questdb

Release 2.0.2

QuestDB

Apr 11, 2024

CONTENTS

This is the official Python client library for QuestDB.

This client library implements QuestDB's variant of the InfluxDB Line Protocol (ILP) over HTTP and TCP.

ILP provides the fastest way to insert data into QuestDB.

This implementation supports authentication and full-connection encryption with TLS.

CHAPTER

ONE

QUICKSTART

The latest version of the library is 2.0.2 (changelog).

```
python3 -m pip install -U questdb[dataframe]
```

Please start by setting up QuestDB . Once set up, you can use this library to insert data.

The most common way to insert data is from a Pandas dataframe.

```
import pandas as pd
from questdb.ingress import Sender

df = pd.DataFrame({
    'id': pd.Categorical(['toronto1', 'paris3']),
    'temperature': [20.0, 21.0],
    'humidity': [0.5, 0.6],
    'timestamp': pd.to_datetime(['2021-01-01', '2021-01-02'])})

conf = f'http::addr=localhost:9000;'
with Sender.from_conf(conf) as sender:
    sender.dataframe(df, table_name='sensors', at='timestamp')
```

You can also send individual rows. This only requires a more minimal installation:

```
python3 -m pip install -U questdb
```

```
from questdb.ingress import Sender, TimestampNanos

conf = f'http::addr=localhost:9000;'
with Sender.from_conf(conf) as sender:
    sender.row(
        'sensors',
        symbols={'id': 'toronto1'},
        columns={'temperature': 20.0, 'humidity': 0.5},
        at=TimestampNanos.now())
    sender.flush()
```

To connect via the older TCP protocol, set the configuration string to:

```
conf = f'tcp::addr=localhost:9009;'
with Sender.from_conf(conf) as sender:
    ...
```

You can continue by reading the Sending Data Over ILP guide.

CHAPTER

TWO

LINKS

- Core database documentation
- Python library documentation
- GitHub repository
- Package on PyPI

CHAPTER

THREE

COMMUNITY

If you need help, you can ask on Stack Overflow: We monitor the `#questdb` and `#py-questdb-client` tags.

Alternatively, you may find us on Slack.

You can also sign up to our mailing list to get notified of new releases.

CHAPTER

FOUR

LICENSE

The code is released under the Apache License 2.0.

CHAPTER

FIVE

CONTENTS

5.1 Installation

5.1.1 Dependency

The Python QuestDB client does not have any additional run-time dependencies and will run on any version of Python >= 3.8 on most platforms and architectures.

Optional Dependencies

Ingesting dataframes also require the following dependencies to be installed:

- `pandas`
- `pyarrow`
- `numpy`

These are bundled as the `dataframe` extra.

Without this option, the `questdb` package has no dependencies other than to the Python standard library.

PIP

You can install it (or update it) globally by running:

```
python3 -m pip install -U questdb[dataframe]
```

Or, from within a virtual environment:

```
pip install -U questdb[dataframe]
```

If you don't need to work with dataframes:

```
python3 -m pip install -U questdb
```

Poetry

If you're using poetry, you can add `questdb` as a dependency:

```
poetry add questdb[dataframe]
```

Similarly, if you don't need to work with dataframes:

```
poetry add questdb
```

or to update the dependency:

```
poetry update questdb
```

5.1.2 Verifying the Installation

If you want to check that you've installed the wheel correctly, you can run the following statements from a `python3` interactive shell:

```
>>> import questdb.ingress
>>> buf = questdb.ingress.Buffer()
>>> buf.row('test', symbols={'a': 'b'})
<questdb.ingress.Buffer object at 0x104b68240>
>>> str(buf)
'test,a=b\n'
```

If you also want to if check you can serialize from Pandas (which requires additional dependencies):

```
>>> import questdb.ingress
>>> import pandas as pd
>>> df = pd.DataFrame({'a': [1, 2]})
>>> buf = questdb.ingress.Buffer()
>>> buf.dataframe(df, table_name='test')
>>> str(buf)
'test a=1i\ntest a=2i\n'
```

5.2 Sending Data over ILP

5.2.1 Overview

The *Sender* class is a client that inserts rows into QuestDB via the ILP protocol, with support for both ILP over TCP and the newer and recommended ILP over HTTP. The sender also supports TLS and authentication.

```
from questdb.ingress import Sender, TimestampNanos
import pandas as pd

conf = 'http::addr=localhost:9000;'
with Sender.from_conf(conf) as sender:
    # One row at a time
    sender.row(
```

(continues on next page)

(continued from previous page)

```
        'weather_sensor',
        symbols={'id': 'toronto1'},
        columns={'temperature': 23.5, 'humidity': 0.49},
        at=TimestampNanos.now())

    # Whole dataframes at once - MUCH FASTER
    df = pd.DataFrame({
        'id': ['dubai2', 'memphis7'],
        'temperature': [41.2, 33.3],
        'humidity': [0.34, 0.55],
        'timestamp': [
            pd.Timestamp('2021-01-01 12:00:00'),
            pd.Timestamp('2021-01-01 12:00:01')
        ]
    })
    sensor.dataframe('weather_sensor', df, at='timestamp')
```

The `Sender` object holds an internal buffer which will be flushed and sent at when the `with` block ends.

You can read more on *Preparing Data* and *Flushing*.

5.2.2 Constructing the Sender

From Configuration

The `Sender` class is generally initialized from a *configuration string*.

```
from questdb.ingress import Sender

conf = 'http::addr=localhost:9000;'
with Sender.from_conf(conf) as sender:
    ...
```

See the *Configuration* guide for more details.

From Env Variable

You can also initialize the sender from an environment variable:

```
export QDB_CLIENT_CONF='http::addr=localhost:9000;'
```

The content of the environment variable is the same *configuration string* as taken by the *`Sender.from_conf`* method, but moving it to an environment variable is more secure and allows you to avoid hardcoding sensitive information such as passwords and tokens in your code.

```
from questdb.ingress import Sender

with Sender.from_env() as sender:
    ...
```

Programmatic Construction

If you prefer, you can also construct the sender programmatically. See *Programmatic Construction*.

5.2.3 Preparing Data

Appending Rows

You can append as many rows as you like by calling the *Sender.row* method. The full method arguments are documented in the *Buffer.row* method.

Appending Pandas Dataframes

The sender can also append data from a Pandas dataframe.

This is orders of magnitude faster than appending rows one by one.

```
from questdb.ingress import Sender, IngressError

import sys
import pandas as pd


def example(host: str = 'localhost', port: int = 9000):
    df = pd.DataFrame({
            'pair': ['USDGBP', 'EURJPY'],
            'traded_price': [0.83, 142.62],
            'qty': [100, 400],
            'limit_price': [0.84, None],
            'timestamp': [
                pd.Timestamp('2022-08-06 07:35:23.189062', tz='UTC'),
                pd.Timestamp('2022-08-06 07:35:23.189062', tz='UTC')]})
    try:
        with Sender.from_conf(f"http::addr={host}:{port};") as sender:
            sender.dataframe(
                df,
                table_name='trades',  # Table name to insert into.
                symbols=['pair'],  # Columns to be inserted as SYMBOL types.
                at='timestamp')  # Column containing the designated timestamps.

    except IngressError as e:
        sys.stderr.write(f'Got error: {e}\n')


if __name__ == '__main__':
    example()
```

For more details see *Sender.dataframe* and for full argument options see *Buffer.dataframe*.

String vs Symbol Columns

QuestDB has a concept of symbols which are a more efficient way of storing categorical data (identifiers). Internally, symbols are deduplicated and stored as integers.

When sending data, you can specify a column as a symbol by using the `symbols` parameter of the `row` or `dataframe` methods.

Alternatively, if a column is expected to hold a collection of one-off strings, you can use the `strings` parameter.

Here is an example of sending a row with a symbol and a string:

```
from questdb.ingress import Sender, TimestampNanos
import datetime

conf = 'http::addr=localhost:9000;'
with Sender.from_conf(conf) as sender:
    sender.row(
        'news',
        symbols={
            'category': 'sport'},
        columns={
            'headline': 'The big game',
            'url': 'https://dailynews.com/sport/the-big-game',
            'views': 1000},
        at=datetime.datetime(2021, 1, 1, 12, 0, 0))
```

Populating Timestamps

The `at` parameter of the `row` and `dataframe` methods is used to specify the timestamp of the rows.

Set by client

It can be either a *`TimestampNanos`* object or a datetime.datetime object.

In case of dataframes you can also specify the timestamp column name or index. If so, the column type should be a Pandas `datetime64`, with or without timezone information.

Note that all timestamps in QuestDB are stored as microseconds since the epoch, without timezone information. Any timezone information is dropped when the data is appended to the ILP buffer.

Set by server

If you prefer, you can specify `at=ServerTimestamp` which will instruct QuestDB to set the timestamp on your behalf for each row as soon as it's received by the server.

```
from questdb.ingress import Sender, ServerTimestamp

conf = 'http::addr=localhost:9000;'
with Sender.from_conf(conf) as sender:
    sender.row(
        'weather_sensor',
        symbols={'id': 'toronto1'},
```

(continues on next page)

(continued from previous page)

```
        columns={'temperature': 23.5, 'humidity': 0.49},
        at=ServerTimestamp)  # Legacy feature, not recommended.
```

Warning: Using `ServerTimestamp` is not recommended as it removes the ability for QuestDB to deduplicate rows and is considered a *legacy feature*.

5.2.4 Flushing

The sender accumulates data into an internal buffer. Calling *`Sender.flush`* will send the buffered data to QuestDB, and clear the buffer.

Flushing can be done explicitly or automatically.

Explicit Flushing

An explicit call to *`Sender.flush`* will send any pending data immediately.

```
conf = 'http::addr=localhost:9000;'
with Sender.from_conf(conf) as sender:
    sender.row(
        'weather_sensor',
        symbols={'id': 'toronto1'},
        columns={'temperature': 23.5, 'humidity': 0.49},
        at=TimestampNanos.now())
    sender.flush()
    sender.row(
        'weather_sensor',
        symbols={'id': 'dubai2'},
        columns={'temperature': 41.2, 'humidity': 0.34},
        at=TimestampNanos.now())
    sender.flush()
```

Note that the last *sender.flush()* is entirely optional as flushing also happens at the end of the `with` block.

Auto-flushing

To avoid accumulating very large buffers, the sender will - by default - occasionally flush the buffer automatically.

Auto-flushing is triggered when:

- appending a row to the internal sender buffer
- and the buffer either:
  - Reaches 75'000 rows (for HTTP) or 600 rows (for TCP).
  - Hasn't been flushed for 1 second (there are no timers).

Here is an example *configuration string* that auto-flushes sets up a sender to flush every 10 rows and disables the interval-based auto-flushing logic.

```
http::addr=localhost:9000;auto_flush_rows=10;auto_flush_interval=off;
```

Here is a configuration string with auto-flushing completely disabled:

`http::addr=localhost:9000;auto_flush=off;`

See the *Auto-flushing* section for more details. and note that `auto_flush_interval` *does NOT start a timer*.

5.2.5 Error Reporting

TL;DR: Use HTTP for better error reporting

The sender will do its best to check for errors before sending data to the server.

When using the HTTP protocol, the server will send back an error message if the data is invalid or if there is a problem with the server. This will be raised as an *IngressError* exception.

The HTTP layer will also attempt retries, configurable via the *retry_timeout* parameter.`

When using the TCP protocol errors are *not* sent back from the server and must be searched for in the logs. See the *Errors during flushing* section for more details.

5.2.6 HTTP Transactions

When using the HTTP protocol, the sender can be configured to send a batch of rows as a single transaction.

Transactions are limited to a single table.

```
conf = 'http::addr=localhost:9000;'
with Sender.from_conf(conf) as sender:
    with sender.transaction('weather_sensor') as txn:
        txn.row(
            symbols={'id': 'toronto1'},
            columns={'temperature': 23.5, 'humidity': 0.49},
            at=TimestampNanos.now())
        txn.row(
            symbols={'id': 'dubai2'},
            columns={'temperature': 41.2, 'humidity': 0.34},
            at=TimestampNanos.now())
```

If auto-flushing is enabled, any pending data will be flushed before the transaction is started.

Auto-flushing is disabled during the scope of the transaction.

The transaction is automatically completed a the end of the `with` block.

- If the there are no errors, the transaction is committed and sent to the server without delays.
- If an exception is raised with the block, the transaction is rolled back and the exception is propagated.

You can also terminate a transaction explicity by calling the `commit` or the `rollback` methods.

While transactions that span multiple tables are not supported by QuestDB, you can reuse the same sender for mutliple tables.

You can also create parallel transactions by creating multiple sender objects across multiple threads.

5.2.7 Table and Column Auto-creation

When sending data to a table that does not exist, the server will create the table automatically.

This also applies to columns that do not exist.

The server will use the first row of data to determine the column types.

If the table already exists, the server will validate that the columns match the existing table.

If you're using QuestDB enterprise you might need to grant further permissions to the authenticated user.

```
CREATE SERVICE ACCOUNT ingest;
GRANT ilp, create table TO ingest;
GRANT add column, insert ON all tables TO ingest;
--  OR
GRANT add column, insert ON table1, table2 TO ingest;
```

Read more setup details in the Enterprise quickstart and the role-based access control guides.

5.2.8 Advanced Usage

Independent Buffers

All examples so far have shown appending data to the sender's internal buffer.

You can also create independent buffers and send them independently.

This is useful for more complex applications whishing to decouple the serialisation logic from the sending logic.

Note that the sender's auto-flushing logic will not apply to independent buffers.

```
from questdb.ingress import Buffer, Sender, TimestampNanos

buf = Buffer()
buf.row(
    'weather_sensor',
    symbols={'id': 'toronto1'},
    columns={'temperature': 23.5, 'humidity': 0.49},
    at=TimestampNanos.now())
buf.row(
    'weather_sensor',
    symbols={'id': 'dubai2'},
    columns={'temperature': 41.2, 'humidity': 0.34},
    at=TimestampNanos.now())

conf = 'http::addr=localhost:9000;'
with Sender.from_conf(conf) as sender:
    sender.flush(buf, transaction=True)
```

The `transaction` parameter is optional and defaults to `False`. When set to `True`, the buffer is guaranteed to be committed as a single transaction, but must only contain rows for a single table.

Multiple Databases

Handling buffers explicitly is also useful when sending data to multiple databases via the `.flush(buf, clear=False)` option.

```
from questdb.ingress import Buffer, Sender, TimestampNanos

buf = Buffer()
buf.row(
    'weather_sensor',
    symbols={'id': 'toronto1'},
    columns={'temperature': 23.5, 'humidity': 0.49},
    at=TimestampNanos.now())

conf1 = 'http::addr=db1.host.com:9000;'
conf2 = 'http::addr=db2.host.com:9000;'
with Sender.from_conf(conf1) as sender1, Sender.from_conf(conf2) as sender2:
    sender1.flush(buf1, clear=False)
    sender2.flush(buf2, clear=False)

buf.clear()
```

This uses the `clear=False` parameter which otherwise defaults to `True`.

Threading Considerations

Neither buffer API nor the sender object are thread-safe, but can be shared between threads if you take care of exclusive access (such as using a lock) yourself.

Independent buffers also allows you to prepare separate buffers in different threads and then send them later through a single exclusively locked sender.

Alternatively you can also create multiple senders, one per thread.

Notice that the `questdb` python module is mostly implemented in native code and is designed to release the Python GIL whenever possible, so you can expect good performance in multi-threaded scenarios.

As an example, appending a dataframe to a buffer releases the GIL (unless any of the columns reference python objects).

All network activity also fully releases the GIL.

Optimising HTTP Performance

The sender's network communication is implemented in native code and thus does not require access to the GIL, allowing for true parallelism when used using multiple threads.

For simplicity of design and best error feedback, the *.flush()* method blocks until the server has acknowledged the data.

If you need to send a large number of smaller requests (in other words, if you need to flush very frequently) or are in a high-latency network, you can significantly improve performance by creating and sending using multiple sender objects in parallel.

```
from questdb.ingress import Sender, TimestampNanos
import pandas as pd
from concurrent.futures import ThreadPoolExecutor
import datetime
```

(continues on next page)

(continued from previous page)

```
def send_data(df):
    conf_string = 'http::addr=localhost:9000;'
    with Sender.from_conf(conf_string) as sender:
        sender.dataframe(
            df,
            table_name='weather_sensor',
            symbols=['id'],
            at='timestamp')

dfs = [
    pd.DataFrame({
        'id': ['sensor1', 'sensor2'],
        'temperature': [22.5, 24.7],
        'humidity': [0.45, 0.47],
        'timestamp': [
            pd.Timestamp('2017-01-01T12:00:00'),
            pd.Timestamp('2017-01-01T12:00:01')
        ]}),
    pd.DataFrame({
        'id': ['sensor3', 'sensor4'],
        'temperature': [23.1, 25.3],
        'humidity': [0.48, 0.50],
        'timestamp': [
            pd.Timestamp('2017-01-01T12:00:02'),
            pd.Timestamp('2017-01-01T12:00:03')
        ]})
]

with ThreadPoolExecutor() as executor:
    futures = [executor.submit(send_data, df)
        for df in dfs]
    for future in futures:
        future.result()
```

For maxium performance you should also cache the sender objects and reuse them across multiple requests, since internally they maintain a connection pool.

Sender Lifetime Control

Instead of using a `with Sender .. as sender:` block you can also manually control the lifetime of the sender object.

```
from questdb.ingress import Sender

conf = 'http::addr=localhost:9000;'
sender = Sender.from_conf(conf)
sender.establish()
# ...
sender.close()
```

The *establish* method is needs to be called exactly once, but the *close* method is idempotent and can be called

multiple times.

5.2.9 Table and Column Names

The client will validate table and column names while constructing the buffer.

Table names and column names must not be empty and must adhere to the following:

Table Names

Cannot contain the following characters: `?`, `,`, `'`, `"`, `\`, `/`, `:`, `)`, `(`, `+`, `*`, `%`, `~`, carriage return (`\r`), newline (`\n`), null character (`\0`), and Unicode characters from `\u{0001}` to `\u{000F}` and `\u{007F}`. Additionally, the Unicode character for zero-width no-break space (UTF-8 BOM, `\u{FEFF}`) is not allowed.

A dot (`.`) is allowed except at the start or end of the name, and cannot be consecutive (e.g., `valid.name` is valid, but `.invalid`, `invalid.`, and `in..valid` are not).

Column Names

Cannot contain the following characters: `?`, `.`, `,`, `'`, `"`, `\`, `/`, `:`, `)`, `(`, `+`, `-`, `*`, `%`, `~`, carriage return (`\r`), newline (`\n`), null character (`\0`), and Unicode characters from `\u{0001}` to `\u{000F}` and `\u{007F}`. Like table names, the Unicode character for zero-width no-break space (UTF-8 BOM, `\u{FEFF}`) is not allowed.

Unlike table names, a dot (`.`) is not allowed in column names at all.

5.2.10 Programmatic Construction

Sender Constructor

You can also specify the configuration parameters programmatically:

```
from questdb.ingress import Sender, Protocol
from datetime import timedelta

with Sender(Protocol.Tcp, 'localhost', 9009,
        auto_flush=True,
        auto_flush_interval=timedelta(seconds=10)) as sender:
    ...
```

See the *Configuration* section for a full list of configuration parameters: each configuration parameter can be passed as named arguments to the constructor.

Python type mappings:

- Parameters that require strings take a `str`.
- Parameters that require numbers can also take an `int`.
- Millisecond durations can take an `int` or a `datetime.timedelta`.
- Any `'on'` / `'off'` / `'unsafe_off'` parameters can also be specified as a `bool`.
- Paths can also be specified as a `pathlib.Path`.

Note: The constructor arguments have changed between 1.x and 2.x. If you are upgrading, take a look at the *changelog*.

Customising `.from_conf()` and `.from_env()`

If you want to further customise the behaviour of the `.from_conf()` or `.from_env()` methods, you can pass additional parameters to these methods. The parameters are the same as the ones for the `Sender` constructor, as documented above.

For example, here is a *configuration string* that is loaded from an environment variable and then customised to specify a 10 second auto-flush interval:

```
export QDB_CLIENT_CONF='http::addr=localhost:9000;'
```

```
from questdb.ingress import Sender, Protocol
from datetime import timedelta

with Sender.from_env(auto_flush_interval=timedelta(seconds=10)) as sender:
    ...
```

5.2.11 ILP/TCP or ILP/HTTP

The sender supports `tcp`, `tcps`, `http`, and `https` protocols.

You should prefer to use the new ILP/HTTP protocol instead of ILP/TCP in most cases as it provides better feedback on errors and transaction control.

ILP/HTTP is available from:

- QuestDB 7.3.10 and later.
- QuestDB Enterprise 1.2.7 and later.

Since TCP does not block for a response it is useful for high-throughput scenarios in higher latency networks or on older versions of QuestDB which do not support ILP/HTTP quite yet.

It should be noted that you can achieve equivalent or better performance to TCP with HTTP by *using multiple sender objects in parallel*.

Either way, you can easily switch between the two protocols by changing:

- The `<protocol>` part of the *configuration string*.
- The port number (ILP/TCP default is 9009, ILP/HTTP default is 9000).
- Any *authentication parameters* such as `username`, `token`, et cetera.

5.3 Configuration

When constructing a *sender* you can pass a configuration string to the *Sender.from_conf* method.

```
from questdb.ingress import Sender

conf = "http::addr=localhost:9009;username=admin;password=quest;"
with Sender.from_conf(conf) as sender:
    ...
```

The format of the configuration string is:

```
<protocol>::<key>=<value>;<key>=<value>;...;
```

Note:

- The keys are case-sensitive.
- The trailing semicolon is mandatory.

The valid protocols are:

- `tcp`: ILP/TCP
- `tcps`: ILP/TCP with TLS
- `http`: ILP/HTTP
- `https`: ILP/HTTP with TLS

If you're unsure which protocol to use, see *ILP/TCP or ILP/HTTP*.

Only the `addr=host:port` key is mandatory. It specifies the hostname and port of the QuestDB server.

The same configuration string can also be loaded from the `QDB_CLIENT_CONF` environment variable. This is useful for keeping sensitive information out of your code.

```
export QDB_CLIENT_CONF="http::addr=localhost:9009;username=admin;password=quest;"
```

```
from questdb.ingress import Sender

with Sender.from_env() as sender:
    ...
```

5.3.1 Connection

- `addr` - `str`: The address of the server in the form of `host:port`.

  This key-value pair is mandatory, but the port can be defaulted. If omitted, the port will be defaulted to 9009 for TCP(s) and 9000 for HTTP(s).

- `bind_interface` - TCP-only, `str`: Network interface to bind from. Useful if you have an accelerated network interface (e.g. Solarflare) and want to use it.

  The default is `0.0.0.0`.

5.3.2 Authentication

If you're using QuestDB enterprise you can read up on creating and permissioning users in the Enterprise quickstart and the role-based access control guides.

HTTP Bearer Token

- `token` - `str`: Bearer token for HTTP authentication.

HTTP Basic Auth

- `username` - `str`: Username for HTTP basic authentication.
- `password` - `str`: Password for HTTP basic authentication.

TCP Auth

- `username` - `str`: Username for TCP authentication (A.K.A. *kid*).
- `token` - `str`: Token for TCP authentication (A.K.A. *d*).
- `token_x` - `str`: Token X for TCP authentication (A.K.A. *x*).
- `token_y` - `str`: Token Y for TCP authentication (A.K.A. *y*).

You can additionally set the `auth_timeout` parameter (milliseconds) to control how long the client will wait for a response from the server during the authentication process. The default is 15 seconds.

See the *TCP Authentication and TLS* example for more details.

5.3.3 TLS

TLS in enabled by selecting the `tcps` or `https` protocol.

See the QuestDB enterprise TLS documentation on how to enable this feature in the server.

Open source QuestDB does not offer TLS support out of the box, but you can still use TLS by setting up a proxy in front of QuestDB, such as *HAProxy <https://www.haproxy.org/>*.

- `tls_ca` - The remote server's certificate authority verification mechamism.
  - `'webpki_roots'`: Use the webpki-roots Rust crate to recognize certificates.
  - `'os_roots'`: Use the OS-provided certificate store.
  - `'webpki_and_os_roots'`: Use both the webpki-roots Rust crate and the OS-provided certificate store to recognize certificates.
  - `pem_file`: Path to a PEM-encoded certificate authority file. This is useful for testing with self-signed certificates.

  The default is: `'webpki_and_os_roots'`.
- `tls_roots` - `str`: Path to a PEM-encoded certificate authority file. When used it defaults the `tls_ca` to `'pem_file'`.

- `tls_verify` - `'on'` | `'unsafe_off'`: Whether to verify the server's certificate. This should only be used for testing as a last resort and never used in production as it makes the connection vulnerable to man-in-the-middle attacks.

  The default is: `'on'`.

As an example, if you are in a corporate environment and need to use the OS certificate store, you can use the following configuration string:

```
https::addr=localhost:9009;tls_ca=os_roots;
```

Alternatively, if you are testing with a self-signed certificate, you can use the following configuration string:

```
https::addr=localhost:9009;tls_roots=/path/to/cert.pem;
```

For more details on using self-signed test certificates, see:

- For Open Source QuestDB: https://github.com/questdb/c-questdb-client/blob/main/tls_certs/README.md#self-signed-certificates
- For QuestDB Enterprise: https://questdb.io/docs/operations/tls/#demo-certificates

5.3.4 Auto-flushing

The following parameters control the *Auto-flushing* behavior.

- `auto_flush` - `'on'` | `'off'`: Global switch for the auto-flushing behavior.

  Default: `'on'`.

- `auto_flush_rows` - `int > 0` | `'off'`: The number of rows that will trigger a flush. Set to `'off'` to disable.

  Default: 75000 (HTTP) | 600 (TCP).

- `auto_flush_bytes` - `int > 0` | `'off'`: The number of bytes that will trigger a flush. Set to `'off'` to disable.

  Default: `'off'`.

- `auto_flush_interval` - `int > 0` | `'off'`: The time in milliseconds that will trigger a flush. Set to `'off'` to disable.

  Default: 1000 (millis).

auto_flush_interval

The *auto_flush_interval* parameter controls how long the sender's buffer can be left unflushed for after appending a new row via the *Sender.row* or the *Sender.dataframe* methods. It is defined in milliseconds.

Note that this parameter does *not* create a timer that counts down each time data is added. Instead, the client checks the time elapsed since the last flush each time new data is added. If the elapsed time exceeds the specified `auto_flush_interval`, the client automatically flushes the current buffer to the database.

Consider the following example:

```
from questdb.ingress import Sender, TimestampNanos
import time
conf = "http::addr=localhost:9009;auto_flush_interval=1000;"
with Sender.from_conf(conf) as sender:
    # row 1
```

(continues on next page)

(continued from previous page)

```
    sender.row('table1', columns={'val': 1}, at=TimestampNanos.now())
    time.sleep(60)  # sleep for 1 minute
    # row 2
    sender.row('table2', columns={'val': 2}, at=TimestampNanos.now())
```

In this example above, "row 1" will not be flushed for a whole minute, until "row 2" is added and the `auto_flush_interval` limit of 1 second is exceeded, causing both "row 1" and "row 2" to be flushed together.

If you need consistent flushing at specific intervals, you should set `auto_flush_interval=off` and implement your own timer-based logic. The *Advanced Usage* documentation should help you.

5.3.5 Buffer

- `init_buf_size` - `int` > `0`: Initial buffer capacity.

  Default: 65536 (64KiB).

- `max_buf_size` - `int` > `0`: Maximum flushable buffer capacity.

  Default: 104857600 (100MiB).

- `max_name_len` - `int` > `0`: Maximum length of a table or column name.

  Default: 127.

5.3.6 HTTP Request

The following parameters control the HTTP request behavior.

- `retry_timeout` - `int` > `0`: The time in milliseconds to continue retrying after a failed HTTP request. The interval between retries is an exponential backoff starting at 10ms and doubling after each failed attempt up to a maximum of 1 second.

  Default: 10000 (10 seconds).

- `request_timeout` - `int` > `0`: The time in milliseconds to wait for a response from the server. This is in addition to the calculation derived from the `request_min_throughput` parameter.

  Default: 10000 (10 seconds).

- `request_min_throughput` - `int` > `0`: Minimum expected throughput in bytes per second for HTTP requests. If the throughput is lower than this value, the connection will time out. This is used to calculate an additional timeout on top of `request_timeout`. This is useful for large requests. You can set this value to `0` to disable this logic.

  Default: 102400 (100 KiB/s).

The final request timeout calculation is:

```
request_timeout + (buffer_size / request_min_throughput)
```

5.4 Examples

5.4.1 Basics

HTTP with Token Auth

The following example connects to the database and sends two rows (lines).

The connection is made via HTTPS and uses token based authentication.

The data is sent at the end of the `with` block.

```
from questdb.ingress import Sender, IngressError, TimestampNanos
import sys
import datetime


def example(host: str = 'localhost', port: int = 9009):
    try:
        conf = f'https::addr={host}:{port};token=the_secure_token;'
        with Sender.from_conf(conf) as sender:
            # Record with provided designated timestamp (using the 'at' param)
            # Notice the designated timestamp is expected in Nanoseconds,
            # but timestamps in other columns are expected in Microseconds.
            # The API provides convenient functions
            sender.row(
                'trades',
                symbols={
                    'pair': 'USDGBP',
                    'type': 'buy'},
                columns={
                    'traded_price': 0.83,
                    'limit_price': 0.84,
                    'qty': 100,
                    'traded_ts': datetime.datetime(
                        2022, 8, 6, 7, 35, 23, 189062,
                        tzinfo=datetime.timezone.utc)},
                at=TimestampNanos.now())

            # You can call `sender.row` multiple times inside the same `with`
            # block. The client will buffer the rows and send them in batches.

            # You can flush manually at any point.
            sender.flush()

            # If you don't flush manually, the client will flush automatically
            # when a row is added and either:
            #   * The buffer contains 75000 rows (if HTTP) or 600 rows (if TCP)
            #   * The last flush was more than 1000ms ago.
            # Auto-flushing can be customized via the `auto_flush_..` params.

        # Any remaining pending rows will be sent when the `with` block ends.
```

(continues on next page)

(continued from previous page)

```
    except IngressError as e:
        sys.stderr.write(f'Got error: {e}\n')


if __name__ == '__main__':
    example()
```

TCP Authentication and TLS

Continuing from the previous example, the connection is authenticated and also uses TLS.

```
from questdb.ingress import Sender, IngressError, TimestampNanos
import sys
import datetime


def example(host: str = 'localhost', port: int = 9009):
    try:
        conf = (
            f"tcps::addr={host}:{port};" +
            "username=testUser1;" +
            "token=5UjEMuA0Pj5pjK8a-fa24dyIf-Es5mYny3oE_Wmus48;" +
            "token_x=fLKYEaoEb9lrn3nkwLDA-M_xnuFOdSt9y0Z7_vWSHLU;" +
            "token_y=Dt5tbS1dEDMSYfym3fgMv0B99szno-dFc1rYF9t0aac;")
        with Sender.from_conf(conf) as sender:
            # Record with provided designated timestamp (using the 'at' param)
            # Notice the designated timestamp is expected in Nanoseconds,
            # but timestamps in other columns are expected in Microseconds.
            # The API provides convenient functions
            sender.row(
                'trades',
                symbols={
                    'pair': 'USDGBP',
                    'type': 'buy'},
                columns={
                    'traded_price': 0.83,
                    'limit_price': 0.84,
                    'qty': 100,
                    'traded_ts': datetime.datetime(
                        2022, 8, 6, 7, 35, 23, 189062,
                        tzinfo=datetime.timezone.utc)},
                at=TimestampNanos.now())

            # You can call `sender.row` multiple times inside the same `with`
            # block. The client will buffer the rows and send them in batches.

            # You can flush manually at any point.
            sender.flush()

            # If you don't flush manually, the client will flush automatically
            # when a row is added and either:
```

(continues on next page)

(continued from previous page)

```
            #   * The buffer contains 75000 rows (if HTTP) or 600 rows (if TCP)
            #   * The last flush was more than 1000ms ago.
            # Auto-flushing can be customized via the `auto_flush_..` params.

        # Any remaining pending rows will be sent when the `with` block ends.

    except IngressError as e:
        sys.stderr.write(f'Got error: {e}\n')


if __name__ == '__main__':
    example()
```

Explicit Buffers

For more *advanced use cases* where the same messages need to be sent to multiple questdb instances or you want to decouple serialization and sending (as may be in a multi-threaded application) construct *`Buffer`* objects explicitly, then pass them to the *`Sender.flush`* method.

Note that this bypasses *auto-flushing*.

```
from questdb.ingress import Sender, TimestampNanos


def example(host: str = 'localhost', port: int = 9000):
    with Sender.from_conf(f"http::addr={host}:{port};") as sender:
        buffer = sender.new_buffer()
        buffer.row(
            'line_sender_buffer_example',
            symbols={'id': 'Hola'},
            columns={'price': 111222233333, 'qty': 3.5},
            at=TimestampNanos(111222233333))
        buffer.row(
            'line_sender_example',
            symbols={'id': 'Adios'},
            columns={'price': 111222233343, 'qty': 2.5},
            at=TimestampNanos(111222233343))
        sender.flush(buffer)


if __name__ == '__main__':
    example()
```

Ticking Data and Auto-Flush

The following example somewhat mimics the behavior of a loop in an application.

It creates random ticking data at a random interval and uses non-default auto-flush settings.

```
from questdb.ingress import Sender, TimestampNanos
import random
import uuid
import time


def example(host: str = 'localhost', port: int = 9009):
    table_name: str = str(uuid.uuid1())
    conf: str = (
        f"tcp::addr={host}:{port};" +
        "auto_flush_bytes=1024;" +   # Flush if the internal buffer exceeds 1KiB
        "auto_flush_rows=off;"       # Disable auto-flushing based on row count
        "auto_flush_interval=5000;") # Flush if last flushed more than 5s ago
    with Sender.from_conf(conf) as sender:
        total_rows = 0
        try:
            print("Ctrl^C to terminate...")
            while True:
                time.sleep(random.randint(0, 750) / 1000)  # sleep up to 750 ms

                print('Inserting row...')
                sender.row(
                    table_name,
                    symbols={
                        'src': random.choice(('ALPHA', 'BETA', 'OMEGA')),
                        'dst': random.choice(('ALPHA', 'BETA', 'OMEGA'))},
                    columns={
                        'price': random.randint(200, 500),
                        'qty': random.randint(1, 5)},
                    at=TimestampNanos.now())
                total_rows += 1

                # If the internal buffer is empty, then auto-flush triggered.
                if len(sender) == 0:
                    print('Auto-flush triggered.')

        except KeyboardInterrupt:
            print(f"table: {table_name}, total rows sent: {total_rows}")
            print("bye!")


if __name__ == '__main__':
    example()
```

5.4.2 Data Frames

Pandas Basics

The following example shows how to insert data from a Pandas DataFrame to the `'trades'` table.

```
from questdb.ingress import Sender, IngressError

import sys
import pandas as pd


def example(host: str = 'localhost', port: int = 9000):
    df = pd.DataFrame({
            'pair': ['USDGBP', 'EURJPY'],
            'traded_price': [0.83, 142.62],
            'qty': [100, 400],
            'limit_price': [0.84, None],
            'timestamp': [
                pd.Timestamp('2022-08-06 07:35:23.189062', tz='UTC'),
                pd.Timestamp('2022-08-06 07:35:23.189062', tz='UTC')]})
    try:
        with Sender.from_conf(f"http::addr={host}:{port};") as sender:
            sender.dataframe(
                df,
                table_name='trades',  # Table name to insert into.
                symbols=['pair'],  # Columns to be inserted as SYMBOL types.
                at='timestamp')  # Column containing the designated timestamps.

    except IngressError as e:
        sys.stderr.write(f'Got error: {e}\n')


if __name__ == '__main__':
    example()
```

For details on all options, see the *Buffer.dataframe* method.

`pd.Categorical` and multiple tables

The next example shows some more advanced features inserting data from Pandas.

- The data is sent to multiple tables.
- It uses the `pd.Categorical` type to determine the table to insert and also uses it for the sensor name.
- Columns of type `pd.Categorical` are sent as `SYMBOL` types.
- The `at` parameter is specified using a column index: -1 is the last column.

```
from questdb.ingress import Sender, IngressError

import sys
import pandas as pd
```

(continues on next page)

(continued from previous page)

```
def example(host: str = 'localhost', port: int = 9000):
    df = pd.DataFrame({
            'metric': pd.Categorical(
                ['humidity', 'temp_c', 'voc_index', 'temp_c']),
            'sensor': pd.Categorical(
                ['paris-01', 'london-02', 'london-01', 'paris-01']),
            'value': [
                0.83, 22.62, 100.0, 23.62],
            'ts': [
                pd.Timestamp('2022-08-06 07:35:23.189062'),
                pd.Timestamp('2022-08-06 07:35:23.189062'),
                pd.Timestamp('2022-08-06 07:35:23.189062'),
                pd.Timestamp('2022-08-06 07:35:23.189062')]})
    try:
        with Sender.from_conf(f"http::addr={host}:{port};") as sender:
            sender.dataframe(
                df,
                table_name_col='metric',  # Table name from 'metric' column.
                symbols='auto',  # Category columns as SYMBOL. (Default)
                at=-1)  # Last column contains the designated timestamps.

    except IngressError as e:
        sys.stderr.write(f'Got error: {e}\n')


if __name__ == '__main__':
    example()
```

After running this example, the rows will be split across the `'humidity'`, `'temp_c'` and `'voc_index'` tables.

For details on all options, see the *Buffer.dataframe* method.

Loading Pandas from a Parquet File

The following example shows how to load a Pandas DataFrame from a Parquet file.

The example also relies on the dataframe's index name to determine the table name.

```
from questdb.ingress import Sender
import pandas as pd


def write_parquet_file():
    df = pd.DataFrame({
        'location': pd.Categorical(
            ['BP-5541', 'UB-3355', 'SL-0995', 'BP-6653']),
        'provider': pd.Categorical(
            ['BP Pulse', 'Ubitricity', 'Source London', 'BP Pulse']),
        'speed_kwh': pd.Categorical(
            [50, 7, 7, 120]),
        'connector_type': pd.Categorical(
```

(continues on next page)

(continued from previous page)

```
            ['Type 2 & 2+CCS', 'Type 1 & 2', 'Type 1 & 2', 'Type 2 & 2+CCS']),
        'current_type': pd.Categorical(
            ['dc', 'ac', 'ac', 'dc']),
        'price_pence':
            [54, 34, 32, 59],
        'in_use':
            [True, False, False, True],
        'ts': [
            pd.Timestamp('2022-12-30 12:15:00'),
            pd.Timestamp('2022-12-30 12:16:00'),
            pd.Timestamp('2022-12-30 12:18:00'),
            pd.Timestamp('2022-12-30 12:19:00')]})
    name = 'ev_chargers'
    df.index.name = name  # We set the dataframe's index name here!
    filename = f'{name}.parquet'
    df.to_parquet(filename)
    return filename


def example(host: str = 'localhost', port: int = 9000):
    filename = write_parquet_file()

    df = pd.read_parquet(filename)
    with Sender.from_conf(f"http::addr={host}:{port};") as sender:
        # Note: Table name is looked up from the dataframe's index name.
        sender.dataframe(df, at='ts')


if __name__ == '__main__':
    example()
```

For details on all options, see the *Buffer.dataframe* method.

5.5 API Reference

5.5.1 questdb.ingress

API for fast data ingestion into QuestDB.

class questdb.ingress.**Buffer**

Bases: object

Construct QuestDB-flavored InfluxDB Line Protocol (ILP) messages.

The *Buffer.row()* method is used to add a row to the buffer.

You can call this many times.

```
from questdb.ingress import Buffer

buf = Buffer()
buf.row(
```

(continues on next page)

(continued from previous page)

```
    'table_name1',
    symbols={'s1', 'v1', 's2', 'v2'},
    columns={'c1': True, 'c2': 0.5})

buf.row(
    'table_name2',
    symbols={'questdb': ''},
    columns={'like': 100000})

# Append any additional rows then, once ready, call
sender.flush(buffer)  # a `Sender` instance.

# The sender auto-cleared the buffer, ready for reuse.

buf.row(
    'table_name1',
    symbols={'s1', 'v1', 's2', 'v2'},
    columns={'c1': True, 'c2': 0.5})

# etc.
```

Buffer Constructor Arguments:

- `init_buf_size` (`int`): Initial capacity of the buffer in bytes. Defaults to `65536` (64KiB).
- `max_name_len` (`int`): Maximum length of a column name. Defaults to `127` which is the same default value as QuestDB. This should match the `cairo.max.file.name.length` setting of the QuestDB instance you're connecting to.

```
# These two buffer constructions are equivalent.
buf1 = Buffer()
buf2 = Buffer(init_buf_size=65536, max_name_len=127)
```

To avoid having to manually set these arguments every time, you can call the sender's `new_buffer()` method instead.

```
from questdb.ingress import Sender, Buffer

sender = Sender('http', 'localhost', 9009,
    init_buf_size=16384, max_name_len=64)
buf = sender.new_buffer()
assert buf.init_buf_size == 16384
assert buf.max_name_len == 64
```

__str__()

Return the constructed buffer as a string. Use for debugging.

capacity() → int

The current buffer capacity.

clear()

Reset the buffer.

Note that flushing a buffer will (unless otherwise specified) also automatically clear it.

This method is designed to be called only in conjunction with `sender.flush(buffer, clear=False)`.

dataframe(*df*, *, *table_name: str | None = None*, *table_name_col: None | int | str = None*, *symbols: str | bool | List[int] | List[str] = 'auto'*, *at: ServerTimestamp | int | str | TimestampNanos | datetime*)

Add a pandas DataFrame to the buffer.

Also see the *Sender.dataframe()* method if you're not using the buffer explicitly. It supports the same parameters and also supports auto-flushing.

This feature requires the `pandas`, `numpy` and `pyarrow` package to be installed.

Adding a dataframe can trigger auto-flushing behaviour, even between rows of the same dataframe. To avoid this, you can use HTTP and transactions (see *Sender.transaction()*).

Parameters

- **df** (*pandas.DataFrame*) – The pandas DataFrame to serialize to the buffer.
- **table_name** (*str or None*) – The name of the table to which the rows belong.

  If `None`, the table name is taken from the `table_name_col` parameter. If both `table_name` and `table_name_col` are `None`, the table name is taken from the DataFrame's index name (`df.index.name` attribute).
- **table_name_col** (*str or int or None*) – The name or index of the column in the DataFrame that contains the table name.

  If `None`, the table name is taken from the `table_name` parameter. If both `table_name` and `table_name_col` are `None`, the table name is taken from the DataFrame's index name (`df.index.name` attribute).

  If `table_name_col` is an integer, it is interpreted as the index of the column starting from `0`. The index of the column can be negative, in which case it is interpreted as an offset from the end of the DataFrame. E.g. `-1` is the last column.
- **symbols** (*str or bool or list of str or list of int*) – The columns to be serialized as symbols.

  If `'auto'` (default), all columns of dtype `'categorical'` are serialized as symbols. If `True`, all `str` columns are serialized as symbols. If `False`, no columns are serialized as symbols.

  The list of symbols can also be specified explicitly as a `list` of column names (`str`) or indices (`int`). Integer indices start at `0` and can be negative, offset from the end of the DataFrame. E.g. `-1` is the last column.

  Only columns containing strings can be serialized as symbols.
- **at** (*TimestampNanos, datetime.datetime, int or str or None*) – The designated timestamp of the rows.

  You can specify a single value for all rows or column name or index. If `ServerTimestamp`, timestamp is assigned by the server for all rows. To pass in a timestamp explicitly as an integer use the `TimestampNanos` wrapper type. To get the current timestamp, use `TimestampNanos.now()`. When passing a `datetime.datetime` object, the timestamp is converted to nanoseconds. A `datetime` object is assumed to be in the local timezone unless one is specified explicitly (so call `datetime.datetime.now(tz=datetime.timezone.utc)` instead of `datetime.datetime.utcnow()` for the current timestamp to avoid bugs).

  To specify a different timestamp for each row, pass in a column name (`str`) or index (`int`, 0-based index, negative index supported): In this case, the column needs to be of dtype `datetime64[ns]` (assumed to be in the **UTC timezone** and not local, due to differences

in Pandas and Python datetime handling) or `datetime64[ns, tz]`. When a timezone is specified in the column, it is converted to UTC automatically.

A timestamp column can also contain `None` values. The server will assign the current timestamp to those rows.

Note: All timestamps are always converted to nanoseconds and in the UTC timezone. Timezone information is dropped before sending and QuestDB will not store any timezone information.

Note: It is an error to specify both `table_name` and `table_name_col`.

Note: The "index" column of the DataFrame is never serialized, even if it is named.

Example:

```
import pandas as pd
import questdb.ingress as qi

buf = qi.Buffer()
# ...

df = pd.DataFrame({
    'location': ['London', 'Managua', 'London'],
    'temperature': [24.5, 35.0, 25.5],
    'humidity': [0.5, 0.6, 0.45],
    'ts': pd.date_range('2021-07-01', periods=3)})
buf.dataframe(
    df, table_name='weather', at='ts', symbols=['location'])

# ...
sender.flush(buf)
```

Pandas to ILP datatype mappings

See also:

https://questdb.io/docs/reference/api/ilp/columnset-types/

Table 1: Pandas Mappings

Pandas `dtype`	Nulls	ILP Datatype
`'bool'`	N	`BOOLEAN`
`'boolean'`	N	`BOOLEAN`
`'object'` (`bool` objects)	N	`BOOLEAN`
`'uint8'`	N	`INTEGER`
`'int8'`	N	`INTEGER`
`'uint16'`	N	`INTEGER`
`'int16'`	N	`INTEGER`
`'uint32'`	N	`INTEGER`
`'int32'`	N	`INTEGER`
`'uint64'`	N	`INTEGER`
`'int64'`	N	`INTEGER`
`'UInt8'`	Y	`INTEGER`
`'Int8'`	Y	`INTEGER`
`'UInt16'`	Y	`INTEGER`
`'Int16'`	Y	`INTEGER`
`'UInt32'`	Y	`INTEGER`
`'Int32'`	Y	`INTEGER`
`'UInt64'`	Y	`INTEGER`
`'Int64'`	Y	`INTEGER`
`'object'` (`int` objects)	Y	`INTEGER`
`'float32'`	Y (`NaN`)	`FLOAT`
`'float64'`	Y (`NaN`)	`FLOAT`
`'object'` (`float` objects)	Y (`NaN`)	`FLOAT`
`'string'` (`str` objects)	Y	`STRING` (default), `SYMBOL` via `symbols` arg.
`'string[pyarrow]'`	Y	`STRING` (default), `SYMBOL` via `symbols` arg.
`'category'` (`str` objects)	Y	`SYMBOL` (default), `STRING` via `symbols` arg.
`'object'` (`str` objects)	Y	`STRING` (default), `SYMBOL` via `symbols` arg.
`'datetime64[ns]'`	Y	`TIMESTAMP`
`'datetime64[ns, tz]'`	Y	`TIMESTAMP`

Note:

- : Note some pandas dtypes allow nulls (e.g. `'boolean'`), where the QuestDB database does not.
- : The valid range for integer values is -2^63 to 2^63-1. Any `'uint64'`, `'UInt64'` or python `int` object values outside this range will raise an error during serialization.
- : Upcast to 64-bit float during serialization.
- : Columns containing strings can also be used to specify the table name. See `table_name_col`.
- : We only support categories containing strings. If the category contains non-string values, an error will be raised.
- : The '.dataframe()' method only supports datetimes with nanosecond precision. The designated timestamp column (see `at` parameter) maintains the nanosecond precision, whilst values stored as columns have their precision truncated to microseconds. All dates are sent as UTC and any additional timezone information is dropped. If no timezone is specified, we follow the pandas convention of assuming the timezone is UTC. Datetimes before 1970-01-01 00:00:00 UTC are not supported. If a datetime value is specified as `None` (`NaT`), it is interpreted as the current QuestDB server time set on receipt of message.

Error Handling and Recovery

In case an exception is raised during dataframe serialization, the buffer is left in its previous state. The buffer remains in a valid state and can be used for further calls even after an error.

For clarification, as an example, if an invalid `None` value appears at the 3rd row for a `bool` column, neither the 3rd nor the preceding rows are added to the buffer.

Note: This differs from the *Sender.dataframe()* method, which modifies this guarantee due to its `auto_flush` logic.

Performance Considerations

The Python GIL is released during serialization if it is not needed. If any column requires the GIL, the entire serialization is done whilst holding the GIL.

Column types that require the GIL are:

- Columns of `str`, `float` or `int` or `float` Python objects.
- The `'string[python]'` dtype.

init_buf_size

The initial capacity of the buffer when first created.

This may grow over time, see `capacity()`.

max_name_len

Maximum length of a table or column name.

reserve(*additional: int*)

Ensure the buffer has at least *additional* bytes of future capacity.

Parameters
additional (*int*) – Additional bytes to reserve.

row(*table_name: str*, *, *symbols: Dict[str, str | None] | None = None*, *columns: Dict[str, None | bool | int | float | str | TimestampMicros | datetime] | None = None*, *at: ServerTimestamp | TimestampNanos | datetime*)

Add a single row (line) to the buffer.

```
# All fields specified.
buffer.row(
    'table_name',
    symbols={'sym1': 'abc', 'sym2': 'def', 'sym3': None},
    columns={
        'col1': True,
        'col2': 123,
        'col3': 3.14,
        'col4': 'xyz',
        'col5': TimestampMicros(123456789),
        'col6': datetime(2019, 1, 1, 12, 0, 0),
        'col7': None},
    at=TimestampNanos(123456789))

# Only symbols specified. Designated timestamp assigned by the db.
buffer.row(
    'table_name',
    symbols={'sym1': 'abc', 'sym2': 'def'}, at=Server.Timestamp)
```

(continues on next page)

(continued from previous page)

```
# Float columns and timestamp specified as `datetime.datetime`.
# Pay special attention to the timezone, which if unspecified is
# assumed to be the local timezone (and not UTC).
buffer.row(
    'sensor data',
    columns={
        'temperature': 24.5,
        'humidity': 0.5},
    at=datetime.datetime.now(tz=datetime.timezone.utc))
```

Python strings passed as values to `symbols` are going to be encoded as the `SYMBOL` type in QuestDB, whilst Python strings passed as values to `columns` are going to be encoded as the `STRING` type.

Refer to the QuestDB documentation to understand the difference between the `SYMBOL` and `STRING` types (TL;DR: symbols are interned strings).

Column values can be specified with Python types directly and map as so:

Python type	Serialized as ILP type
`bool`	BOOLEAN
`int`	INTEGER
`float`	FLOAT
`str`	STRING
`datetime.datetime` and `TimestampMicros`	TIMESTAMP
`None`	*Column is skipped and not serialized.*

If the destination table was already created, then the columns types will be cast to the types of the existing columns whenever possible (Refer to the QuestDB documentation pages linked above).

Adding a row can trigger auto-flushing behaviour.

Parameters

- **table_name** – The name of the table to which the row belongs.
- **symbols** – A dictionary of symbol column names to `str` values. As a convenience, you can also pass a `None` value which will have the same effect as skipping the key: If the column already existed, it will be recorded as `NULL`, otherwise it will not be created.
- **columns** – A dictionary of column names to `bool`, `int`, `float`, `str`, `TimestampMicros` or `datetime` values. As a convenience, you can also pass a `None` value which will have the same effect as skipping the key: If the column already existed, it will be recorded as `NULL`, otherwise it will not be created.
- **at** – The timestamp of the row. This is required! If `ServerTimestamp`, timestamp is assigned by QuestDB. If `datetime`, the timestamp is converted to nanoseconds. A nanosecond unix epoch timestamp can be passed explicitly as a `TimestampNanos` object.

exception questdb.ingress.**IngressError**(*code*, *msg*)

Bases: `Exception`

An error whilst using the `Sender` or constructing its `Buffer`.

__init__(*code*, *msg*)

property code: *IngressErrorCode*

Return the error code.

class questdb.ingress.**IngressErrorCode**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

Category of Error.

AuthError = 6

BadDataFrame = 10

ConfigError = 10

CouldNotResolveAddr = 0

HttpNotSupported = 8

InvalidApiCall = 1

InvalidName = 4

InvalidTimestamp = 5

InvalidUtf8 = 3

ServerFlushError = 9

SocketError = 2

TlsError = 7

class questdb.ingress.**Protocol**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: TaggedEnum

Protocol to use for sending data to QuestDB.

See *ILP/TCP or ILP/HTTP* for more information.

Http = ('http', 2)

Https = ('https', 3)

Tcp = ('tcp', 0)

Tcps = ('tcps', 1)

property tls_enabled

class questdb.ingress.**Sender**

Bases: object

Ingest data into QuestDB.

See the *Sending Data over ILP* documentation for more information.

__enter__() → *Sender*

Call *Sender.establish()* at the start of a with block.

__exit__(*exc_type*, *_exc_val*, *_exc_tb*)

Flush pending and disconnect at the end of a `with` block.

If the `with` block raises an exception, any pending data will *NOT* be flushed.

This is implemented by calling *Sender.close()*.

__init__(**args*, ***kwargs*)

__str__()

Inspect the contents of the internal buffer.

The `str` value returned represents the unsent data.

Also see `Sender.__len__()`.

auto_flush

Auto-flushing is enabled.

Consult the *.auto_flush_rows*, *.auto_flush_bytes* and *.auto_flush_interval* properties for the current active thresholds.

auto_flush_bytes

Byte-count threshold for the auto-flush logic, or None if disabled.

auto_flush_interval

Time interval threshold for the auto-flush logic, or None if disabled.

auto_flush_rows

Row count threshold for the auto-flush logic, or None if disabled.

close(*flush=True*)

Disconnect.

This method is idempotent and can be called repeatedly.

Once a sender is closed, it can't be re-used.

Parameters

flush (*bool*) – If `True`, flush the internal buffer before closing.

dataframe(*df*, ***, *table_name: str | None = None*, *table_name_col: None | int | str = None*, *symbols: str | bool | List[int] | List[str] = 'auto'*, *at: ServerTimestamp | int | str | TimestampNanos | datetime*)

Write a Pandas DataFrame to the internal buffer.

Example:

```
import pandas as pd
import questdb.ingress as qi

df = pd.DataFrame({
    'car': pd.Categorical(['Nic 42', 'Eddi', 'Nic 42', 'Eddi']),
    'position': [1, 2, 1, 2],
    'speed': [89.3, 98.2, 3, 4],
    'lat_gforce': [0.1, -0.2, -0.6, 0.4],
    'accelleration': [0.1, -0.2, 0.6, 4.4],
    'tyre_pressure': [2.6, 2.5, 2.6, 2.5],
    'ts': [
        pd.Timestamp('2022-08-09 13:56:00'),
        pd.Timestamp('2022-08-09 13:56:01'),
```

(continues on next page)

(continued from previous page)

```
        pd.Timestamp('2022-08-09 13:56:02'),
        pd.Timestamp('2022-08-09 13:56:03')]})

with qi.Sender.from_env() as sender:
    sender.dataframe(df, table_name='race_metrics', at='ts')
```

This method builds on top of the *Buffer.dataframe()* method. See its documentation for details on arguments.

Additionally, this method also supports auto-flushing the buffer as specified in the `Sender`'s `auto_flush` constructor argument. Auto-flushing is implemented incrementally, meaning that when calling `sender.dataframe(df)` with a large `df`, the sender may have sent some of the rows to the server already whist the rest of the rows are going to be sent at the next auto-flush or next explicit call to *Sender.flush()*.

In case of data errors with auto-flushing enabled, some of the rows may have been transmitted to the server already.

establish()

Prepare the sender for use.

If using ILP/HTTP this will initialize the HTTP connection pool.

If using ILP/TCP this will cause connection to the server and block until the connection is established.

If the TCP connection is set up with authentication and/or TLS, this method will return only *after* the handshake(s) is/are complete.

flush(*buffer=None*, *clear=True*, *transactional=False*)

If called with no arguments, immediately flushes the internal buffer.

Alternatively you can flush a buffer that was constructed explicitly by passing `buffer`.

The buffer will be cleared by default, unless `clear` is set to `False`.

This method does nothing if the provided or internal buffer is empty.

Parameters

- **buffer** – The buffer to flush. If `None`, the internal buffer is flushed.
- **clear** – If `True`, the flushed buffer is cleared (default). If `False`, the flushed buffer is left in the internal buffer. Note that `clear=False` is only supported if `buffer` is also specified.
- **transactional** – If `True` ensures that the flushed buffer contains row for a single table, ensuring all data can be written transactionally. This feature requires ILP/HTTP and is not available when connecting over TCP. *Default: False.*

The Python GIL is released during the network IO operation.

static from_conf(*conf_str*, ***, *bind_interface=None*, *username=None*, *password=None*, *token=None*, *token_x=None*, *token_y=None*, *auth_timeout=None*, *tls_verify=None*, *tls_ca=None*, *tls_roots=None*, *max_buf_size=None*, *retry_timeout=None*, *request_min_throughput=None*, *request_timeout=None*, *auto_flush=None*, *auto_flush_rows=None*, *auto_flush_bytes=None*, *auto_flush_interval=None*, *init_buf_size=None*, *max_name_len=None*)

Construct a sender from a *configuration string*.

The additional arguments are used to specify additional parameters which are not present in the configuration string.

Note that any parameters already present in the configuration string cannot be overridden.

static from_env(*, *bind_interface=None*, *username=None*, *password=None*, *token=None*, *token_x=None*, *token_y=None*, *auth_timeout=None*, *tls_verify=None*, *tls_ca=None*, *tls_roots=None*, *max_buf_size=None*, *retry_timeout=None*, *request_min_throughput=None*, *request_timeout=None*, *auto_flush=None*, *auto_flush_rows=None*, *auto_flush_bytes=None*, *auto_flush_interval=None*, *init_buf_size=None*, *max_name_len=None*)

Construct a sender from the `QDB_CLIENT_CONF` environment variable.

The environment variable must be set to a valid *configuration string*.

The additional arguments are used to specify additional parameters which are not present in the configuration string.

Note that any parameters already present in the configuration string cannot be overridden.

init_buf_size

The initial capacity of the sender's internal buffer.

max_name_len

Maximum length of a table or column name.

new_buffer()

Make a new configured buffer.

The buffer is set up with the configured *init_buf_size* and *max_name_len*.

row(*table_name: str*, *, *symbols: Dict[str, str] | None = None*, *columns: Dict[str, bool | int | float | str | TimestampMicros | datetime] | None = None*, *at: TimestampNanos | datetime | ServerTimestamp*)

Write a row to the internal buffer.

This may be sent automatically depending on the `auto_flush` setting in the constructor.

Refer to the *Buffer.row()* documentation for details on arguments.

transaction(*table_name: str*)

Start a *HTTP Transactions* block.

class questdb.ingress.**TimestampMicros**

Bases: `object`

A timestamp in microseconds since the UNIX epoch (UTC).

You may construct a `TimestampMicros` from an integer or a `datetime.datetime`, or simply call the *TimestampMicros.now()* method.

```
# Recommended way to get the current timestamp.
TimestampMicros.now()

# The above is equivalent to:
TimestampMicros(time.time_ns() // 1000)

# You can provide a numeric timestamp too. It can't be negative.
TimestampMicros(1657888365426838)
```

`TimestampMicros` can also be constructed from a `datetime.datetime` object.

```
TimestampMicros.from_datetime(
    datetime.datetime.now(tz=datetime.timezone.utc))
```

We recommend that when using `datetime` objects, you explicitly pass in the timezone to use. This is because `datetime` objects without an associated timezone are assumed to be in the local timezone and it is easy to make mistakes (e.g. passing `datetime.datetime.utcnow()` is a likely bug).

classmethod from_datetime(*dt: datetime*)

Construct a `TimestampMicros` from a `datetime.datetime` object.

classmethod now()

Construct a `TimestampMicros` from the current time as UTC.

value

Number of microseconds (Unix epoch timestamp, UTC).

class questdb.ingress.**TimestampNanos**

Bases: `object`

A timestamp in nanoseconds since the UNIX epoch (UTC).

You may construct a `TimestampNanos` from an integer or a `datetime.datetime`, or simply call the *TimestampNanos.now()* method.

```
# Recommended way to get the current timestamp.
TimestampNanos.now()

# The above is equivalent to:
TimestampNanos(time.time_ns())

# You can provide a numeric timestamp too. It can't be negative.
TimestampNanos(1657888365426838016)
```

`TimestampNanos` can also be constructed from a `datetime` object.

```
TimestampNanos.from_datetime(
    datetime.datetime.now(tz=datetime.timezone.utc))
```

We recommend that when using `datetime` objects, you explicitly pass in the timezone to use. This is because `datetime` objects without an associated timezone are assumed to be in the local timezone and it is easy to make mistakes (e.g. passing `datetime.datetime.utcnow()` is a likely bug).

classmethod from_datetime(*dt: datetime*)

Construct a `TimestampNanos` from a `datetime.datetime` object.

classmethod now()

Construct a `TimestampNanos` from the current time as UTC.

value

Number of nanoseconds (Unix epoch timestamp, UTC).

class questdb.ingress.**TlsCa**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: `TaggedEnum`

Verification mechanism for the server's certificate.

Here `webpki` refers to the WebPKI library and `os` refers to the operating system's certificate store.

See *TLS* for more information.

OsRoots = ('os_roots', 1)

```
PemFile = ('pem_file', 3)

WebpkiAndOsRoots = ('webpki_and_os_roots', 2)

WebpkiRoots = ('webpki_roots', 0)
```

5.6 Troubleshooting

5.6.1 Common issues

You may be experiencing one of the issues below.

Production-optimized QuestDB configuration

If you can't initially see your data through a `select` SQL query straight away, this is normal: by default the database will only commit data it receives though the line protocol periodically to maximize throughput.

For dev/testing you may want to tune the following database configuration parameters as so:

```
# server.conf
cairo.max.uncommitted.rows=1
line.tcp.maintenance.job.interval=100
```

The default QuestDB configuration is more applicable for a production environment.

For these and more configuration parameters refer to database configuration documentation.

Infrequent Flushing

You may not see data appear in a timely manner because you're not calling *`flush`* often enough.

You might be having issues with the *`Sender`*'s *auto-flush* feature.

Errors during flushing

ILP/TCP Server disconnects

If you're using TCP instead of HTTP, you may see a server disconnect after flushing.

If the server receives invalid data over ILP/TCP it will drop the connection.

The ILP/TCP protocol does not send errors back to the client. Instead, by design, it will disconnect a client if it encounters any insertion errors. This is to avoid errors going unnoticed.

As an example, if a client were to insert a `STRING` value into a `BOOLEAN` column, the QuestDB server would disconnect the client.

To determine the root cause of a disconnect, inspect the server logs.

Note: For a better developer experience consider using *HTTP instead of TCP*.

Logging outgoing messages

To understand what data was sent to the server, you may log outgoing messages from Python.

Here's an example if you append rows to the `Sender` object:

```
import textwrap

with Sender.from_conf(...) as sender:
    # sender.row(...)
    # sender.row(...)
    # ...
    pending = str(sender)
    logging.info('About to flush:\n%s', textwrap.indent(pending, '    '))
    sender.flush()
```

Alternatively, if you're constructing buffers explicitly:

```
import textwrap

buffer = sender.new_buffer()
# buffer.row(...)
# buffer.row(...)
# ...
pending = str(buffer)
logging.info('About to flush:\n%s', textwrap.indent(pending, '    '))
sender.flush(buffer)
```

Note that to handle out-of-order messages efficiently, the QuestDB server will delay appling changes it receives over ILP after a configurable commit lag.

Due to this commit lag, the line that caused the error may not be the last line.

5.6.2 Asking for help

The best way to get help is through Slack.

5.7 Community

5.7.1 Contributions

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

5.7.2 Bug reports

When reporting a bug please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

5.7.3 Feature requests and feedback

The best way to send feedback is to file an issue through GitHub.

5.7.4 Development

To get set up for local development follow the development notes.

5.7.5 Chat to us

If you want to engage with us to discuss your changes or if you need help, there's a *#contributors* channel on our slack server just for that.

5.7.6 Asking for help

The best thing to do is to ask on Stack Overflow. We monitor the *questdb* tag and will get back to you.

Stack overflow helps us to keep track of the questions and answers, and it helps other users who might have the same question.

Alternatively, you can ask on our slack server.

5.8 Changelog

5.8.1 2.0.2 (2024-04-11)

Patch release with performance bug fix. No breaking changes.

Bug fixes

- Fixed the defaulting logic for `auto_flush_rows` parameter for HTTPS. It is now correctly set to 75000 rows by default. The old incorrect default of 600 rows was causing the sender to flush too often, impacting performance. Note that TCP, TCPS and HTTP were not affected.

Features

- The sender now exposes the `auto_flush` settings as read-only properties. You can inspect the values in use with `.auto_flush`, `.auto_flush_rows`, `.auto_flush_interval` and `.auto_flush_bytes`.

5.8.2 2.0.1 (2024-04-03)

Patch release with bug fixes, no API changes and some documentation tweaks.

Bug fixes

- Fixed a bug where an internal "last flushed" timestamp used by `auto_flush_interval` wasn't updated correctly causing the auto-flush logic to trigger after each row.
- Removed two unnecessary debugging `print()` statements that were accidentally left in the code in `Sender.from_conf()` and `Sender.from_env()`.

Documentation

- Introduced the ability to optionally install `pandas` and `pyarrow` via `python3 -m pip install -U questdb[dataframe]` and updated the documentation to reflect this.

5.8.3 2.0.0 (2024-03-19)

This is a major release with new features and breaking changes.

Features

- Support for ILP over HTTP. The sender can now send data to QuestDB via HTTP instead of TCP. This provides error feedback from the server and new features.

```
conf = 'http::addr=localhost:9000;'
with Sender.from_conf(conf) as sender:
    sender.row(...)
    sender.dataframe(...)

    # Will raise `IngressError` if there is an error from the server.
    sender.flush()
```

- New configuration string construction. The sender can now be also constructed from a *configuration string* in addition to the constructor arguments. This allows for more flexible configuration and is the recommended way to construct a sender. The same string can also be loaded from the `QDB_CLIENT_CONF` environment variable. The constructor arguments have been updated and some options have changed.
- Explicit transaction support over HTTP. A set of rows for a single table can now be committed via the sender transactionally. You can do this using a `with sender.transaction('table_name') as txn:` block.

```
conf = 'http::addr=localhost:9000;'
with Sender.from_conf(conf) as sender:
    with sender.transaction('test_table') as txn:
        # Same arguments as the sender methods, minus the table name.
```

(continues on next page)

(continued from previous page)

```
        txn.row(...)
        txn.dataframe(...)
```

- A number of documentation improvements.

Breaking Changes

- New `protocol` parameter in the *Sender* constructor.

  In previous version the protocol was always TCP. In this new version you must specify the protocol explicitly.

- New auto-flush defaults. In previous versions *auto-flushing* was enabled by default and triggered by a maximum buffer size. In this new version auto-flushing is enabled by row count (600 rows by default) and interval (1 second by default), while auto-flushing by buffer size is disabled by default.

  The old behaviour can be still be achieved by tweaking the auto-flush settings.

Setting	Old default	New default
auto_flush_rows	off	600
auto_flush_interval	off	1000
auto_flush_bytes	64512	off

- The `at=..` argument of *row* and *dataframe* methods is now mandatory. Omitting it would previously use a server-generated timestamp for the row. Now if you want a server generated timestamp, you can pass the *ServerTimestamp* singleton to this parameter. _The `ServerTimestamp` behaviour is considered legacy._
- The `auth=(u, t, x, y)` argument of the `Sender` constructor has now been broken up into multiple arguments: `username`, `token`, `token_x`, `token_y`.
- The `tls` argument of the `Sender` constructor has been removed and replaced with the `protocol` argument. Use `Protocol.Tcps` (or `Protocol.Https`) to enable TLS. The `tls` values have been moved to new `tls_ca` and `tls_roots` *configuration settings*.
- The `net_interface` argument of the `Sender` constructor has been renamed to `bind_interface` and is now only available for TCP connections.

The following example shows how to migrate to the new API.

Old questdb 1.x code

```
from questdb.ingress import Sender

auth = (
    'testUser1',
    '5UjEMuA0Pj5pjK8a-fa24dyIf-Es5mYny3oE_Wmus48',
    'token_x=fLKYEaoEb9lrn3nkwLDA-M_xnuFOdSt9y0Z7_vWSHLU',
    'token_y=Dt5tbS1dEDMSYfym3fgMv0B99szno-dFc1rYF9t0aac')
with Sender('localhost', 9009, auth=auth, tls=True) as sender:
    sender.row(
        'test_table',
        symbols={'sym': 'AAPL'},
        columns={'price': 100.0})  # `at=None` was defaulted for server time
```

Equivalent questdb 2.x code

```
from questdb.ingress import Sender, Protocol, ServerTimestamp

sender = Sender(
    Protocol.Tcps,
    'localhost',
    9009,
    username='testUser1',
    token='5UjEMuA0Pj5pjK8a-fa24dyIf-Es5mYny3oE_Wmus48',
    token_x='token_x=fLKYEaoEb9lrn3nkwLDA-M_xnuFOdSt9y0Z7_vWSHLU',
    token_y='token_y=Dt5tbS1dEDMSYfym3fgMv0B99szno-dFc1rYF9t0aac',
    auto_flush_rows='off',
    auto_flush_interval='off',
    auto_flush_bytes=64512)
with sender:
    sender.row(
        'test_table',
        symbols={'sym': 'AAPL'},
        columns={'price': 100.0},
        at=ServerTimestamp)
```

Equivalent questdb 2.x code with configuration string

```
from questdb.ingress import Sender

conf = (
    'tcp::addr=localhost:9009;' +
    'username=testUser1;' +
    'token=5UjEMuA0Pj5pjK8a-fa24dyIf-Es5mYny3oE_Wmus48;' +
    'token_x=token_x=fLKYEaoEb9lrn3nkwLDA-M_xnuFOdSt9y0Z7_vWSHLU;' +
    'token_y=token_y=Dt5tbS1dEDMSYfym3fgMv0B99szno-dFc1rYF9t0aac;' +
    'auto_flush_rows=off;' +
    'auto_flush_interval=off;' +
    'auto_flush_bytes=64512;')
with Sender.from_conf(conf) as sender:
    sender.row(
        'test_table',
        symbols={'sym': 'AAPL'},
        columns={'price': 100.0},
        at=ServerTimestamp)
```

5.8.4 1.2.0 (2023-11-23)

This is a minor release bringing in minor new features and a few bug fixes, without any breaking changes.

Most changes are inherited by internally upgrading to version `3.1.0` of the `c-questdb-client`.

Features

- `Sender(..., tls=True)` now also uses the OS-provided certificate store. The *tls* argument can now also be set to `tls='os_roots'` (to *only* use the OS-provided certs) or `tls='webpki_roots'` (to *only* use the certs provided by the `webpki-roots`, i.e. the old behaviour prior to this release). The new default behaviour for `tls=True` is equivalent to setting `tls='webpki_and_os_roots'`.
- Upgraded dependencies to newer library versions. This also includes the latest webpki-roots crate providing updated TLS CA certificate roots.
- Various example code and documentation improvements.

Bug fixes

- Fixed a bug where timestamp columns could not accept values before Jan 1st 1970 UTC.
- TCP connections now enable `SO_KEEPALIVE`: This should ensure that connections don't drop after a period of inactivity.

5.8.5 1.1.0 (2023-01-04)

Features

- High-performance ingestion of Pandas dataframes into QuestDB via ILP. We now support most Pandas column types. The logic is implemented in native code and is orders of magnitude faster than iterating the dataframe in Python and calling the `Buffer.row()` or `Sender.row()` methods: The `Buffer` can be written from Pandas at hundreds of MiB/s per CPU core. The new `dataframe()` method continues working with the `auto_flush` feature. See API documentation and examples for the new `dataframe()` method available on both the `Sender` and `Buffer` classes.
- New `TimestampNanos.now()` and `TimestampMicros.now()` methods. *These are the new recommended way of getting the current timestamp.*
- The Python GIL is now released during calls to `Sender.flush()` and when `auto_flush` is triggered. This should improve throughput when using the `Sender` from multiple threads.

Errata

- In previous releases the documentation for the `from_datetime()` methods of the `TimestampNanos` and `TimestampMicros` types recommended calling `datetime.datetime.utcnow()` to get the current timestamp. This is incorrect as it will (confusinly) return object with the local timezone instead of UTC. This documentation has been corrected and now recommends calling `datetime.datetime.now(tz=datetime.timezone.utc)` or (more efficiently) the new `TimestampNanos.now()` and `TimestampMicros.now()` methods.

5.8.6 1.0.2 (2022-10-31)

Features

- Support for Python 3.11.
- Updated to version 2.1.1 of the `c-questdb-client` library:
  - Setting `SO_REUSEADDR` on outbound socket. This is helpful to users with large number of connections who previously ran out of outbound network ports.

5.8.7 1.0.1 (2022-08-16)

Features

- As a matter of convenience, the `Buffer.row` method can now take `None` column values. This has the same semantics as skipping the column altogether. Closes #3.

Bug fixes

- Fixed a major bug where Python `int` and `float` types were handled with 32-bit instead of 64-bit precision. This caused certain `int` values to be rejected and other `float` values to be rounded incorrectly. Closes #13.
- Fixed a minor bug where an error auto-flush caused a second clean-up error. Closes #4.

5.8.8 1.0.0 (2022-07-15)

Features

- First stable release.
- Insert data into QuestDB via ILP.
- Sender and Buffer APIs.
- Authentication and TLS support.
- Auto-flushing of buffers.

5.8.9 0.0.3 (2022-07-14)

Features

- Initial set of features to connect to the database.
- `Buffer` and `Sender` classes.
- First release where `pip install questdb` should work.

5.8.10 0.0.1 (2022-07-08)

Features

- First release on PyPI.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

Symbols

A

B

C

D

E

F

H

I

M

N

O

P

Q

R

S

T

V

W