomes the class name
TIMEINT	Timestamp is a nanosecond UNIX timestamp
TIMESTR	Timestamp is a datetime string (somewhat compliant to ISO 8601)
TIMEDT	Timestamp is a <code>datetime.datetime</code> (or subclasses such as <code>pandas.Timestamp</code>)
TAG	Treats field as an InfluxDB tag
TAGENUM	Same as TAG but allows the use of <code>Enum</code>
BOOL	Boolean field
INT	Integer field
FLOAT	Float field
STR	String field
ENUM	Same as STR but allows the use of <code>Enum</code>

TAG* types are optional. One and only one TIME* type must present. At least ONE field type be present.

@lineprotocol options

The `lineprotocol()` function/decorator provides some options to customize how object serialization is performed. See the [API reference](#) for details.

Performance

Serialization using `lineprotocol()` is about 3x faster than dictionary-like objects (or about 10x faster than the official Python client). See this [notebook](#) for a simple benchmark.

Beware that setting `rm_none=True` can have substantial performance impact especially when the number of fields/tags is very large (20+).

2.4 Querying data

Querying data is as simple as passing an InfluxDB query string to `query()`:

```
await client.query('SELECT myfield FROM mymeasurement')
```

By default, this returns JSON data:

```
{'results': [{'series': [{'columns': ['time', 'Price', 'Volume'],
    'name': 'mymeasurement',
    'values': [[1491963424224703000, 5783, 100],
    [1491963424375146000, 5783, 200],
    [1491963428374895000, 5783, 100],
    [1491963429645478000, 5783, 1100],
    [1491963429655289000, 5783, 100],
    [1491963437084443000, 5783, 100],
    [1491963442274656000, 5783, 900],
    [1491963442274657000, 5782, 5500],
    [1491963442274658000, 5781, 3200],
    [1491963442314710000, 5782, 100]]]}],
    'statement_id': 0}]}
```

See [InfluxDB official docs](#) for more on the InfluxDB's HTTP API specifics.

2.4.1 Output formats

When using `query()` data can return data in one of the following formats:

- 1) `json`: Default. Returns the a dictionary containing the JSON response received from InfluxDB.
- 2) `bytes`: Returns raw, non-parsed JSON binary blob as received from InfluxDB. The contents of the returns JSON blob are not checked at all. Useful for response caching.
- 3) `dataframe`: Parses the result into a Pandas dataframe or a dictionary of dataframes. See [Retrieving DataFrames](#) for details.
- 4) `iterable`: Wraps the JSON response in a `InfluxDBResult` or `InfluxDBChunkedResult` object. This object main purpose is to facilitate iteration of data. See [Iterating responses](#) for details.

The output format for can be switched on-the-fly by changing the `output` attribute:

```
client = InfluxDBClient(output='dataframe')
client.mode = 'json'
```

Beware that when passing `chunked=True`, the result type will be an async generator. See [Chunked responses](#) for details.

2.4.2 Retrieving DataFrames

When the client is in `dataframe` mode, `query()` will usually return a `pandas.DataFrame`:

	Price	Volume
2017-04-12 02:17:04.224703+00:00	5783	100
2017-04-12 02:17:04.375146+00:00	5783	200
2017-04-12 02:17:08.374895+00:00	5783	100
2017-04-12 02:17:09.645478+00:00	5783	1100
2017-04-12 02:17:09.655289+00:00	5783	100
2017-04-12 02:17:17.084443+00:00	5783	100
2017-04-12 02:17:22.274656+00:00	5783	900
2017-04-12 02:17:22.274657+00:00	5782	5500
2017-04-12 02:17:22.274658+00:00	5781	3200
2017-04-12 02:17:22.314710+00:00	5782	100

Note: On multi-statement queries and/or statements that return multiple InfluxDB series (such as a `GROUP by "tag"` query), a list of dictionaries of dataframes will be returned. Aioinflux generates a dataframe for each series contained in the JSON returned by InfluxDB. See this [Github issue](#) for further discussion.

When generating dataframes, InfluxDB types are mapped to the following Numpy/Pandas dtypes:

InfluxDB type	Dataframe column dtype
Float	float64
Integer	int64
String	object
Boolean	bool
Timestamp	datetime64

2.4.3 Chunked responses

Aioinflux supports InfluxDB chunked queries. Passing `chunked=True` when calling `query()`, returns an `AsyncGenerator` object, which can asynchronously iterated. Using chunked requests allows response processing to be partially done before the full response is retrieved, reducing overall query time.

```
chunks = await client.query("SELECT * FROM mymeasurement", chunked=True)
async for chunk in chunks:
    # do something
    await process_chunk(...)
```

Chunked responses are not supported when using the `dataframe` output format.

See the [InfluxDB official docs](#) for more on chunked responses.

2.4.4 Iterating responses

By default, `query()` returns a parsed JSON response from InfluxDB. In order to easily iterate over that JSON response point by point, Aioinflux provides the `iterpoints()` function, which returns a generator object:

```
from aioinflux import iterpoints

r = client.query('SELECT * from h2o_quality LIMIT 10')
for i in iterpoints(r):
    print(i)
```

```
[1439856000000000000, 41, 'coyote_creek', '1']
[1439856000000000000, 99, 'santa_monica', '2']
[1439856360000000000, 11, 'coyote_creek', '3']
[1439856360000000000, 56, 'santa_monica', '2']
[1439856720000000000, 65, 'santa_monica', '3']
```

`iterpoints()` can also be used with chunked responses:

```
chunks = await client.query('SELECT * from h2o_quality', chunked=True)
async for chunk in chunks:
    for point in iterpoints(chunk):
        # do something
```

By default, the generator returned by `iterpoints()` yields a plain list of values without doing any expensive parsing. However, in case a specific format is needed, an optional parser argument can be passed. `parser` is a function that takes the raw value list for each data point and an additional metadata dictionary containing all or a subset of the following: `{ 'columns', 'name', 'tags', 'statement_id' }`.

```
r = await client.query('SELECT * from h2o_quality LIMIT 5')
for i in iterpoints(r, lambda x, meta: dict(zip(meta['columns'], x))):
    print(i)
```

```
{ 'time': 1439856000000000000, 'index': 41, 'location': 'coyote_creek', 'randtag': '1' }
{ 'time': 1439856000000000000, 'index': 99, 'location': 'santa_monica', 'randtag': '2' }
{ 'time': 1439856360000000000, 'index': 11, 'location': 'coyote_creek', 'randtag': '3' }
{ 'time': 1439856360000000000, 'index': 56, 'location': 'santa_monica', 'randtag': '2' }
{ 'time': 1439856720000000000, 'index': 65, 'location': 'santa_monica', 'randtag': '3' }
```

Besides explicitly parsing a with a raw JSON response, `iterpoints()` is also used by `InfluxDBResult` and `InfluxDBChunkedResult` when using iterable mode:

```
client.output = 'iterable'
# Returns InfluxDBResult object
r = client.query('SELECT * from h2o_quality LIMIT 10')
for i in r:
    # do something

# Returns InfluxDBChunkedResult object
r = await client.query('SELECT * from h2o_quality', chunked=True)
async for i in r:
    # do something

# Returns InfluxDBChunkedResult object
r = await client.query('SELECT * from h2o_quality', chunked=True)
async for chunk in r.iterchunks():
    # do something with JSON chunk
```

2.4.5 Query patterns

Aioinflux provides a wrapping mechanism around `query()` in order to provide convenient access to commonly used query patterns.

Query patterns are query strings containing optional named “replacement fields” surrounded by curly braces `{ }`, just as in `str_format()`. Replacement field values are defined by keyword arguments when calling the method associated with the query pattern. Differently from plain `str_format()`, positional arguments are also supported and can be mixed with keyword arguments.

Aioinflux built-in query patterns are defined [here](#). Users can also dynamically define additional query patterns by using the `set_query_pattern()` helper method. User-defined query patterns have the disadvantage of not being shown for auto-completion in IDEs such as Pycharm. However, they do show up in dynamic environments such as Jupyter. If you have a query pattern that you think will be used by many people and should be built-in, please submit a PR.

Built-in query pattern examples:

```
client.create_database(db='foo')      # CREATE DATABASE {db}
client.drop_measurement('bar')        # DROP MEASUREMENT {measurement}
client.show_users()                  # SHOW USERS

# Positional and keyword arguments can be mixed
client.show_tag_values_from('bar', key='spam') # SHOW TAG VALUES FROM {measurement}
↪WITH key = "{key}"
```

Please refer to InfluxDB [documentation](#) for further query-related information.

2.5 Other functionality

2.5.1 Authentication

Aioinflux supports basic HTTP authentication provided by `aiohttp.BasicAuth`. Simply pass username and password when instantiating `InfluxDBClient`:

```
client = InfluxDBClient(username='user', password='pass')
```

2.5.2 Unix domain sockets

If your InfluxDB server uses UNIX domain sockets you can use `unix_socket` when instantiating `InfluxDBClient`:

```
client = InfluxDBClient(unix_socket='/path/to/socket')
```

See `aiohttp.UnixConnector` for details.

2.5.3 HTTPS/SSL

Aioinflux/InfluxDB uses HTTP by default, but HTTPS can be used by passing `ssl=True` when instantiating `InfluxDBClient`. If you are accessing your InfluxDB instance over the public internet, setting up HTTPS is strongly recommended.

```
client = InfluxDBClient(host='my.host.io', ssl=True)
```

2.5.4 Database selection

After the instantiation of the *InfluxDBClient* object, database can be switched by changing the db attribute:

```
client = InfluxDBClient(db='db1')
client.db = 'db2'
```

Beware that differently from some NoSQL databases (such as MongoDB), InfluxDB requires that a databases is explicitly created (by using the `CREATE DATABASE` query) before doing any operations on it.

2.5.5 Debugging

If you are having problems while using Aioinflux, enabling logging might be useful.

Below is a simple way to setup logging from your application:

```
import logging

logging.basicConfig()
logging.getLogger('aioinflux').setLevel(logging.DEBUG)
```

For further information about logging, please refer to the [official documentation](#).

Implementation details

Since InfluxDB exposes all its functionality through an [HTTP API](#), *InfluxDBClient* tries to be nothing more than a thin and simple wrapper around that API.

The InfluxDB HTTP API exposes exactly three endpoints/functions: *ping()*, *write()* and *query()*.

InfluxDBClient merely wraps these three functions and provides some parsing functionality for generating line protocol data (when writing) and parsing JSON responses (when querying).

Additionally, *partials* are used in order to provide convenient access to commonly used query patterns. See the [Query patterns](#) section for details.

- *Client Interface*
 - *Result iteration*
- *Serialization*
 - *Mapping*
 - *Dataframe*
 - *User-defined classes*

4.1 Client Interface

```
class aioinflux.client.InfluxDBClient (host='localhost', port=8086, mode='async',
                                       output='json', db=None, *, ssl=False,
                                       unix_socket=None, username=None, pass-
                                       word=None, database=None, loop=None)
```

ping()

Pings InfluxDB. Returns a dictionary containing the headers of the response from *influxd*.

Return type `dict`

query (*q*, **args*, *epoch*='ns', *chunked*=False, *chunk_size*=None, *db*=None, *parser*=None, ***kwargs*)

Sends a query to InfluxDB. Please refer to the InfluxDB documentation for all the possible queries: https://docs.influxdata.com/influxdb/latest/query_language/

Parameters

- **q** (`AnyStr`) – Raw query string
- **args** – Positional arguments for query patterns

- **db** (*Optional[str]*) – Database to be queried. Defaults to *self.db*.
- **epoch** (*str*) – Precision level of response timestamps. Valid values: {'ns', 'u', 'µ', 'ms', 's', 'm', 'h'}.
- **chunked** (*bool*) – If True, makes InfluxDB return results in streamed batches rather than as a single response. Returns an AsyncGenerator which yields responses in the same format as non-chunked queries.
- **chunk_size** (*Optional[int]*) – Max number of points for each chunk. By default, InfluxDB chunks responses by series or by every 10,000 points, whichever occurs first.
- **kwargs** – Keyword arguments for query patterns
- **parser** (*Optional[Callable]*) – Optional parser function for 'iterable' mode

Return type *Union[Asyncgenerator[+T_co, -T_contra], dict, bytes, InfluxDBResult, InfluxDBChunkedResult]*

Returns Response in the format specified by the combination of `InfluxDBClient.output` and `chunked`

classmethod set_query_pattern (*name, qp*)

Defines custom methods to provide quick access to commonly used query patterns. Query patterns are plain strings, with optional the named placed holders. Named placed holders are processed as keyword arguments in `str.format`. Positional arguments are also supported.

Sample query pattern: "SELECT mean(load) FROM cpu_stats WHERE host = '{host}' AND time > now() - {days}d"

Parameters

- **name** (*str*) – Name of the query pattern class method. Must be a valid Python identifier.
- **qp** (*str*) – Query pattern string

Return type *None*

write (*data, measurement=None, db=None, precision=None, rp=None, tag_columns=None, **extra_tags*)

Writes data to InfluxDB. Input can be:

1. A mapping (e.g. dict) containing the keys: `measurement, time, tags, fields`
2. A Pandas `DataFrame` with a `DatetimeIndex`
3. A user defined class decorated w/ `lineprotocol()`
4. A string (*str* or *bytes*) properly formatted in InfluxDB's line protocol
5. An iterable of one of the above

Input data in formats 1-3 are parsed to the line protocol before being written to InfluxDB. See the [InfluxDB docs](#) for more details.

Parameters

- **data** (*Union[AnyStr, Mapping[~KT, +VT_co], Iterable[Union[AnyStr, Mapping[~KT, +VT_co]]]]*) – Input data (see description above).
- **measurement** (*Optional[str]*) – Measurement name. Mandatory when when writing DataFrames only. When writing dictionary-like data, this field is treated as the default value for points that do not contain a *measurement* field.
- **db** (*Optional[str]*) – Database to be written to. Defaults to *self.db*.

- **precision** (`Optional[str]`) – Sets the precision for the supplied Unix time values. Ignored if input timestamp data is of non-integer type. Valid values: {'ns', 'u', 'µ', 'ms', 's', 'm', 'h'}
- **rp** (`Optional[str]`) – Sets the target retention policy for the write. If unspecified, data is written to the default retention policy.
- **tag_columns** (`Optional[Iterable[+T_co]]`) – Columns to be treated as tags (used when writing DataFrames only)
- **extra_tags** – Additional tags to be added to all points passed.

Return type `bool`

Returns Returns *True* if insert is successful. Raises *ValueError* exception otherwise.

exception `aioinflux.client.InfluxDBError`

Raised when an server-side error occurs

exception `aioinflux.client.InfluxDBWriteError` (*resp*)

Raised when a server-side writing error occurs

4.1.1 Result iteration

class `aioinflux.iterutils.InfluxDBChunkedResult` (*gen, parser=None, query=None*)

Wrapper around an InfluxDB chunked response

class `aioinflux.iterutils.InfluxDBResult` (*data, parser=None, query=None*)

Wrapper around an InfluxDB response

`aioinflux.iterutils.iterpoints` (*resp, parser=None*)

Iterates a response JSON yielding data point by point.

Can be used with both regular and chunked responses. By default, returns just a plain list of values representing each point, without column names, or other metadata.

In case a specific format is needed, an optional `parser` argument can be passed. `parser` is a function that takes a list of values for each data point and a metadata dictionary containing all or a subset of the following: {'columns', 'name', 'tags', 'statement_id'}.

Sample parser function:

```
def parser(x, meta):
    return dict(zip(meta['columns'], x))
```

Parameters

- **resp** (`dict`) – Dictionary containing parsed JSON (output from `InfluxDBClient.query`)
- **parser** (`Optional[Callable]`) – Optional parser function

Return type `Iterator[+T_co]`

Returns Generator object

4.2 Serialization

4.2.1 Mapping

`aioinflux.serialization.mapping.serialize(point, measurement=None, **extra_tags)`

Converts dictionary-like data into a single line protocol line (point)

Return type `bytes`

4.2.2 Dataframe

4.2.3 User-defined classes

`aioinflux.serialization.usertype.lineprotocol(cls=None, *, schema=None, rm_none=False, extra_tags=None, placeholder=False)`

Adds `to_lineprotocol` method to arbitrary user-defined classes

Parameters

- **schema** – Schema dictionary (attr/type pairs).
- **rm_none** – Whether apply a regex to remove `None` values. If `False`, passing `None` values to boolean, integer or float or time fields will result in write errors. Setting to `True` is “safer” but impacts performance.
- **extra_tags** – Hard coded tags to be added to every point generated.
- **placeholder** – If no field attributes are present, add a placeholder attribute (`_`) which is always equal to `True`. This is a workaround for creating field-less points (which is not supported natively by InfluxDB)

exception `aioinflux.serialization.usertype.SchemaError`

Raised when invalid schema is passed to `lineprotocol()`

CHAPTER 5

Contributing

To contribute, fork the repository on GitHub, make your changes and submit a pull request.
Aioinflux is not a mature project yet, so just simply raising issues is also greatly appreciated :)

CHAPTER 6

Alternatives

- [InfluxDB-Python](#): The official blocking-only client. Based on Requests.
- [influx-sansio](#): Fork of aioinflux using curio/trio and asks as a backend.

CHAPTER 7

Indices and tables

- `genindex`
- `modindex`
- `search`

a

`aioinflux.iterutils`, [21](#)

`aioinflux.serialization.mapping`, [22](#)

`aioinflux.serialization.usertype`, [22](#)

A

`aioinflux.iterutils` (*module*), 21
`aioinflux.serialization.mapping` (*module*), 22
`aioinflux.serialization.usertype` (*module*), 22

I

`InfluxDBChunkedResult` (*class in aioinflux.iterutils*), 21
`InfluxDBClient` (*class in aioinflux.client*), 19
`InfluxDBError`, 21
`InfluxDBResult` (*class in aioinflux.iterutils*), 21
`InfluxDBWriteError`, 21
`iterpoints()` (*in module aioinflux.iterutils*), 21

L

`lineprotocol()` (*in module aioinflux.serialization.usertype*), 22

P

`ping()` (*aioinflux.client.InfluxDBClient method*), 19

Q

`query()` (*aioinflux.client.InfluxDBClient method*), 19

S

`SchemaError`, 22
`serialize()` (*in module aioinflux.serialization.mapping*), 22
`set_query_pattern()` (*aioinflux.client.InfluxDBClient class method*), 20

W

`write()` (*aioinflux.client.InfluxDBClient method*), 20