questdb

Release 1.1.0

QuestDB

Feb 20, 2023

CONTENTS

This library makes it easy to insert data into QuestDB.

This client library implements QuestDB's variant of the InfluxDB Line Protocol (ILP) over 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 1.1.0.

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

```
from questdb.ingress import Sender

with Sender('localhost', 9009) as sender:
    sender.row(
        'sensors',
        symbols={'id': 'toronto1'},
        columns={'temperature': 20.0, 'humidity': 0.5})
    sender.flush()
```

You can also send Pandas dataframes:

```
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'])'})

with Sender('localhost', 9009) as sender:
    sender.dataframe(df, table_name='sensors')
```

CHAPTER

TWO

DOCS

https://py-questdb-client.readthedocs.io/

CHAPTER

THREE

CODE

https://github.com/questdb/py-questdb-client

CHAPTER

FOUR

PACKAGE ON PYPI

https://pypi.org/project/questdb/

CHAPTER

FIVE

COMMUNITY

If you need help, have additional questions or want to provide feedback, you may find us on Slack.

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

CHAPTER

SIX

LICENSE

The code is released under the Apache License 2.0.

CHAPTER

SEVEN

CONTENTS

7.1 Installation

The Python QuestDB client does not have any additional run-time dependencies and will run on any version of Python >= 3.7 on most platforms and architectures.

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

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

Or, from within a virtual environment:

```
pip install questdb
```

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

```
poetry add questdb
```

Note that the *questdb.ingress.Buffer.dataframe()* and the *questdb.ingress.Sender.dataframe()* methods also require the following dependencies to be installed:

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

7.1.1 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'
```

7.2 Examples

7.2.1 Basics

Row-by-row Insertion

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

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

Here the *questdb.ingress.Sender* is constructed with just `host` and `port`.

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


def example(host: str = 'localhost', port: int = 9009):
    try:
        with Sender(host, port) 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())

            # If no 'at' param is passed, the server will use its own timestamp.
            sender.row(
                'trades',
                symbols={'pair': 'EURJPY'},
                columns={
```

(continues on next page)

(continued from previous page)

```
                    'traded_price': 135.97,
                    'qty': 400,
                    'limit_price': None})  # NULL columns can be passed as None,
                                           # or simply be left out.

            # We recommend flushing periodically, for example every few seconds.
            # If you don't flush explicitly, the client will flush automatically
            # once the buffer is reaches 63KiB and just before the connection
            # is closed.
            sender.flush()

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


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

Authentication and TLS

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

Here the *questdb.ingress.Sender* is also constructed with the `auth` and `tls` arguments.

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


def example(host: str = 'localhost', port: int = 9009):
    try:
        # See: https://questdb.io/docs/reference/api/ilp/authenticate
        auth = (
            "testUser1",                                    # kid
            "5UjEMuA0Pj5pjK8a-fa24dyIf-Es5mYny3oE_Wmus48",  # d
            "fLKYEaoEb9lrn3nkwLDA-M_xnuFOdSt9y0Z7_vWSHLU",  # x
            "Dt5tbS1dEDMSYfym3fgMv0B99szno-dFc1rYF9t0aac")  # y
        with Sender(host, port, auth=auth, tls=True) 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,
```

(continues on next page)

(continued from previous page)

```
                    'traded_ts': datetime.datetime(
                        2022, 8, 6, 7, 35, 23, 189062,
                        tzinfo=datetime.timezone.utc)},
                at=TimestampNanos.now())

            # If no 'at' param is passed, the server will use its own timestamp.
            sender.row(
                'trades',
                symbols={'pair': 'EURJPY'},
                columns={
                    'traded_price': 135.97,
                    'qty': 400,
                    'limit_price': None})  # NULL columns can be passed as None,
                                           # or simply be left out.

            # We recommend flushing periodically, for example every few seconds.
            # If you don't flush explicitly, the client will flush automatically
            # once the buffer is reaches 63KiB and just before the connection
            # is closed.
            sender.flush()

    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 *questdb.ingress.Buffer* objects explicitly, then pass them to the *questdb.ingress.Sender.flush()* method.

Note that this bypasses `auto-flush` logic (see *questdb.ingress.Sender*) and you are fully responsible for ensuring all data is sent.

```
from questdb.ingress import Sender, TimestampNanos


def example(host: str = 'localhost', port: int = 9009):
    with Sender(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'},
```

(continues on next page)

(continued from previous page)

```
            columns={'price': 111222233343, 'qty': 2.5},
            at=TimestampNanos(111222233343))
        sender.flush(buffer)


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

Ticking Random Data and Timer-based Flush

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

It creates random ticking data at a random interval and flushes it explicitly based on a timer if the auto-flushing logic was not triggered recently.

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


def example(host: str = 'localhost', port: int = 9009):
    table_name: str = str(uuid.uuid1())
    watermark = 1024  # Flush if the internal buffer exceeds 1KiB
    with Sender(host=host, port=port, auto_flush=watermark) as sender:
        total_rows = 0
        last_flush = time.monotonic()
        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)})
                total_rows += 1

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

                # Flush at least once every five seconds.
                if time.monotonic() - last_flush > 5:
                    print('Timer-flushing triggered.')
```

(continues on next page)

(continued from previous page)

```
                    sender.flush()
                    last_flush = time.monotonic()

        except KeyboardInterrupt:
            print(f"table: {table_name}, total rows sent: {total_rows}")
            print("(wait commitLag for all rows to be available)")
            print("bye!")


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

7.2.2 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 = 9009):
    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(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 *questdb.ingress.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


def example(host: str = 'localhost', port: int = 9009):
    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(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 *questdb.ingress.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(
            ['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 = 9009):
    filename = write_parquet_file()

    df = pd.read_parquet(filename)
    with Sender(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 *questdb.ingress.Buffer.dataframe()* method.

7.3 API Reference

7.3.1 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(
    '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_capacity` (`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_capacity=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(host='localhost', port=9009,
    init_capacity=16384, max_name_len=64)
buf = sender.new_buffer()
assert buf.init_capacity == 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: Optional[str]*, *table_name_col: Union[None, int, str]*, *symbols: Union[str, bool, List[int], List[str]]*, *at: Union[None, 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.

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 `None`, timestamp is assigned by the server for all rows. To pass in a timestamp explicity 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_capacity

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: unicode*, *, *symbols: Optional[Dict[str, Optional[str]]]*, *columns: Optional[Dict[str, Union[None, bool, int, float, str,* TimestampMicros*, datetime]]]*, *at: Union[None,* 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'})
```

(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).

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. If `None`, timestamp is assigned by the server. 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*)

Bases: Enum

Category of Error.

AuthError = 6

BadDataFrame = 8

CouldNotResolveAddr = 0

InvalidApiCall = 1

InvalidName = 4

InvalidTimestamp = 5

InvalidUtf8 = 3

SocketError = 2

TlsError = 7

class questdb.ingress.**Sender**

Bases: object

A sender is a client that inserts rows into QuestDB via the ILP protocol.

Inserting two rows

In this example, data will be flushed and sent at the end of the `with` block.

```
with Sender('localhost', 9009) as sender:
    sender.row(
        'weather_sensor',
        symbols={'id': 'toronto1'},
        columns={'temperature': 23.5, 'humidity': 0.49})
    sensor.row(
        'weather_sensor',
        symbols={'id': 'dubai2'},
        columns={'temperature': 41.2, 'humidity': 0.34})
```

The Sender object holds an internal buffer. The call to `.row()` simply forwards all arguments to the *Buffer.row()* method.

Explicit flushing

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

```
with Sender('localhost', 9009) as sender:
    sender.row(
        'weather_sensor',
        symbols={'id': 'toronto1'},
        columns={'temperature': 23.5, 'humidity': 0.49})
    sender.flush()
    sender.row(
        'weather_sensor',
        symbols={'id': 'dubai2'},
        columns={'temperature': 41.2, 'humidity': 0.34})
    sender.flush()
```

Auto-flushing (on by default, watermark at 63KiB)

To avoid accumulating very large buffers, the sender will flush the buffer automatically once its buffer reaches a certain byte-size watermark.

You can control this behavior by setting the `auto_flush` argument.

```
# Never flushes automatically.
sender = Sender('localhost', 9009, auto_flush=False)
sender = Sender('localhost', 9009, auto_flush=None) # Ditto.
sender = Sender('localhost', 9009, auto_flush=0)  # Ditto.

# Flushes automatically when the buffer reaches 1KiB.
sender = Sender('localhost', 9009, auto_flush=1024)

# Flushes automatically after every row.
sender = Sender('localhost', 9009, auto_flush=True)
sender = Sender('localhost', 9009, auto_flush=1)  # Ditto.
```

Authentication and TLS Encryption

This implementation supports authentication and TLS full-connection encryption.

The `Sender(.., auth=..)` argument is a tuple of `(kid, d, x, y)` as documented on the QuestDB ILP authentication documentation. Authentication is optional and disabled by default.

The `Sender(.., tls=..)` argument is one of:

- `False`: No TLS encryption (default).
- `True`: TLS encryption, accepting all common certificates as recognized by the webpki-roots Rust crate which in turn relies on https://mkcert.org/.
- A `str` or `pathlib.Path`: Path to a PEM-encoded certificate authority file. This is useful for testing with self-signed certificates.
- A special `'insecure_skip_verify'` string: Dangerously disable all TLS certificate verification (do *NOT* use in production environments).

Positional constructor arguments for the Sender(..)

- `host`: Hostname or IP address of the QuestDB server.
- `port`: Port number of the QuestDB server.

Keyword-only constructor arguments for the Sender(..)

- `interface` (`str`): Network interface to bind to. Set this if you have an accelerated network interface (e.g. Solarflare) and want to use it.
- `auth` (`tuple`): Authentication tuple or `None` (default). *See above for details*.
- `tls` (`bool`, `pathlib.Path` or `str`): TLS configuration or `False` (default). *See above for details*.
- `read_timeout` (`int`): How long to wait for messages from the QuestDB server during the TLS handshake or authentication process. This field is expressed in milliseconds. The default is 15 seconds.
- `init_capacity` (`int`): Initial buffer capacity of the internal buffer. *Default: 65536 (64KiB). See Buffer's constructor for more details.*
- `max_name_length` (`int`): Maximum length of a table or column name. *See Buffer's constructor for more details.*

- `auto_flush` (`bool` or `int`): Whether to automatically flush the buffer when it reaches a certain byte-size watermark. *Default: 64512 (63KiB). See above for details.*

__enter__() → *Sender*

Call *Sender.connect()* 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()*.

__str__()

Inspect the contents of the internal buffer.

The `str` value returned represents the unsent data.

Also see `Sender.__len__()`.

close(*flush*)

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.

connect()

Connect to the QuestDB server.

This method is synchronous and will block until the connection is established.

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

dataframe(*df*, ***, *table_name: Optional[str]*, *table_name_col: Union[None, int, str]*, *symbols: Union[str, bool, List[int], List[str]]*, *at: Union[None, 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'),
        pd.Timestamp('2022-08-09 13:56:02'),
        pd.Timestamp('2022-08-09 13:56:03')]})
```

(continues on next page)

(continued from previous page)

```
with qi.Sender('localhost', 9000) 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.

flush(*buffer*, *clear*)

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.

The Python GIL is released during the network IO operation.

init_capacity

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_capacity* and *max_name_len*.

row(*table_name: unicode*, *, *symbols: Optional[Dict[str, str]]*, *columns: Optional[Dict[str, Union[bool, int, float, str,* TimestampMicros*, datetime]]]*, *at: Union[None,* TimestampNanos*, datetime]*)

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.

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).

7.4 Troubleshooting

7.4.1 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 *questdb.ingress.Sender.flush()* often enough.

The *questdb.ingress.Sender* class only provides auto-flushing based on a buffer size and *not on a timer*.

Errors during flushing

Server disconnects

A failure in *questdb.ingress.Sender.flush()* generally indicates that the network connection was dropped.

The ILP 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.

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(...) 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.

7.4.2 Asking for help

The best way to get help is through Slack.

7.5 Contributing

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

7.5.1 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.

7.5.2 Feature requests and feedback

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

7.5.3 Development

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

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.

7.6 Changelog

7.6.1 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 (confusingly) 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.

7.6.2 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.

7.6.3 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.

Bugfixes

- 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.

7.6.4 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.

7.6.5 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.

7.6.6 0.0.1 (2022-07-08)

Features

- First release on PyPI.

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

Symbols

A

B

C

D

F

I

M

N

Q

R

S

T

V