btrdb-python Documentation

Release v5.32

PingThingsIO

Apr 19, 2024

CONTENTS

Welcome to btrdb-python's documentation. We provide Python access to the Berkeley Tree Database (BTrBD) along with some select convenience methods. If you are familiar with other NoSQL libraries such as pymongo then you will likely feel right at home using this library.

BTrDB is a very, very fast timeseries database. Specifically, it is a time partitioned, version annotated, clustered solution for high density univariate data. It's also incredibly easy to use. Checkout out our *Installing* page to get setup and then visit *Quick Start* for a brief tour. Some sample code is below to whet your appetite.

```
import btrdb
from btrdb.utils.timez import to_nanoseconds

# establish connection to a server
conn = btrdb.connect("192.168.1.101:4410")

# search for streams and view metadata
streams = conn.streams_in_collection("USEAST_NOC1/90807")
for stream in streams:
    print(stream.collection, stream.name, stream.tags())

# retrieve a single stream
stream = conn.stream_from_uuid("07d28a44-4991-492d-b9c5-2d8cec5aa6d4")

# print one hour of time series data starting at 1/1/2018 12:30:00 UTC
start = to_nanoseconds(datetime(2018,1,1,12,30))
end = start + (60 * 60 * 1e9)
for point, _ in stream.values(start, end):
    print(point.time, point.value)

# return the data as an arrow table instead
data = stream.arrow_values(start, end)
```

CHAPTER

ONE

USER GUIDE

The remaining documentation can be found below. If there is anything you'd like added or corrected, please feel free to submit a pull request or open an issue in Github!

1.1 Quick Start

1.1.1 Connecting to a server

Connecting to a server is easy with the supplied `connect` function from the btrdb package.

```
>>> import btrdb
>>> # connect with API key
>>> conn = btrdb.connect("192.168.1.101:4411", apikey="123456789123456789")
>>> conn
<btrdb.conn.BTrDB at 0x...>
```

Get Platform Information

```
>>> conn.info()
{'majorVersion': ...,
 'minorVersion': ...,
 'build': ...,
 'proxy': {...}}
```

Refer to *the connection API documentation page.*

1.1.2 Retrieving a Stream

In order to interact with data, you'll need to obtain or create a `Stream` object. A number of options are available to get existing streams.

Find streams by collection

Multiple streams are often organized under a single collection which is similar to the concept of a directory path. To search for all streams under a given collection you can use the `streams_in_collection` method.

```
>>> streams = conn.streams_in_collection("USEAST_NOC1/90807")
>>> for stream in streams:
>>>     print(stream.uuid, stream.name)
```

Find stream by UUID

A method has also been provided if you already know the UUID of a single stream you would like to retrieve. For convenience, this method accepts instances of either `str` or `UUID`.

```
>>> stream = conn.stream_from_uuid("07d28a44-4991-492d-b9c5-2d8cec5aa6d4")
```

1.1.3 Viewing a Stream's Data

To view data within a stream, you'll need to specify a time range to query for as well as a version number (defaults to latest version). Remember that BTrDB stores data to the nanosecond and so Unix timestamps will need to be converted if needed.

```
>>> start = datetime(2018,1,1,12,30, tzinfo=timezone.utc)
>>> start = start.timestamp() * 1e9
>>> end = start + (3600 * 1e9)

>>> for point, _ in stream.values(start, end):
>>>   print(point.time, point.value)
```

Some convenience functions are available to make it easier to deal with converting to nanoseconds.

```
>>> from btrdb.utils.timez import to_nanoseconds, currently_as_ns

>>> start = to_nanoseconds(datetime(2018,1,1, tzinfo=timezone.utc))
>>> end = currently_as_ns()

>>> for point, _ in stream.values(start, end):
>>>   print(point.time, point.value)
```

You can also view windows of data at arbitrary levels of detail. One such windowing feature is shown below.

```
>>> # query for windows of data 10,000 nanoseconds wide using a depth of zero
>>> # which is accurate to the nanosecond.
>>> params = {
...     "start": 1500000000000000000,
...     "end": 1500000000010000000,
...     "width": 2000000,
...     "depth": 0,
... }
>>> for window in stream.windows(**params):
>>>     for point, version in window:
>>>         print(point, version)
```

Return data as `arrow` tables

Instead of returning data a `RawPoint` at a time, which can be more computationally intensive, there is now the ability to return the data in a tabular format from the start, which can drastically save on run time as well as facilitate interoperability with many more data-science driven tools. Apache Arrow is a language agnostic columnar data schema that has become a defacto standard for in-memory data analytics. All data retrieval methods in `BTrDB` now have corresponding `arrow-` prepended methods that natively return `pyarrow` data tables.

```
>>> s.arrow_values(start=1500000000000000000, end=1500000002000000001).to_pandas()
                                 time  value
    0        2017-07-14 02:40:00+00:00    1.0
    1 2017-07-14 02:40:00.100000+00:00    2.0
    2 2017-07-14 02:40:00.200000+00:00    3.0
    3 2017-07-14 02:40:00.300000+00:00    4.0
    4 2017-07-14 02:40:00.400000+00:00    5.0
    5 2017-07-14 02:40:00.500000+00:00    6.0
    6 2017-07-14 02:40:00.600000+00:00    7.0
    7 2017-07-14 02:40:00.700000+00:00    8.0
    8 2017-07-14 02:40:00.800000+00:00    9.0
    9 2017-07-14 02:40:00.900000+00:00   10.0
```

1.1.4 Using StreamSets

A `StreamSet` is a wrapper around a list of `Stream` objects with a number of convenience methods available. Future updates will allow you to query for streams using a SQL-like syntax but for now you will need to provide a list of UUIDs.

The StreamSet allows you to interact with a group of streams rather than at the level of the individual `Stream` object. Aside from being useful to see concurrent data across streams, you can also easily transform the data to other data structures or even serialize the data to disk in one operation.

Some quick examples are shown below but please review the *API docs* for the full list of features.

Note: In the following examples, notice that the end time is **not** inclusive of the data that is present at `end` . `start` is **inclusive** while `end` is **exclusive**. This is the case for **all** `BTrDB` data query operations.

$$[start, end)$$

```
>>> streams = db.streams(*uuid_list)

>>> # serialize data to disk as CSV
>>> streams.filter(start=1500000000000000000, end=1500000000900000000).to_csv("data.csv")

>>> # convert data to a pandas DataFrame
>>> streams.filter(start=1500000000000000000, end=1500000000900000000).to_dataframe()
                     nw/stream0  nw/stream1
time
1500000000000000000         nan         1.0
1500000000100000000         2.0         nan
1500000000200000000         nan         3.0
1500000000300000000         4.0         nan
```

(continues on next page)

(continued from previous page)

```
1500000000400000000          nan          5.0
1500000000500000000          6.0          nan
1500000000600000000          nan          7.0
1500000000700000000          8.0          nan
1500000000800000000          nan          9.0



>>> # materialize the streams' data
>>> streams.filter(start=1500000000000000000,  end=1500000000900000000).values()
[[RawPoint(1500000000100000000, 2.0),
    RawPoint(1500000000300000000, 4.0),
    RawPoint(1500000000500000000, 6.0),
    RawPoint(1500000000700000000, 8.0),
    RawPoint(1500000000900000000, 10.0)],
   [RawPoint(1500000000000000000, 1.0),
    RawPoint(1500000000200000000, 3.0),
    RawPoint(1500000000400000000, 5.0),
    ...
```

Return data as `arrow` tables

`StreamSets` are also able to return `arrow` tables for the group of streams they represent. This is especially convenient and is usually **much** faster than using the traditional `RawPoint` -based data representation. We recommend using the `arrow` functions whenever possible.

```
>>> # convert data to a pandas DataFrame, using pyarrow
>>> streams.filter(start=1500000000000000000, end=1500000000900000000)
...        .arrow_to_dataframe()
                                 NW/stream0  NW/stream1
time
2017-07-14 02:40:00+00:00               NaN         1.0
2017-07-14 02:40:00.100000+00:00        2.0         NaN
2017-07-14 02:40:00.200000+00:00        NaN         3.0
2017-07-14 02:40:00.300000+00:00        4.0         NaN
2017-07-14 02:40:00.400000+00:00        NaN         5.0
2017-07-14 02:40:00.500000+00:00        6.0         NaN
2017-07-14 02:40:00.600000+00:00        NaN         7.0
2017-07-14 02:40:00.700000+00:00        8.0         NaN
2017-07-14 02:40:00.800000+00:00        NaN         9.0


>>> # materialize the streams' data as an arrow table
>>> streams.filter(start=1500000000000000000, end=1500000000900000000).arrow_values()
    pyarrow.Table
    time: timestamp[ns, tz=UTC] not null
    b29204f4-6c13-4ec7-a149-88e2ff950a72: double not null
    99a0d0b0-e24f-4875-b7d8-eae0036f2149: double not null
    ----
    time: [
```

(continues on next page)

(continued from previous page)

```
    ... [2017-07-14 02:40:00.000000000Z,2017-07-14 02:40:00.100000000Z,
    ... 2017-07-14 02:40:00.200000000Z,2017-07-14 02:40:00.300000000Z,
    ... 2017-07-14 02:40:00.400000000Z,2017-07-14 02:40:00.500000000Z,
    ... 2017-07-14 02:40:00.600000000Z,2017-07-14 02:40:00.700000000Z,
    ... 2017-07-14 02:40:00.800000000Z]]
    b29204f4-6c13-4ec7-a149-88e2ff950a72: [[nan,2,nan,4,nan,6,nan,8,nan]]
    99a0d0b0-e24f-4875-b7d8-eae0036f2149: [[1,nan,3,nan,5,nan,7,nan,9]]
```

1.2 Installing

The `btrdb` package has only a few requirements and is relatively easy to install. A number of installation options are available as detailed below.

1.2.1 Installing with pip

We recommend using `pip` to install `btrdb-python` on all platforms:

```
$ pip install btrdb
```

With `btrdb>=5.30.2`, there are now extra dependencies that can be installed with `pip`. We recommend installing the `data` extra dependencies (the second option in the code block below).

```
$ pip install "btrdb>=5.30.2" # standard btrdb
$ pip install "btrdb[data]>=5.30.2" # btrdb with data science packages included␣
↪(recommended)
$ pip install "btrdb[all]>=5.30.2" # btrdb with testing, data science and all other␣
↪optional packages
```

To get a specific version of `btrdb-python` supply the version number. The major version of this library is tied to the major version of the BTrDB database as in the 4.X bindings are best used to speak to a 4.X BTrDB database, the 5.X bindings for 5.X platform..

```
$ pip install "btrdb[data]==5.30.2"
```

To upgrade using pip:

```
$ pip install --upgrade btrdb
```

1.2.2 Installing with Anaconda

We recommend installing using `pip`.

1.3 Concepts

If you are relatively new to BTrDB, then there are a few things you should be aware of about interacting with the server. First of all, time series databases such as BTrDB are not relational databases and so they behave differently, have different access methods, and provide different guarantees.

The following sections provide insight into the high level objects and aspects of their behavior which will allow you to use them effectively.

Note: Data requests are fully materialized at this time. A future release will include the option to process data using generators to save on memory usage.

1.3.1 BTrDB Server

Like most time series databases, the BTrDB server contains multiple streams of data in which each stream contains a data point at a given time. However, BTrDB focuses on univariate data which opens a host of benefits and is one of the reasons BTrDB is able to process incredibly large amounts of data quickly and easily.

1.3.2 Points

Points of data within a time series make up the smallest objects you will be dealing with when making calls to the database. Because there are different types of interactions with the database, there are different types of points that could be returned to you: `RawPoint` and `StatPoint`.

RawPoint

The RawPoint represents a single time/value pair and is the simpler of the two types of points. This is most useful when you need to process every single value within the stream.

```
>>> # view time and value of a single point in the stream
>>> point.time
1547241923338098176

>>> point.value
120.5
```

StatPoint

The StatPoint provides statistics about multiple points and gives aggregation values such as `min`, `max`, `mean`, `count` and `stddev` (standard deviation). This is most useful when you don't need to touch every individual value such as when you only need the count of the values over a range of time.

These statistical queries execute in time proportional to the number of results, not the number of underlying points (i.e logarithmic time) and so you can attain valuable data in a fraction of the time when compared with retrieving all of the individual values. Due to the internal data structures, `BTrDB` does not need to read the underlying points to return these statistics!

```
>>> # view aggregate values for points in a stream
>>> point.time
1547241923338098176

>>>point.min
42.1

>>> point.mean
78.477

>>> point.max
122.4

>>> point.count
18600

>>> point.stddev
3.4
```

1.3.3 Tabular Data

In addition to working with the `RawPoint` or `StatPoint` objects, newer versions of the platform now natively support some tabular data formats as well. This is enabled for commercial customers and are available using the `stream.arrow_` or `streamset.arrow_` methods. Refer to the *arrow enabled queries page* and the *API docs*

1.3.4 Streams

`Stream` s represent a single series of time/value pairs. As such, the database can hold an almost unlimited amount of individual streams. Each stream has a `collection` which is similar to a "path" or grouping for multiple streams. Each steam will also have a `name` as well as a `uuid` which is guaranteed to be unique across streams.

`BTrDB` data is versioned such that changes to a given stream (time series) will result in a new version for the stream. In this manner, you can pin your interactions to a specific version ensuring the values do not change over the course of your interactions.

Note: If you want to work with the most recent version/data then specify a version of `0` (the default).

Each stream has a number of attributes and methods available and these are documented within the *API Reference* section of this publication. But the most common interactions by users are to access the `UUID`, `tags`, `annotations`, `version`, and underlying data.

Each stream uses a `UUID` as its unique identifier which can also be used when querying for streams. Metadata is provided by `tags` and `annotations` which are both provided as dictionaries of data. `tags` are used internally and have very specific keys while `annotations` are more free-form and can be used by you to store your own metadata.

```
>>> # retrieve stream's UUID
>>> stream.uuid
UUID("0d22a53b-e2ef-4e0a-ab89-b2d48fb2592a")

>>> # retrieve stream's current version
>>> stream.version()
```

(continues on next page)

(continued from previous page)

```
244

>>> # retrieve stream tags
>>> stream.tags()
{'name': 'L1MAG', 'unit': 'volts', 'ingress': ''}

>>> # retrieve stream annotations
>>> stream.annotations()
({'poc': 'Salvatore McFesterson', 'region': 'northwest', 'state': 'WA'}, 23)

>>> # loop through points in the stream
>>> for point, _ in stream.values(end=1547241923338098176, version=133):
>>>     print(point)
RawPoint(1500000000100000000, 2.4)
RawPoint(1500000000200000000, 2.8)
RawPoint(1500000000300000000, 3.6)
...
```

1.3.5 StreamSets

Often you will want to query and work with multiple streams instead of just an individual stream - `StreamSets` allow you to do this effectively. It is a light wrapper around a list of `Stream` objects with convenience methods provided to help you work with multiple streams of data.

As an example, you can filter the stream data with a single method call and then easily transform the data into other data types such as a pandas DataFrame or to disk as a CSV file. See the examples below for a quick sample and then visit our API docs to see the full list of features provided to you.

Note: `StreamSet` methods that filter and operate on the `StreamSet` object (like `StreamSet.filter`) return new copies of the `StreamSet` itself rather than modifying in place. Similar to how most `pandas.DataFrame` methods return a new `DataFrame` object. This lets you compose multiple functions in a single call, which can improve readability, but can be tricky if you are not expecting this behavior.

Lets explore a common use-case, filtering a streamset.

```
>>> # create a streamset and apply a few filters
>>> streamset = btrdb.stream.StreamSet(list_of_streams)
>>> print(f"Total streams: {len(streamset)}")
Total streams: 89

>>> streamset.filter(units="Volts")
>>> print(f"Total streams: {len(streamset)}")
Total streams: 89

>>> filtered_streamset = streamset.filter(units="Volts")
>>> print(f"Total streams: {len(filtered_streamset)}")
Total streams: 23

>>> multiple_filters_streamset = (streamset.filter(unit="Volts")
>>>                               .filter(name="Sensor 1")
```

(continues on next page)

(continued from previous page)

```
>>>                                 .filter(annotations={"phase":"A"})
>>>                             )
>>> print(f"Total streams: {len(multiple_filters_streamset)}")
Total streams: 1
```

```
>>> # establish database connection and query for streams by UUID
>>> db = connect()
>>> uuid_list = ["0d22a53b-e2ef-4e0a-ab89-b2d48fb2592a", ...]
>>> streams = db.streams(*uuid_list)

>>> streams.filter(start=1500000000000000000).to_csv("data.csv")

>>> streams.filter(start=1500000000000000000).to_dataframe()
              time              NW/stream0  NW/stream1
    0  1500000000000000000         NaN         1.0
    1  1500000000100000000         2.0         NaN
    2  1500000000200000000         NaN         3.0
    3  1500000000300000000         4.0         NaN
    4  1500000000400000000         NaN         5.0
    5  1500000000500000000         6.0         NaN
    6  1500000000600000000         NaN         7.0
    7  1500000000700000000         8.0         NaN
    8  1500000000800000000         NaN         9.0
    9  1500000000900000000        10.0         NaN
```

1.3.6 Apache-Arrow Accelerated Methods

- Refer to *Arrow-enabled Queries*

1.4 Working with btrdb

Please review the guided tour linked below to get a better understanding of how to interact with the BTrDB database.

1.4.1 Server Connection and Info

There are a number of options available when connecting to a BTrDB server or server cluster. First, you will need to identify the appropriate IP or FQDN to use as well as the access port.

By default BTrDB servers expose port 4410 for unencrypted access and 4411 for encrypted access using TLS. You may also opt for authentication using an API key which can be provided to you by the BTrDB server administrators. Using such a key will require the TLS port (4411) as attempting to use a different port with an API key will raise an exception.

Connecting to servers

The btrdb library comes with a high level `connect` function to interface with a BTrDB server. Upon successfully connecting, you will be returned a `BTrDB` object which is the starting point for all of your server interactions.

For your convenience, you may default all connection parameters to environment variables if these are configured on your system. If no arguments are provided, the `btrdb.connect` function will attempt to connect using the `BTRDB_ENDPOINTS` and `BTRDB_API_KEY` environment variables.

Several connection options are shown in the code below:

```
import btrdb

# connect using BTRDB_ENDPOINTS and BTRDB_API_KEY ENV variables
conn = btrdb.connect()

# connect without credentials
conn = btrdb.connect("192.168.1.101:4410")

# connect without credentials using TLS
conn = btrdb.connect("192.168.1.101:4411")

# connect with API key
conn = btrdb.connect("192.168.1.101:4411", apikey="123456789123456789")
```

Using Profiles

In addition to providing the endpoint and API key directly (or through environment variables), you may provide a profile name which looks into your PredictiveGrid credentials file at *$HOME/.predictivegrid/credentials.yaml*. Using profiles is meant as a (optional) convenience device and may also be supplied through the environmental variable *$BTRDB_PROFILE*.

```
import btrdb

# connect using your own "research" profile
conn = btrdb.connect(profile="research")
```

The credentials file is in YAML format as shown below.

```
research:
  name: "research"
  btrdb:
    endpoints: "research.example.com:4411"
    api_key: "d976a2d61103feb2235441fd6887955c"
default:
  name: "default"
  btrdb:
    endpoints: "btrdb.example.com:4411"
    api_key: "e666a2d61103feb2235441fd68879440"
```

Connection Info Resolution

The `connect` function is quite aggressive about finding ways to connect to the server and power users could get into odd edge cases if using multiple profiles with incomplete entries. For troubleshooting purposes, the `connect` function performs the following steps to determine the correct server credentials.

1. Load profile connection info with the `BTRDB_PROFILE` environment variable or load the default profile if not found.
2. Overwrite the profile data with `BTRDB_ENDPOINTS` and `BTRDB_API_KEY` environment variables if available.
3. Overwrite accumulated connection data with `endpoints` and `api_key` arguments if supplied.

Viewing server status

Server version and connection information can be viewed by calling the `info` method of the server object as shown below.

```
conn = btrdb.connect()
conn.info()
>> {'majorVersion': 5, 'build': '5.0.0', 'proxy': {'proxyEndpoints': '192.168.1.101:4410
↪'}}
```

1.4.2 Querying and Managing Streams

With BTrDB, you can easily create, delete, and query for streams using simple method calls. Simple examples are included below but please review the API docs for further options.

Create a Stream

Creating a stream requires only a UUID, collection, and dictionary for the initial tags.

```
conn = btrdb.connect()

stream = conn.create(
    uuid=uuid.uuid4(),
    collection="NORTHWEST/90001",
    tags={"name": "L1MAG", "unit": "volts"}
)
```

Delete a Stream

Deleting a stream can be performed by calling the *obliterate* method on the stream object. If the stream could not be found than an error is raised.

```
conn = btrdb.connect()
stream = conn.stream_from_uuid("66466a91-dcfe-42ea-9e88-87c51f847944")
stream.obliterate()
```

Find by UUID

To retrieve your stream from the server at a later date, you can easily query for it by using the UUID it was created with. As a convenience, you can provide either a UUID object or a string of the UUID value. If a stream matching the supplied UUID cannot be found then `None` will be returned.

```
conn = btrdb.connect()
stream = conn.stream_from_uuid("71466a91-dcfe-42ea-9e88-87c51f847942")
```

Look up collections

You can look up collections found in the server by using the `list_collections` method, which returns a list of string collection names. Additionally, you can use the `starts_with` parameter to filter the results to include only collections that begin with the provided prefix. Omitting the `starts_with` parameter will return all available collections from the server.

```
conn = btrdb.connect()
collections = conn.list_collections(starts_with="NORTHWEST")
```

Finding by collection

You can also search for multiple streams by collection using the server object's `streams_in_collection` method which will return a simple list of `Stream` instances. Aside from the collection name, you can provide more information such as tags and annotations. Please see the API docs for more detail.

```
conn = btrdb.connect()
streams = conn.streams_in_collection("NORTHEAST/NH")
```

Querying Metadata

Finally, you can query for metadata using standard SQL although at the moment, only the `streams` table is available. SQL queries can be submitted using the `query` method which accepts both a `stmt` and `params` argument. The `stmt` should contain the SQL you'd like executed with parameter placeholders such as `$1` or `$2` as shown below.

```
conn = btrdb.connect()
stmt = "SELECT uuid FROM streams WHERE name = $1 OR name = $2"
params = ["Boston_1", "Boston_2"]

for row in conn.query(stmt, params):
  print(row)
```

The SQL query results are returned as a list of dictionaries where each key matches a column from your SQL projection. You can choose your columns from the schema of the streams table as follows.

Column	Type	Nullable
uuid	uuid	not null
collection	character varying(256)	not null
name	character varying(256)	not null
unit	character varying(256)	not null
ingress	character varying(256)	not null
property_version	bigint	not null
annotations	hstore	

1.4.3 Managing Stream Data

BTrDB allows you to insert data and delete data using Stream objects.

Inserting Data

You can insert data into a Stream at any time - even for times that already exist! As we will later see, querying data will return `RawPoint` and `StatPoint` objects however inserting data requires only a time `int` and value `float` within in a `tuple` object (`tuple(int, float)`).

After inserting your data, the server will return a new version number for your stream.

```
payload = [
    (1500000000000000000, 1.0), (1500000000000100000, 2.1),
    (1500000000000200000, 3.3), (1500000000000300000, 5.1),
    (1500000000000400000, 5.7), (1500000000000500000, 6.1),
]
version = stream.insert(payload)
```

Deleting Data

To delete data from a stream you must provide a range (start/end) of time to the `delete` method.

Because you are modifying data, the version number is incremented and will be returned from the server at the end of your call. Keep in mind that data is never truly gone as you can query for the deleted data using an older version of the Stream.

```
version = stream.delete(start=1500000000000000000, end=1520000000000000000)
```

1.4.4 Managing Stream Metadata

BTrDB has multiple options for storing stream metadata including collection, tags, annotations, and others. Most metadata is returned as a `string`, or specialized object such as the UUID. Tags and annotations are returned as `dict` objects.

There is also the concept of the "property version" which is a version counter that applies only to the metadata and is separate from the version incremented with changes to the data. See the API docs for `Stream.annotations` or `Stream.update` for more information.

Viewing Metadata

Viewing the metadata for a Stream is as simple as calling the appropriate property or method. In cases where the data is not expected to change quickly, a Stream instance will provide you with cached values unless you force it to refresh with the server.

UUID

The `uuid` property of a `Stream` is read-only and will return an instance of class `UUID`.

```
stream.uuid
>> UUID('07d28a44-4991-492d-b9c5-2d8cec5aa6d4')
```

Tags

Tags are special key/value metadata that is most often used by the database for internal purposes. As an example, the name of a `Stream` is actually stored in the tags. While you can update tags, it is not recommended that you add new tags or delete existing tags. Tag values have a 255 character limit.

```
stream.tags(refresh=True)
>> {'name': 'L1MAG', 'unit': 'volts', 'ingress': ''}
```

Annotations

Similar to tags, annotations are key/value pairs that are available for your use to store extra information about the `Stream`.

Because annotations may change more often than tags, a metadata version number is also returned when asking for annotations. This version number is incremented whenever metadata (tags, annotations, collection, etc.) are updated but not when making changes to the underlying time series data.

By default the method will attempt to provide a cached copy of the annotations however you can request the latest version from the server using the *refresh* argument. As with tags, annotations values also have a 255 character limit.

```
stream.annotations(refresh=True)
>> ({'owner': 'Salvatore McFesterson', 'state': 'NH'}, 44)
```

Name and Collection

The `name` and `collection` properties of a Stream are read-only and will return instances of `str`. Note that the `name` property is just a convenience as this value can also be found within the tags.

```
stream.collection
>> 'NORTHEAST/VERMONT/Burlington'

stream.name
>> 'L1MAG'
```

Updating Metadata

An `update` method is available if you would like to make changes to the tags, annotations, or collection. By default, all updates are implemented as an UPSERT operation and a single change could result in multiple increments to the property version (the version of the metadata).

Upon calling this method, the library will first verify that the local property version of your stream object matches the version found on the server. If the two versions do not match then you will not be allowed to perform an update as this implies that the data has already been changed by another user or process.

```
collection = 'NORTHEAST/VERMONT'
annotations = {
    'owner': 'Salvatore McFesterson',
    'state': 'VT',
    'created': '2018-01-01 12:42:03 -0500'
}
property_version = stream.update(
    collection=collection,
    annotations=annotations
)
```

If you would like to remove any keys from your annotations you must use the `replace=True` keyword argument. This will ensure that the annotations dictionary you provide completely replaces the existing values rather than perform an UPSERT operation. The example below shows how you could remove an existing key from the annotations dictionary.

```
annotations, _ = stream.anotations()
del annotations["key_to_delete"]
stream.update(annotations=annotations, replace=True)
```

1.4.5 Viewing Stream Data

At a high level, there are two options available when you are ready to retrieve the time series data in a stream. You may view the values directly by timestamp or you can view a window of data at a resolution of your choice. When viewing by window, there are further options available with different arguments and related performance benefits.

View Individual Data Points

To view the values directly, call the `Stream.values` method which will fully materialize the stream values at the stream version you specify (use the default value of zero as the latest version). A `start` and `end` argument is required when making this request.

Calling `Stream.values` will return a series of `tuple`, with each item containing a `RawPoint`, and version of the stream (`int`). As described in the API reference, a `RawPoint` has both a `time` and `value` property.

```
start = 1500000000000000000
end = 1547241923338098176

for point, _ in stream.values(start=start, end=end, version=133):
    print(point)
>> RawPoint(1500000000000000000, 2.35)
>> RawPoint(1500000000100000000, 2.41)
>> RawPoint(1500000000200000000, 2.8)
```

(continues on next page)

(continued from previous page)

```
>> RawPoint(1500000000300000000, 3.66)
...
```

Helpers for Dates/Times

If you are interested in finding the closest point to a particular datetime, there is the `Stream.nearest` method. Alternatively, if you want to know the first or last points in a stream, you can call the `Stream.earliest` and `Stream.latest` methods. These two are often useful if you would like to view all of the data within the stream using the `Stream.windows` method below (it is not recommended that you query for all the data using the `Stream.values` method due to the memory consumption implied). Each of these three methods returns a tuple containing a RawPoint and the data version number. The exact timestamp can be obtained from the RawPoint. Keep in mind that all of these methods accept a `version` argument so that you can ask for the `earliest`, `latest`, or `nearest` point from a previous version of the stream.

```
stream = db.stream_from_uuid("6f8ebaf0-78ea-416e-a0ff-5c3c5d83c279")
stream.earliest()
>> (RawPoint(1364860800000000000, 42516.03), 3934)
stream.earliest()[0].time
>> 1364860800000000000
```

View Windows of Data

If you don't need to view every single point of data, then it is faster to view higher order representations of the data. BTrDB stores data in a tree structure such that the leaves of the tree contain actual values and higher nodes store statistical data (min, max, mean, etc.) summaries. In this schema viewing summaries of data involves reading from higher levels of the tree and therefore less nodes need to be read from disk.

This use case of wanting a high level summary of data is quite common. For example, when rendering the plot of a time series it will often be useful to present a view at the resolution of one hour, one day, or perhaps one year. With samples that occur at greater than 1Hz this requires you to summarize the values and plot the average (or min, max, etc.) values rather than each individual value.

Because BTrDB is usually providing summaries of data when windowing, it returns instances of `StatPoint` rather than `RawPoint`. A `StatPoint` contains statistical information about a range of time and specifically provides properties for `min`, `mean`, `max`, `count`, `stddev`, and the start `time` for which the statistical summaries cover.

For statistical aggregates of your data, the `Stream.aligned_windows` method is the fastest way to query your data. Each point returned is a statistical aggregate of all the raw data within a window of width 2^pointwidth nanoseconds.

Note that `start` is inclusive, but `end` is exclusive. That is, results will be returned for all windows that start in the interval $[start, end)$. If end < start+2^pointwidth you will not get any results. If start and end are not powers of two, the bottom pointwidth bits will be cleared. Each window will contain statistical summaries of the window. Statistical points with count == 0 will be omitted.

```
start = 1500000000000000000
end = 1500000001000000000

# view underlying data for comparison
for point, _ in stream.values(start=start, end=end):
    print(point)
>> RawPoint(1500000000000000000, 1.0)
>> RawPoint(1500000000100000000, 2.0)
```

(continues on next page)

(continued from previous page)

```
>> RawPoint(1500000000200000000, 3.0)
>> RawPoint(1500000000300000000, 4.0)
>> RawPoint(1500000000400000000, 5.0)
>> RawPoint(1500000000500000000, 6.0)
>> RawPoint(1500000000600000000, 7.0)
>> RawPoint(1500000000700000000, 8.0)
>> RawPoint(1500000000800000000, 9.0)
>> RawPoint(1500000000900000000, 10.0)

# aggregate over 2^28 nanoseconds (268,435,456)
pointwidth = 28

# view data aggregates
for point, _ in stream.aligned_windows(start=start, end=end,
                                       pointwidth=pointwidth):
    print(point)
>> StatPoint(1499999999814008832, 1.0, 1.0, 1.0, 1, 0.0)
>> StatPoint(1500000000082444288, 2.0, 3.0, 4.0, 3, 0.816496580927726)
>> StatPoint(1500000000350879744, 5.0, 6.0, 7.0, 3, 0.816496580927726)
>> StatPoint(1500000000619315200, 8.0, 8.5, 9.0, 2, 0.5)
```

The `Stream.windows` method of a Stream allows you to request windows of data while specifying the precision of the data you require. Each window will cover `width` nanoseconds in length. Precision of the result is determined by the `depth` parameter such that each window will be accurate to 2^depth nanoseconds.

Using a larger depth value will result in faster query execution from the database. For instance, if you are viewing a 24 hours of data you may only require a precision of +/- 1 second and so a depth of 30 may be appropriate. A chart of sample depths are provided below.

Depth	Calculation	Precision in Nanoseconds	Time
0	2^0	1	1 nanosecond
10	2^10	1024	~1 microsecond
20	2^20	1048576	~1 millesecond
30	2^30	1073741824	~1 second

As usual when querying data from BTrDB, the `start` time is inclusive while the `end` time is exclusive. **Note**: that if your last window spans across the end time then it will not be included in the results.

```
>>> start = 1500000000000000000
>>> end = 1500000001000000000

>>> # view underlying data for comparison
>>> for point, _ in stream.values(start=start, end=end):
>>>     print(point)
RawPoint(1500000000000000000, 1.0)
RawPoint(1500000000100000000, 2.0)
RawPoint(1500000000200000000, 3.0)
RawPoint(1500000000300000000, 4.0)
RawPoint(1500000000400000000, 5.0)
RawPoint(1500000000500000000, 6.0)
RawPoint(1500000000600000000, 7.0)
```

(continues on next page)

(continued from previous page)

```
RawPoint(1500000000700000000, 8.0)
RawPoint(1500000000800000000, 9.0)
RawPoint(1500000000900000000, 10.0)

>>> # each window spans 300 milliseconds
>>> width = 300000000

>>> # request a precision of roughly 1 millisecond
>>> depth = 20

>>> # view windowed data
>>> for point, _ in stream.windows(start=start, end=end,
...                                width=width, depth=depth):
 StatPoint(1500000000000000000, 1.0, 2.0, 3.0, 3, 0.816496580927726)
 StatPoint(1500000000300000000, 4.0, 5.0, 6.0, 3, 0.816496580927726)
 StatPoint(1500000000600000000, 7.0, 8.0, 9.0, 3, 0.816496580927726)
```

1.4.6 Working with StreamSets

Often you will want to query and work with multiple streams instead of just an individual stream - StreamSets allow you to do this effectively. It is a light wrapper around a list of Stream objects with convenience methods provided to help you work with multiple streams of data.

Creating a StreamSet

Creating a `StreamSet` is relatively simple assuming you have a `UUID` for each stream that should be a member. In the future, other options may exist such as providing collection or tag matching parameters.

```
UUIDs = [
    uuid.UUID('0d22a53b-e2ef-4e0a-ab89-b2d48fb2592a'),
    uuid.UUID('17dbe387-89ea-42b6-864b-f505cdb483f5'),
    uuid.UUID('71466a91-dcfe-42ea-9e88-87c51f847942'),
    uuid.UUID('570aa71d-fb4f-456f-8533-2b11a28fa1f5')
]

streams = conn.streams(*UUIDs)
```

If you've already obtained a list of `Stream` objects, you may create a StreamSet directly by providing a list of streams for initialization.

```
streams = StreamSet([stream1, stream2, stream3])
```

Filtering

To apply query parameters to your request, you should use the `filter` method to supply a `start` or `end` argument.

Keep in mind that `filter` will **return a new object so you can keep multiple filtered StreamSets in memory while you explore your data**. The `filter` method may be called multiple times but only the final values will be used when it is time to fulfill the request by the server.

```
from btrdb.utils.timez import currently_as_ns, to_nanoseconds

streams = conn.streams(*UUIDs)

start = to_nanoseconds(datetime(2016, 1, 1, 0, 0, 0))
end = to_nanoseconds(datetime(2016, 1, 3, 12, 0, 0))

# replace instance with a filtered version from 1/1/2016 00:00:00 to
# 1/3/2016 12:00:00
streams = streams.filter(start=start, end=end)

# create a new instance with epoch as start and the current time as
# the end parameters
alt = streams.filter(start=0, end=currently_as_ns())
```

Aside from filtering results at query execution, you may also filter the streams that should be included in the new object. For instance, you may wish to create a new StreamSet containing only voltage streams or only from a specific collection.

To filter the available streams, you may provide a `collection`, `name`, or `unit` argument. If you provide a string, then a case-insensitive exact match will be used to select the desired streams. You may instead provide a compiled regex expression which will be used with re.search to choose the streams to include.

```
# select only voltage streams
voltage_streams = streams.filter(unit="volts")

# select only voltage or amperage streams using regex pattern
other_streams = streams.filter(unit=re.compile("volts|amps"))
```

Retrieving Data

There are three options available when you are ready to process the data from the server. All options are fully materialized but are organized in different ways according to what is more convenient for you.

StreamSet.values()

Calling the `values` method will materialize the streams using the filtering parameters you specified. The data will be returned to you as a list of lists. Each member list contains tuples of `RawPoint`, `int` for the data and stream version.

This method aligns data by stream so you can easily deal with all of the data on a stream by stream basis. The following example shows a toy dataset which consists of 4 streams.

```
streams.values()
>>[[RawPoint(1500000000100000000, 2.0),
>>  RawPoint(1500000000300000000, 4.0),
>>  RawPoint(1500000000500000000, 6.0),
```

(continues on next page)

(continued from previous page)

```
>>  RawPoint(1500000000700000000, 8.0),
>>  RawPoint(1500000000900000000, 10.0)],
>> [RawPoint(1500000000000000000, 1.0),
>>  RawPoint(1500000000200000000, 3.0),
>>  RawPoint(1500000000400000000, 5.0),
>>  RawPoint(1500000000600000000, 7.0),
>>  RawPoint(1500000000800000000, 9.0)],
>> [RawPoint(1500000000000000000, 1.0),
>>  RawPoint(1500000000100000000, 2.0),
>>  RawPoint(1500000000300000000, 4.0),
>>  RawPoint(1500000000400000000, 5.0),
>>  RawPoint(1500000000600000000, 7.0),
>>  RawPoint(1500000000700000000, 8.0),
>>  RawPoint(1500000000800000000, 9.0),
>>  RawPoint(1500000000900000000, 10.0)],
>> [RawPoint(1500000000000000000, 1.0),
>>  RawPoint(1500000000100000000, 2.0),
>>  RawPoint(1500000000200000000, 3.0),
>>  RawPoint(1500000000300000000, 4.0),
>>  RawPoint(1500000000400000000, 5.0),
>>  RawPoint(1500000000500000000, 6.0),
>>  RawPoint(1500000000600000000, 7.0),
>>  RawPoint(1500000000700000000, 8.0),
>>  RawPoint(1500000000800000000, 9.0),
>>  RawPoint(1500000000900000000, 10.0)]]
```

StreamSet.rows()

By contrast, the `rows` method aligns data by time rather than by stream. Each row of data contains points for a specific time with the `None` value used if a given stream does not contain a value at that time.

Stream data is ordered according to the order of the initial UUIDs that were used when creating the StreamSet.

```
for row in streams.rows():
    print(row)
>> (None, RawPoint(1500000000000000000, 1.0), RawPoint(1500000000000000000, 1.0),␣
↪RawPoint(1500000000000000000, 1.0))
>> (RawPoint(1500000000100000000, 2.0), None, RawPoint(1500000000100000000, 2.0),␣
↪RawPoint(1500000000100000000, 2.0))
>> (None, RawPoint(1500000000200000000, 3.0), None, RawPoint(1500000000200000000, 3.0))
>> (RawPoint(1500000000300000000, 4.0), None, RawPoint(1500000000300000000, 4.0),␣
↪RawPoint(1500000000300000000, 4.0))
>> (None, RawPoint(1500000000400000000, 5.0), RawPoint(1500000000400000000, 5.0),␣
↪RawPoint(1500000000400000000, 5.0))
>> (RawPoint(1500000000500000000, 6.0), None, None, RawPoint(1500000000500000000, 6.0))
>> (None, RawPoint(1500000000600000000, 7.0), RawPoint(1500000000600000000, 7.0),␣
↪RawPoint(1500000000600000000, 7.0))
>> (RawPoint(1500000000700000000, 8.0), None, RawPoint(1500000000700000000, 8.0),␣
↪RawPoint(1500000000700000000, 8.0))
>> (None, RawPoint(1500000000800000000, 9.0), RawPoint(1500000000800000000, 9.0),␣
↪RawPoint(1500000000800000000, 9.0))
```

(continues on next page)

(continued from previous page)

```
>> (RawPoint(1500000000900000000, 10.0), None, RawPoint(1500000000900000000, 10.0),␣
↪RawPoint(1500000000900000000, 10.0))
```

Transforming to Other Formats

A number of transformation features have been added so that you can work in the tools and APIs you are most comfortable and productive with. At the moment, we support the *numpy* and *pandas* libraries if you have them installed and available to be imported.

Keep in mind that calling these methods will materialize the requested data in memory. A few examples follow but please visit the API documentation to see the full list of transformation methods available.

```
# materialize data as tuple of numpy arrays
conn.streams(*UUIDs).filter(start, end).to_array()
>> (array([RawPoint(1500000000100000000, 2.0),
>>         RawPoint(1500000000300000000, 4.0),
>>         RawPoint(1500000000500000000, 6.0),
>>         RawPoint(1500000000700000000, 8.0),
>>         RawPoint(1500000000900000000, 10.0)], dtype=object),
>>  array([RawPoint(1500000000000000000, 1.0),
>>         RawPoint(1500000000200000000, 3.0),
>>         RawPoint(1500000000400000000, 5.0),
>>         RawPoint(1500000000600000000, 7.0),
>>         RawPoint(1500000000800000000, 9.0)], dtype=object),
>> ...

# materialize data as list of pandas Series
conn.streams(*UUIDs).filter(start, end).to_series()
>> [1500000000100000000     2.0
>> 1500000000300000000     4.0
>> 1500000000500000000     6.0
>> 1500000000700000000     8.0
>> 1500000000900000000    10.0
>> dtype: float64,
>>  1500000000000000000    1.0
>> 1500000000200000000    3.0
>> 1500000000400000000    5.0
>> 1500000000600000000    7.0
>> 1500000000800000000    9.0
>> dtype: float64,
>> ...

# materialize data as pandas DataFrame
conn.streams(*UUIDs).filter(start, end).to_dataframe()
>>                    time         sensors/stream0         sensors/stream1
>> 0  1500000000000000000                     NaN                     1.0
>> 1  1500000000100000000                     2.0                     NaN
>> 2  1500000000200000000                     NaN                     3.0
>> 3  1500000000300000000                     4.0                     NaN
>> 4  1500000000400000000                     NaN                     5.0
>> 5  1500000000500000000                     6.0                     NaN
```

(continues on next page)

(continued from previous page)

```
>> 6  1500000000600000000                    NaN                    7.0
>> 7  1500000000700000000                    8.0                    NaN
>> 8  1500000000800000000                    NaN                    9.0
>> 9  1500000000900000000                   10.0                    NaN
```

Serializing Data

If you would like to save your data to disk for later use or to import into another program, we have several options available with more planned in the future.

Most serialization methods will save to disk however there is also a `to_table` method which produces a tabular view of your data as a string for display or printing. Some examples are shown below.

```
# export data and save as CSV
streams.to_csv("export.csv")

# convert table of data as a string
print(streams.to_table())
>>                 time    sensors/stream0    sensors/stream1
>> -------------------  -----------------  -----------------
>> 1500000000000000000                                     1
>> 1500000000100000000                   2
>> 1500000000200000000                                     3
>> 1500000000300000000                   4
>> 1500000000400000000                                     5
>> 1500000000500000000                   6
>> 1500000000600000000                                     7
>> 1500000000700000000                   8
>> 1500000000800000000                                     9
>> 1500000000900000000                  10
```

1.4.7 Multiprocessing

Complex analytics in Python may require additional speedups that can be gained by using the Python multiprocessing library. Other libraries like web applications take advantage of multiprocessing to serve a large number of users. Because btrdb-python uses grpc under the hood, it is important to understand how to connect and reuse connections to the database in a multiprocess or multithread context.

The most critical thing to note is that `btrdb.Connection` objects *are not thread or multiprocess-safe*. This means that in your code you should use either a lock or a semaphore to share a single connection object or that each process or thread should create their own connection object and clean up after themselves when they are done using the connection.

Let's take the following simple example: we want to perform a data quality analysis on 12 hour chunks of data for all the streams in our `staging/sensors` collection. If we have hundreds of sensor streams across many months, this job can be sped up dramatically by using multiprocessing. Instead of having a single process churning through the each chunk of data one at a time, several workers can process multiple data chunks simultanously using multiple CPU cores and taking advantage of other CPU scheduling optimizations.

Consider the processing architecture shown in Fig. 1.1. At first glance, this architecture looks similar to the one used by `multiprocessing.Pool`, which is true. However, consider the following code:

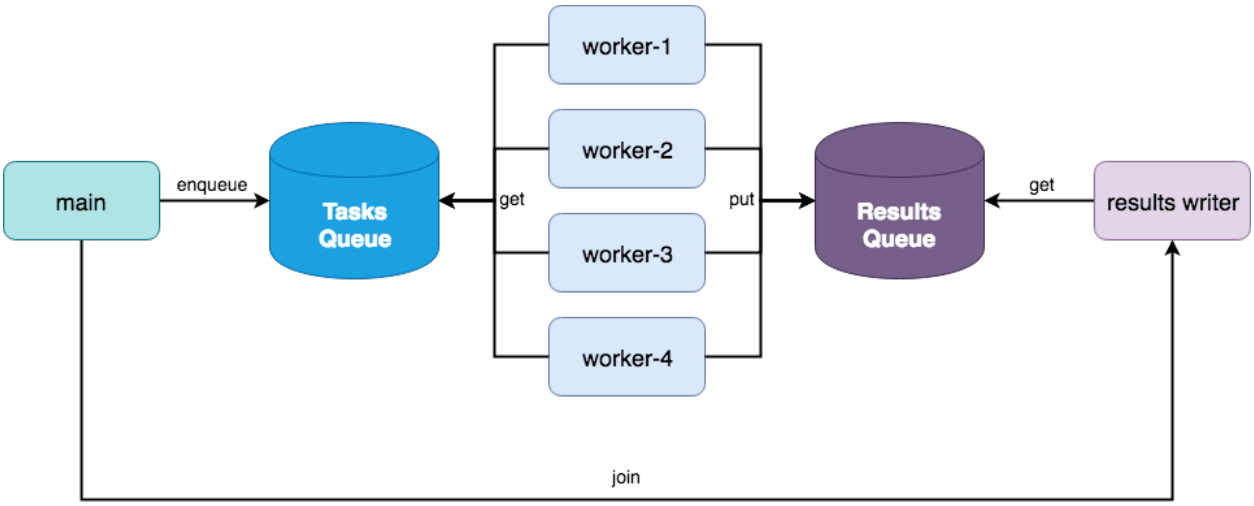


Fig. 1.1: A two queue multiprocessing architecture for data parallel processing.

```
import json
import math
import btrdb
import multiprocessing as mp

from btrdb.utils.timez import ns_delta

# This is just an example method
from qa import data_quality


def time_ranges(stream):
    """
    Returns all 12 hour time ranges for the given stream
    """
    earliest = stream.earliest()[0].time
    latest = stream.latest()[0].time
    hours = int(math.ceil((latest-earliest)/3.6e12))

    for i in range(0, hours, 12):
        start = earliest + ns_delta(hours=i)
        end = start + ns_delta(hours=12)
        yield start, end


def stream_quality(uuid):
    """
    Connects to BTrDB and applies the data quality to 12 hour chunks
    """
    # Connect to DB and get the stream and version
    db = btrdb.connect()
    stream = db.stream_from_uuid(uuid)
    version = stream.version()
```

(continues on next page)

(continued from previous page)

```
    # Get the data quality scores for each 12 hour chunk of data
    quality = []
    for start, end in time_ranges(stream):
        values = stream.values(start=start, end=end, version=version)
        quality.append(data_quality(values))

    # Return the quality scores
    return json.dumps({"uuid": str(uuid), "version": version, "quality": quality})


if __name__ == "__main__":
    # Get the list of streams to get scores for
    db = btrdb.connect()
    streams = db.streams_in_collection("staging/sensors")

    # Create the multiprocessing pool and execute the analytic
    pool = mp.Pool(processes=mp.cpu_count())

    for result in pool.imap_unordered(stream_quality, [s.uuid for s in streams]):
        print(result)
```

Let's break this down quickly since this is a very common design pattern. First the `time_ranges` function gets the earliest and latest timestamp from a stream, then returns all 12 hour intervals between those two timestamps with no overlap. An imaginary `stream_quality` function takes a uuid for a stream, connects to the database and then applies the example `data_quality` method to all 12 hour chunks of data using the `time_ranges` method, returning a JSON string with the results.

The `stream_quality` function is our parallelizable function (e.g. computing the data quality for multiple streams at a time). Depending on how long the `data_quality` function takes to compute, we may also want to parallelize `(stream, start, end)` tuples.

If you would like features like a connection pool object (as other databases have) or multiprocessing helpers, please leave us a note in our GitHub issues!

1.4.8 Multistream Queries

Refer to *True Multistream Support*.

1.4.9 Arrow-enabled Queries

In more recent deployments of the BTrDB platform (>=5.30.0), commercial customers also have access to additional accelerated functionality for data fetching and inserting.

Also, most `StreamSet` based value queries (`AlignedWindows`, `Windows`, `Values`) are multithreaded by default. This leads to decent performance improvements for fetching and inserting data using the standard `StreamSet` api without any edits by the user. Refer to *The StreamSet API*

In addition to these improvements to the standard API, commercial customers also have access to additional accelerated data fetching and inserting methods that can dramatically speed up their workflows.

Apache Arrow Data Format

While keeping our standard API consistent with our `Point` and `StatPoint` *python object model*, we have also created additional methods that will provide this same type of data, but in a tabular format by default. Leveraging the language-agnostic columnar data format Arrow, we can transmit our timeseries data in a format that is already optimized for data analytics with well-defined schemas that take the guesswork out of the data types, timezones, etc. To learn more about these methods, please refer to the *arrow_ prefixed methods for both Stream and StreamSet objects* and the *StreamSet transformer methods*.

True Multistream Support

Until now, there has not been a true multistream query support, our previous api and with the new edits, emulates multistream support with `StreamSet` s and using multithreading. However, this will still only scale to an amount of streams based on the amount of threads that the python threadpool logic can support.

Due to this, raw data queries for `StreamSet` s using our arrow api `StreamSet.filter(start=X, end=Y,).arrow_values()` will now perform true multistream queries. The platform, instead of the python client, will now quickly grab all the stream data for all streams in your streamset, and then package that back to the python client in an `arrow` table! This leads to data fetch speedups on the order of 10-50x based on the amount and kind of streams.

1.4.10 Working with Dash and Plotly

From Plotly's getting start guide: "The plotly Python library is an interactive, open-source plotting library that supports over 40 unique chart types covering a wide range of statistical, financial, geographic, scientific, and 3-dimensional use-cases."

These tools are usable in jupyter notebooks and can also be ran as their own standalone apps using plotly-dash.

Below are two examples using the standard API and the Arrow enabled API to retrieve data as a pandas.DataFrame, and then plotting the results. These examples are based off of the minimal dash app.

Non-Multistream API

```
from dash import Dash, html, dcc, callback, Output, Input
import plotly.express as px
import pandas as pd
import btrdb

conn = btrdb.connect()
streams = conn.streams_in_collection("YOUR_COLLECTION_HERE")
streamset = conn.streams(*[s.uuid for s in streams])
latest = streamset.latest()
end = min([pt.time for pt in latest])
start = end - btrdb.utils.timez.ns_delta(minutes=5)

df = streamset.filter(start=start, end=end).to_dataframe()

app = Dash(__name__)

app.layout = html.Div([
    html.H1(children='Title of Dash App', style={'textAlign':'center'}),
    dcc.Dropdown(df.columns, id='dropdown-selection'),
```

(continues on next page)

(continued from previous page)

```
    dcc.Graph(id='graph-content')
])

@callback(
    Output('graph-content', 'figure'),
    Input('dropdown-selection', 'value')
)
def update_graph(value):
    dff = df[value]
    return px.line(dff, x=dff.index, y=value)

if __name__ == '__main__':
    app.run(debug=True)
```

Multistream API

```
from dash import Dash, html, dcc, callback, Output, Input
import plotly.express as px
import pandas as pd
import btrdb

conn = btrdb.connect()
streams = conn.streams_in_collection("YOUR_COLLECTION_HERE")
streamset = conn.streams(*[s.uuid for s in streams])
latest = streamset.latest()
end = min([pt.time for pt in latest])
start = end - btrdb.utils.timez.ns_delta(minutes=5)

df = streamset.filter(start=start, end=end).arrow_to_dataframe()
df = df.set_index('time')

app = Dash(__name__)

app.layout = html.Div([
    html.H1(children='Title of Dash App', style={'textAlign':'center'}),
    dcc.Dropdown(df.columns, id='dropdown-selection'),
    dcc.Graph(id='graph-content')
])

@callback(
    Output('graph-content', 'figure'),
    Input('dropdown-selection', 'value')
)
def update_graph(value):
    dff = df[value]
    return px.line(dff, x=dff.index, y=value)

if __name__ == '__main__':
    app.run(debug=True)
```

1.4.11 Working with Ray

To use BTrDB connection, stream and streamsets objects in the parallelization library ray, a special serializer is required. BTrDB provides a utility function that register the serializer with ray. An example is shown below.

Setting up the ray serializer

```
import btrdb
import ray
from btrdb.utils.ray import register_serializer

uuids = ["b19592fc-fb71-4f61-9d49-8646d4b1c2a1",
         "07b2cff3-e957-4fa9-b1b3-e14d5afb1e63"]
ray.init()

conn_params = {"profile": "profile_name"}

# register serializer with the connection parameters
register_serializer(**conn_params)

conn = btrdb.connect(**conn_params)

# BTrDB connection object can be passed as an argument
# to a ray remote function
@ray.remote
def test_btrdb(conn):
    print(conn.info())

# Stream object can be passed as an argument
# to a ray remote function
@ray.remote
def test_stream(stream):
    print(stream.earliest())

# StreamSet object can be passed as an argument
# to a ray remote function
@ray.remote
def test_streamset(streamset):
    print(streamset.earliest())
    print(streamset)


ids = [test_btrdb.remote(conn),
    test_stream.remote(conn.stream_from_uuid(uuids[0])),
    test_streamset.remote(conn.streams(*uuids))]

ray.get(ids)
# output of test_btrdb
>>(pid=28479) {'majorVersion': 5, 'build': '5.10.5', 'proxy': {'proxyEndpoints': []}}
# output of test_stream
>>(pid=28482) (RawPoint(1533210100000000000, 0.0), 0)
# output of test_streamset
```

(continues on next page)

(continued from previous page)

```
>>(pid=28481) (RawPoint(1533210100000000000, 0.0), RawPoint(1533210100000000000, 0.0))
>>(pid=28481) StreamSet with 2 streams
```

1.5 BTrDB Explained

The Berkeley Tree DataBase (BTrDB) is pronounced "**Better DB**".

A next-gen timeseries database for dense, streaming telemetry.

Problem: Existing timeseries databases are poorly equipped for a new generation of ultra-fast sensor telemetry. Specifically, millions of high-precision power meters are to be deployed through the power grid to help analyze and prevent blackouts. Thus, new software must be built to facilitate the storage and analysis of its data.

Baseline: We need 1.4M inserts/second and 5x that in reads if we are to support 1000 micro-synchrophasors per server node. No timeseries database can do this.

1.5.1 Summary

Goals: Develop a multi-resolution storage and query engine for many 100+ Hz streams at nanosecond precision—and operate at the full line rate of underlying network or storage infrastructure for affordable cluster sizes (less than six).

Developed at The University of California Berkeley, BTrDB offers new ways to support the aforementioned high throughput demands and allows efficient querying over large ranges.

Fast writes/reads

Measured on a 4-node cluster (large EC2 nodes):

- 53 million inserted values per second
- 119 million queried values per second

Fast analysis

In under *200ms*, it can query a year of data at nanosecond-precision (2.1 trillion points) at any desired window—returning statistical summary points at any desired resolution (containing a min/max/mean per point).

Fig. 1.2: BTrDB enables rapid timeseries queries to support analyses that zoom from years of data to nanosecond granularity smoothly, similar to how you might zoom into a street level view on Google Maps.

High compression

Data is compressed by 2.93x—a significant improvement for high-precision nanosecond streams. To achieve this, a modified version of *run-length encoding* was created to encode the *jitter* of delta values rather than the delta values themselves. Incidentally, this outperforms the popular audio codec FLAC which was the original inspiration for this technique.

Efficient Versioning

Data is version-annotated to allow queries of data as it existed at a certain time. This allows reproducible query results that might otherwise change due to newer realtime data coming in. Structural sharing of data between versions is done to make this process as efficient as possible.

1.5.2 The Tree Structure

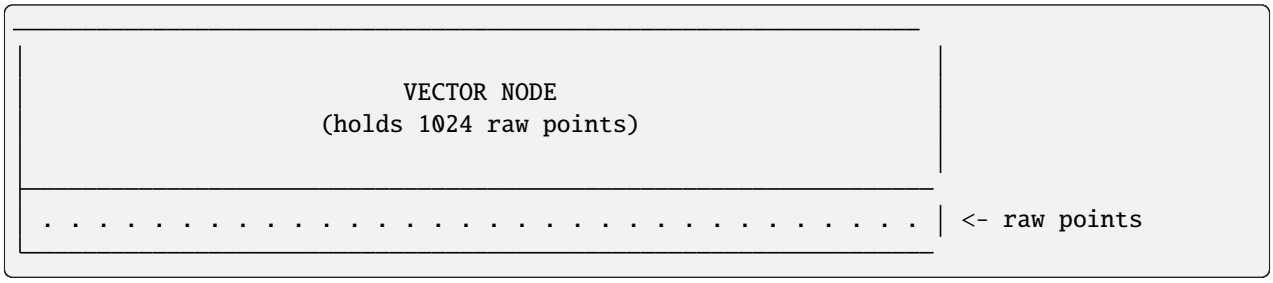
BTrDB stores its data in a time-partitioned tree.

All nodes represent a given time slot. A node can describe all points within its time slot at a resolution corresponding to its depth in the tree.

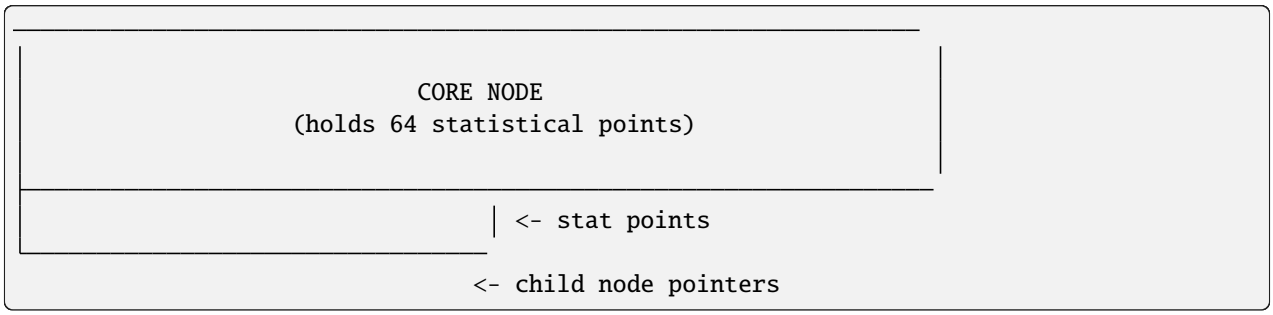
The root node covers ~146 years. With a branching factor of 64, bottom nodes at ten levels down cover 4ns each.

level	node width	time granularity
1	2^{62} ns	~146 years
2	2^{56} ns	~2.28 years
3	2^{50} ns	~13.03 days
4	2^{44} ns	~4.88 hours
5	2^{38} ns	~4.58 minutes
6	2^{32} ns	~4.29 seconds
7	2^{26} ns	~67.11 ms
8	2^{20} ns	~1.05 ms
9	2^{14} ns	~16.38 µs
10	2^{8} ns	256 ns
11	2^{2} ns	4 ns

A node starts as a **vector node**, storing raw points in a vector of size 1024. This is considered a leaf node, since it does not point to any child nodes.:



Once this vector is full and more points need to be inserted into its time slot, the node is converted to a **core node** by time-partitioning itself into 64 "statistical" points.:



A **statistical point** represents a 1/64 slice of its parent's time slot. It holds the min/max/mean/count of all points inside its time slot, and points to a new node holding extra details. When a vector node is first converted to a core node, the raw points are pushed into new vector nodes pointed to by the new statistical points.

level	node width	stat point width	total nodes	total stat points
1	2^{62} ns (~146 years)	2^{56} ns (~2.28 years)	2^{0} nodes	2^{6} points
2	2^{56} ns (~2.28 years)	2^{50} ns (~13.03 days)	2^{6} nodes	2^{12} points
3	2^{50} ns (~13.03 days)	2^{44} ns (~4.88 hours)	2^{12} nodes	2^{18} points
4	2^{44} ns (~4.88 hours)	2^{38} ns (~4.58 min)	2^{18} nodes	2^{24} points
5	2^{38} ns (~4.58 min)	2^{32} ns (~4.29 s)	2^{24} nodes	2^{30} points
6	2^{32} ns (~4.29 s)	2^{26} ns (~67.11 ms)	2^{30} nodes	2^{36} points
7	2^{26} ns (~67.11 ms)	2^{20} ns (~1.05 ms)	2^{36} nodes	2^{42} points
8	2^{20} ns (~1.05 ms)	2^{14} ns (~16.38 µs)	2^{42} nodes	2^{48} points
9	2^{14} ns (~16.38 µs)	2^{8} ns (256 ns)	2^{48} nodes	2^{54} points
10	2^{8} ns (256 ns)	2^{2} ns (4 ns)	2^{54} nodes	2^{60} points
11	2^{2} ns (4 ns)		2^{60} nodes	

The sampling rate of the data at different moments will determine how deep the tree will be during those slices of time. Regardless of the depth of the actual data, the time spent querying at some higher level (lower resolution) will remain fixed (quick) due to summaries provided by parent nodes.

…

1.5.3 Appendix

The original version of this page can be found at:

- github.com/PingThingsIO/btrdb-explained

This page is written based on the following sources:

- Whitepaper
- Code

1.6 API Reference

1.6.1 btrdb

Package for the btrdb database library.

`btrdb.`**`connect`**(*conn_str=None*, *apikey=None*, *profile=None*, *shareable=False*)

Connect to a BTrDB server.

Parameters

- **`conn_str`** (*`str, default=None`*) – The address and port of the cluster to connect to, e.g. `192.168.1.1:4411`. If set to None, will look in the environment variable `$BTRDB_ENDPOINTS` (recommended).
- **`apikey`** (*`str, default=None`*) – The API key used to authenticate requests (optional). If None, the key is looked up from the environment variable `$BTRDB_API_KEY`.
- **`profile`** (*`str, default=None`*) – The name of a profile containing the required connection information as found in the user's predictive grid credentials file `~/.predictivegrid/credentials.yaml`.

- **shareable** (*bool, default=False*) – Whether the connection can be "shared" in a distributed setting such as Ray workers. If set to True, the connection can be serialized and sent to other workers so that data can be retrieved in parallel; **however**, this is less secure because it is possible for other users of the Ray cluster to use your API key to fetch data.

Returns
db – An instance of the BTrDB context to directly interact with the database.

Return type
BTrDB

Examples

This example looks for the env variables: `BTRDB_ENDPOINTS` and `BTRDB_API_KEY`.

```
>>> conn = btrdb.connect()
<btrdb.conn.BTrDB at 0x...>
```

Connect to the platform by looking for the relevant platform profile in `${HOME}/.predictivegrid/credentials.yaml` if the file is present.

```
>>> conn = btrdb.connect(profile='test')
<btrdb.conn.BTrDB at 0x...>
```

If you provide incorrect credentials, you will get an error.

```
>>> conn = btrdb.connect(conn_str="192.168.1.1:4411", apikey="NONSENSICAL_API_KEY")
```

1.6.2 btrdb.conn

Connection related objects for the BTrDB library

class btrdb.conn.**BTrDB**(*endpoint*)

The primary server connection object for communicating with a BTrDB server.

Methods

collection_metadata(prefix[, auto_retry, ...])	Gives statistics about metadata for collections that match a `prefix`.
create(uuid, collection[, tags, ...])	Tells BTrDB to create a new stream with UUID *uuid* in *collection* with specified *tags* and *annotations*.
info()	Returns information about the platform proxy server.
list_collections([starts_with])	Returns a list of collection paths using the *starts_with* argument for filtering.
list_unique_annotations([collection])	Returns a list of annotation keys used in a given collection prefix.
list_unique_names([collection])	Returns a list of names used in a given collection prefix.
list_unique_units([collection])	Returns a list of units used in a given collection prefix.
query(stmt[, params, auto_retry, retries, ...])	Performs a SQL query on the database metadata and returns a list of dictionaries from the resulting cursor.
stream_from_uuid(uuid)	Creates a stream handle to the BTrDB stream with the UUID *uuid*.
streams(*identifiers[, versions, ...])	Returns a StreamSet object with BTrDB streams from the supplied identifiers.
streams_in_collection(*collection[, ...])	Search for streams matching given parameters

collection_metadata(*prefix*, *auto_retry=False*, *retries=5*, *retry_delay=3*, *retry_backoff=4*)

Gives statistics about metadata for collections that match a `prefix`.

Parameters

- **prefix** (*str, required*) – A prefix of the collection names to look at
- **auto_retry** (*bool, default: False*) – Whether to retry this request in the event of an error
- **retries** (*int, default: 5*) – Number of times to retry this request if there is an error. Will be ignored if auto_retry is False
- **retry_delay** (*int, default: 3*) – initial time to wait before retrying function call if there is an error. Will be ignored if auto_retry is False
- **retry_backoff** (*int, default: 4*) – Exponential factor by which the backoff increases between retries. Will be ignored if auto_retry is False

Returns

A tuple of dictionaries containing metadata on the streams in the provided collection.

Return type

tuple

Examples

```
>>> conn.collection_metadata("sunshine/PMU1")
({'name': 0, 'unit': 0, 'ingress': 0, 'distiller': 0},
.. {'foo': 1, 'impedance': 12, 'location': 12})
```

```
>>> conn.collection_metadata("sunshine/")
({'name': 0, 'unit': 0, 'ingress': 0, 'distiller': 0},
.. {'foo': 1, 'impedance': 72, 'location': 72})
```

create(*uuid*, *collection*, *tags=None*, *annotations=None*, *auto_retry=False*, *retries=5*, *retry_delay=3*, *retry_backoff=4*)

Tells BTrDB to create a new stream with UUID *uuid* in *collection* with specified *tags* and *annotations*.

Parameters

- **uuid** (*UUID, required*) – The uuid of the requested stream.
- **collection** (*str, required*) – The collection string prefix that the stream will belong to.
- **tags** (*dict, required*) – The tags-level metadata key:value pairs.
- **annotations** (*dict, optional*) – The mutable metadata of the stream, key:value pairs
- **auto_retry** (*bool, default: False*) – Whether to retry this request in the event of an error
- **retries** (*int, default: 5*) – Number of times to retry this request if there is an error. Will be ignored if auto_retry is False
- **retry_delay** (*int, default: 3*) – initial time to wait before retrying function call if there is an error. Will be ignored if auto_retry is False
- **retry_backoff** (*int, default: 4*) – Exponential factor by which the backoff increases between retries. Will be ignored if auto_retry is False

Returns

instance of Stream class

Return type

Stream

Examples

```
>>> import btrdb
>>> from uuid import uuid4 # this generates a random uuid
>>> conn = btrdb.connect()
>>> collection = "new/stream/collection"
>>> tags = {"name":"foo", "unit":"V"}
>>> annotations = {"bar": "baz"}
>>> s = conn.create(uuid=uuid4(), tags=tags, annotations=annotations,␣
↪collection=collection)
<Stream collection=new/stream/collection name=foo>
```

info()

Returns information about the platform proxy server.

Returns
Proxy server connection and status information

Return type
dict

Examples

```
>>> conn = btrdb.connect()
>>> conn.info()
{
..        'majorVersion': 5,
..        'minorVersion': 8,
..        'build': ...,
..        'proxy': ...,
}
```

`list_collections`(*starts_with=''*)

Returns a list of collection paths using the *starts_with* argument for filtering.

Parameters
starts_with (*str, optional, default: ''*) – Filter collections that start with the string provided, if none passed, will list all collections.

Returns
collections

Return type
List[str]

Examples

Assuming we have the following collections in the platform: `foo`, `bar`, `foo/baz`, `bar/baz`

```
>>> conn = btrdb.connect()
>>> conn.list_collections().sort()
["bar", "bar/baz", "foo", "foo/bar"]
```

```
>>> conn.list_collections(starts_with="foo")
["foo", "foo/bar"]
```

`list_unique_annotations`(*collection=None*)

Returns a list of annotation keys used in a given collection prefix.

Parameters
collection (*str*) – Prefix of the collection to filter.

Returns
annotations

Return type
list[str]

Notes

This query treats the `collection` string as a prefix, so `collection="foo"` will match with the following wildcard syntax `foo%`. If you only want to filter for a single collection, you will need to provide the full collection, if there are other collections that match the `foo%` pattern, you might need to use a custom SQL query using `conn.query`.

Examples

```
>>> conn.list_unique_annotations(collection="sunshine/PMU1")
['foo', 'location', 'impedance']
```

list_unique_names(*collection=None*)

Returns a list of names used in a given collection prefix.

Parameters
collection (*str*) – Prefix of the collection to filter.

Returns
names

Return type
list[str]

Examples

Can specify a full `collection` name.

```
>>> conn.list_unique_names(collection="sunshine/PMU1")
['C1ANG', 'C1MAG', 'C2ANG', 'C2MAG', 'C3ANG', 'C3MAG', 'L1ANG', 'L1MAG', 'L2ANG
↪', 'L2MAG', 'L3ANG', 'L3MAG', 'LSTATE']
```

And also provide a `collection` prefix.

```
>>> conn.list_unique_names(collection="sunshine/")
['C1ANG', 'C1MAG', 'C2ANG', 'C2MAG', 'C3ANG', 'C3MAG', 'L1ANG', 'L1MAG', 'L2ANG
↪', 'L2MAG', 'L3ANG', 'L3MAG', 'LSTATE']
```

list_unique_units(*collection=None*)

Returns a list of units used in a given collection prefix.

Parameters
collection (*str*) – Prefix of the collection to filter.

Returns
units

Return type
list[str]

Examples

```
>>> conn.list_unique_units(collection="sunshine/PMU1")
['amps', 'deg', 'mask', 'volts']
```

query(*stmt: str*, *params: Tuple[str] | List[str] = None*, *auto_retry=False*, *retries=5*, *retry_delay=3*, *retry_backoff=4*)

Performs a SQL query on the database metadata and returns a list of dictionaries from the resulting cursor.

Parameters

- **stmt** (*str*) – a SQL statement to be executed on the BTrDB metadata. Available tables are noted below. To sanitize inputs use a *$1* style parameter such as *select * from streams where name = $1 or name = $2*.
- **params** (*list or tuple*) – a list of parameter values to be sanitized and interpolated into the SQL statement. Using parameters forces value/type checking and is considered a best practice at the very least.
- **auto_retry** (*bool, default: False*) – Whether to retry this request in the event of an error
- **retries** (*int, default: 5*) – Number of times to retry this request if there is an error. Will be ignored if auto_retry is False
- **retry_delay** (*int, default: 3*) – initial time to wait before retrying function call if there is an error. Will be ignored if auto_retry is False
- **retry_backoff** (*int, default: 4*) – Exponential factor by which the backoff increases between retries. Will be ignored if auto_retry is False

Returns
a list of dictionary object representing the cursor results.

Return type
list

Notes

Parameters will be inserted into the SQL statement as noted by the parameter number such as *$1*, *$2*, or *$3*. The *streams* table is available for *SELECT* statements only.

See https://btrdb.readthedocs.io/en/latest/ for more info.

The following are the queryable columns in the postgres `streams` table.

Column	Type	Nullable
uuid	uuid	not null
collection	character varying(256)	not null
name	character varying(256)	not null
unit	character varying(256)	not null
ingress	character varying(256)	not null
property_version	bigint	not null
annotations	hstore	

Examples

Count all streams in the platform.

```
>>> conn = btrdb.connect()
>>> conn.query("SELECT COUNT(uuid) FROM streams")
[{'count': ...}]
```

Count all streams in the collection `foo/bar` by passing in the variable as a parameter.

```
>>> conn.query("SELECT COUNT(uuid) FROM streams WHERE collection=$1::text",␣
↪params=["foo/bar"])
[{'count': ...}]
```

Count all streams in the platform that has a non-null entry for the metadata annotation `foo`.

```
>>> conn.query("SELECT COUNT(uuid) FROM streams WHERE annotations->$1::text IS␣
↪NOT NULL", params=["foo"])
[{'count': ...}]
```

stream_from_uuid(*uuid*)

Creates a stream handle to the BTrDB stream with the UUID *uuid*. This method does not check whether a stream with the specified UUID exists. It is always good form to check whether the stream existed using *stream.exists()*.

Parameters
uuid (*UUID*) – The uuid of the requested stream.

Returns
instance of Stream class or None

Return type
Stream

Examples

```
>>> import btrdb
>>> conn = btrdb.connect()
>>> uuid = "f98f4b4e-9fab-46b5-8a80-f282059d69b1"
>>> stream = conn.stream_from_uuid(uuid)
>>> stream
<Stream collection=foo/test name=test_stream>
```

streams(**identifiers*, *versions=None*, *is_collection_prefix=False*)

Returns a StreamSet object with BTrDB streams from the supplied identifiers. If any streams cannot be found matching the identifier then a `StreamNotFoundError` will be returned.

Parameters

- **identifiers** (*str or UUID*) – a single item or iterable of items which can be used to query for streams. Identifiers are expected to be UUID as string, UUID as UUID, or collection/name string.
- **versions** (*list[int]*) – a single or iterable of version numbers to match the identifiers

- **is_collection_prefix** (*bool, default=False*) – If providing a collection string, is that string just a prefix, or the entire collection name? This will impact how many streams are returned.

Returns
Collection of streams.

Return type
StreamSet

Examples

With a sequence of uuids.

```
>>> conn = btrdb.connect()
>>> conn.streams(identifiers=list_of_uuids)
<btrdb.stream.StreamSet at 0x...>
```

With a sequence of uuids and version numbers. Here we are using version 0 to use the latest data points.

```
>>> conn.streams(identifiers=list_of_uuids, versions=[0 for _ in list_of_uuids])
<btrdb.stream.StreamSet at 0x...>
```

Filtering by `collection` prefix `"foo"` where multiple collections exist like the following: `foo/bar`, `foo/baz`, `foo/bar/new`, and `foo`. If we set *is_collection_prefix* ` to `True`, this will return all streams that exist in the collections defined above. It is similar to a regex pattern `^foo.*` for matching purposes.

```
>>> conn.streams(identifiers="foo", is_collection_prefix=True)
<btrdb.stream.StreamSet at 0x...>
```

If you set `is_collection_prefix` to `False`, this will assume that the string identifier you provide is the full collection name. Matching like the regex here: `^foo`

```
>>> conn.streams(identifiers="foo", is_collection_prefix=False)
<btrdb.stream.StreamSet at 0x...>
```

streams_in_collection(**collection*, *is_collection_prefix=True*, *tags=None*, *annotations=None*, *auto_retry=False*, *retries=5*, *retry_delay=3*, *retry_backoff=4*)

Search for streams matching given parameters

This function allows for searching

Parameters

- **collection** (*str*) – collections to use when searching for streams, case sensitive.
- **is_collection_prefix** (*bool*) – Whether the collection is a prefix.
- **tags** (*Dict[str, str]*) – The tags to identify the stream.
- **annotations** (*Dict[str, str]*) – The annotations to identify the stream.
- **auto_retry** (*bool, default: False*) – Whether to retry this request in the event of an error
- **retries** (*int, default: 5*) – Number of times to retry this request if there is an error. Will be ignored if auto_retry is False

- **retry_delay** (*int, default: 3*) – initial time to wait before retrying function call if there is an error. Will be ignored if auto_retry is False
- **retry_backoff** (*int, default: 4*) – Exponential factor by which the backoff increases between retries. Will be ignored if auto_retry is False

Returns
A list of `Stream` objects found with the provided search arguments.

Return type
list[*Stream*]

Note: In a future release, the default return value of this function will be a `StreamSet`

Examples

```
>>> conn = btrdb.connect()
>>> conn.streams_in_collection(collection="foo", is_collection_prefix=True)
[<Stream collection=foo name=test1>, <Stream collection=foo name=test2,
... <Stream collection=foo/bar, name=testX>, <Stream collection=foo/baz/bar␣
↪name=testY>]
```

```
>>> conn.streams_in_collection(collection="foo", is_collection_prefix=False)
[<Stream collection=foo, name=test1>, <Stream collection=foo, name=test2>]
```

```
>>> conn.streams_in_collection(collection="foo",
...  is_collection_prefix=False, tags={"unit":"Volts"})
[<Stream collection=foo, name=test1>]
```

```
>>> conn.streams_in_collection(collection="foo",
...  is_collection_prefix=False, tags={"unit":"UNKNOWN"})
[]
```

1.6.3 btrdb.stream

Module for Stream and related classes

class `btrdb.stream.`**Stream**(*btrdb*, *uuid*, ***db_values*)

An object that represents a specific time series stream in the BTrDB database.

Parameters

- **btrdb** (BTrDB) – A reference to the BTrDB object connecting this stream back to the physical server.
- **uuid** (*UUID*) – The unique UUID identifier for this stream.
- **db_values** (*kwargs*) – Framework only initialization arguments. Not for developer use.

Attributes

btrdb
Returns the stream's BTrDB object.

collection
Returns the collection of the stream.

name
Returns the stream's name which is parsed from the stream tags.

unit
Returns the stream's unit which is parsed from the stream tags.

uuid
Returns the stream's UUID.

Methods

aligned_windows(start, end, pointwidth[, ...])	Read statistical aggregates of windows of data from BTrDB.
annotations([refresh, auto_retry, retries, ...])	Returns a stream's annotations
arrow_aligned_windows(start, end, pointwidth)	Read statistical aggregates of windows of data from BTrDB.
arrow_insert(data[, merge])	Insert new data in the form of a pyarrow Table with (time, value) columns.
arrow_values(start, end[, version, ...])	Read raw values from BTrDB between time [a, b) in nanoseconds.
arrow_windows(start, end, width[, version, ...])	Read arbitrarily-sized windows of data from BTrDB.
count([start, end, pointwidth, precise, version])	Compute the total number of points in the stream
current([version, auto_retry, retries, ...])	Returns the point that is closest to the current timestamp, e.g. the latest point in the stream up until now.
delete(start, end[, auto_retry, retries, ...])	"Delete" all points between [`start`, `end`)
earliest([version, auto_retry, retries, ...])	Returns the first point of data in the stream.
exists()	Check if stream exists
flush([auto_retry, retries, retry_delay, ...])	Flush writes the stream buffers out to persistent storage.
insert(data[, merge])	Insert new data in the form (time, value) into the series.
latest([version, auto_retry, retries, ...])	Returns last point of data in the stream.
nearest(time, version[, backward, ...])	Finds the closest point in the stream to a specified time.
obliterate([auto_retry, retries, ...])	Obliterate deletes a stream from the BTrDB server.
refresh_metadata()	Refreshes the locally cached metadata for a stream from the server.
tags([refresh, auto_retry, retries, ...])	Returns the stream's tags.
update([tags, annotations, collection, ...])	Updates metadata including tags, annotations, and collection as an UPSERT operation.
values(start, end[, version, auto_retry, ...])	Read raw values from BTrDB between time [a, b) in nanoseconds.
version([auto_retry, retries, retry_delay, ...])	Returns the current data version of the stream.
windows(start, end, width[, depth, version, ...])	Read arbitrarily-sized windows of data from BTrDB.

aligned_windows(*start*, *end*, *pointwidth*, *version=0*, *auto_retry=False*, *retries=5*, *retry_delay=3*, *retry_backoff=4*)

Read statistical aggregates of windows of data from BTrDB.

Query BTrDB for aggregates (or roll ups or windows) of the time series with *version* between time `start` (inclusive) and `end` (exclusive) in nanoseconds. Each point returned is a statistical aggregate of all the

raw data within a window of width `2**pointwidth` nanoseconds. These statistical aggregates currently include the mean, minimum, and maximum of the data and the count of data points composing the window.

`start` is inclusive, but `end` is exclusive. That is, results will be returned for all windows that start in the interval [start, end). If end < start+2^pointwidth you will not get any results. If start and end are not powers of two, the bottom `pointwidth` bits will be cleared. Each window will contain statistical summaries of the window. Statistical points with count == 0 will be omitted.

Parameters

- **start** (*int or datetime like object*) – The start time in nanoseconds for the range to be queried. (see *btrdb.utils.timez.to_nanoseconds()* for valid input types)
- **end** (*int or datetime like object*) – The end time in nanoseconds for the range to be queried. (see *btrdb.utils.timez.to_nanoseconds()* for valid input types)
- **pointwidth** (*int*) – Specify the number of ns between data points (2**pointwidth)
- **version** (*int*) – Version of the stream to query
- **auto_retry** (*bool, default: False*) – Whether to retry this request in the event of an error
- **retries** (*int, default: 5*) – Number of times to retry this request if there is an error. Will be ignored if auto_retry is False
- **retry_delay** (*int, default: 3*) – initial time to wait before retrying function call if there is an error. Will be ignored if auto_retry is False
- **retry_backoff** (*int, default: 4*) – Exponential factor by which the backoff increases between retries. Will be ignored if auto_retry is False

Returns

Returns a tuple containing windows of data. Each window is a tuple containing data tuples. Each data tuple contains a StatPoint and the stream version.

Return type

tuple

Note: As the window-width is a power-of-two, it aligns with `BTrDB` internal tree data structure and is faster to execute than `windows()`.

---

annotations(*refresh=False*, *auto_retry=False*, *retries=5*, *retry_delay=3*, *retry_backoff=4*)

Returns a stream's annotations

Annotations returns the annotations of the stream (and the annotation version).

Do not modify the resulting map.

Parameters

- **refresh** (*bool, default: False*) – Indicates whether a round trip to the server should be implemented regardless of whether there is a local copy.
- **auto_retry** (*bool, default: False*) – Whether to retry this request in the event of an error
- **retries** (*int, default: 5*) – Number of times to retry this request if there is an error. Will be ignored if auto_retry is False
- **retry_delay** (*int, default: 3*) – initial time to wait before retrying function call if there is an error. Will be ignored if auto_retry is False

- **retry_backoff** (*int, default: 4*) – Exponential factor by which the backoff increases between retries. Will be ignored if auto_retry is False

Returns
A tuple containing a dictionary of annotations and an integer representing the version of the metadata (tuple(dict, int)).

Return type
tuple

Note: This `version` is not the same as the `stream.version`.

Examples

Accessing a streams annotations.

```
>>> stream.annotations()
({"foo":"bar", "baz":"bazaar"}, 231)
```

Extract the version and metadata separately.

```
>>> annotations, metadata_version = stream.annotations()
>>> annotations
{"foo":"bar", "baz":"bazaar"}
>>> metadata_version
231
```

arrow_aligned_windows(*start: int*, *end: int*, *pointwidth: int*, *version: int = 0*, *auto_retry=False*, *retries=5*, *retry_delay=3*, *retry_backoff=4*) → Table

Read statistical aggregates of windows of data from BTrDB.

Query BTrDB for aggregates (or roll ups or windows) of the time series with *version* between time *start* (inclusive) and *end* (exclusive) in nanoseconds [start, end). Each point returned is a statistical aggregate of all the raw data within a window of width 2**`pointwidth` nanoseconds. These statistical aggregates currently include the mean, minimum, and maximum of the data and the count of data points composing the window.

Note: `start` is inclusive, but `end` is exclusive. That is, results will be returned for all windows that start in the interval [start, end). If end < start+2^pointwidth you will not get any results. If start and end are not powers of two, the bottom `pointwidth` bits will be cleared. Each window will contain statistical summaries of the window. Statistical points with `count == 0` will be omitted.

Parameters

- **start** (*int or datetime like object, required*) – The start time in nanoseconds for the range to be queried. (see :func:`btrdb.utils.timez.to_nanoseconds` for valid input types)
- **end** (*int or datetime like object, required*) – The end time in nanoseconds for the range to be queried. (see :func:`btrdb.utils.timez.to_nanoseconds` for valid input types)

- **pointwidth** (*int, required*) – Specify the number of ns between data points (2**pointwidth)
- **version** (*int, default: 0*) – Version of the stream to query
- **auto_retry** (*bool, default: False*) – Whether to retry this request in the event of an error
- **retries** (*int, default: 5*) – Number of times to retry this request if there is an error. Will be ignored if auto_retry is False
- **retry_delay** (*int, default: 3*) – initial time to wait before retrying function call if there is an error. Will be ignored if auto_retry is False
- **retry_backoff** (*int, default: 4*) – Exponential factor by which the backoff increases between retries. Will be ignored if auto_retry is False

Returns
Returns a pyarrow table containing the windows of data.

Return type
pyarrow.Table

Note: As the window-width is a power-of-two, it aligns with BTrDB internal tree data structure and is faster to execute than *windows()*.

Note: This method is available for commercial customers with arrow-enabled servers.

---

arrow_insert(*data: Table*, *merge: str = 'never'*) → int

Insert new data in the form of a pyarrow Table with (time, value) columns.

Inserts a table of new (time, value) columns into the stream. The values in the table need not be sorted by time. If the arrays are larger than appropriate, this function will automatically chunk the inserts. As a consequence, the insert is not necessarily atomic, but can be used with a very large array.

Parameters

- **data** (*pyarrow.Table, required*) – A pyarrow table with a schema of time:Timestamp[ns, tz=UTC], value:float64 This schema will be validated and converted if necessary.
- **merge** (*str*) –

  A string describing the merge policy. Valid policies are:

  - 'never': the default, no points are merged
  - 'equal': points are deduplicated if the time and value are equal
  - 'retain': if two points have the same timestamp, the old one is kept
  - 'replace': if two points have the same timestamp, the new one is kept

Returns
The version of the stream after inserting new points.

Return type
int

Note: This method is available for commercial customers with arrow-enabled servers.

Examples

Assuming we have a sequence of `times` and `values` where `times` are in nanoseconds. Insert the data as a pyarrow table, and if there are duplicate timestamps already in the database, replace with the new ones in `payload`.

```
>>> conn = btrdb.connect()
>>> import pyarrow as pa
>>> for t, v in zip(times, vals):
...     print(t,v)
1500000000000000000 1.0
1500000000100000000 2.0
1500000000200000000 3.0
1500000000300000000 4.0
1500000000400000000 5.0
1500000000500000000 6.0
1500000000600000000 7.0
1500000000700000000 8.0
1500000000800000000 9.0
1500000000900000000 10.0
>>> schema = pa.schema(
... [
...    pa.field("time", pa.timestamp("ns", tz="UTC"), nullable=False),
...    pa.field("value", pa.float64(), nullable=False),
... ]
... )
>>> payload = pa.Table.from_arrays([times, vals], schema=schema)
>>> version = stream.arrow_insert(payload, merge="replace")
```

arrow_values(*start*, *end*, *version: int = 0*, *auto_retry=False*, *retries=5*, *retry_delay=3*, *retry_backoff=4*, *schema=None*) → Table

Read raw values from BTrDB between time [a, b) in nanoseconds.

RawValues queries BTrDB for the raw time series data points between *start* and *end* time, both in nanoseconds since the Epoch for the specified stream *version*.

start
: [int or datetime like object] The start time in nanoseconds for the range to be queried. (see *btrdb.utils.timez.to_nanoseconds()* for valid input types)

end
: [int or datetime like object] The end time in nanoseconds for the range to be queried. (see *btrdb.utils.timez.to_nanoseconds()* for valid input types)

version: int, default: 0
: The version of the stream to be queried

schema: pyarrow.Schema
: Optional arrow schema the server will cast the returned data to before sending it over the network. You can use this to change the timestamp format, column names or data sizes.

auto_retry: bool, default: False
Whether to retry this request in the event of an error

retries: int, default: 5
Number of times to retry this request if there is an error. Will be ignored if auto_retry is False

retry_delay: int, default: 3
initial time to wait before retrying function call if there is an error. Will be ignored if auto_retry is False

retry_backoff: int, default: 4
Exponential factor by which the backoff increases between retries. Will be ignored if auto_retry is False

Returns
A pyarrow table of the raw values with time and value columns.

Return type
pyarrow.Table

Note: Note that the raw data points are the original values at the sensor's native sampling rate (assuming the time series represents measurements from a sensor). This is the lowest level of data with the finest time granularity. In the tree data structure of BTrDB, this data is stored in the vector nodes.

Note: This method is available for commercial customers with arrow-enabled servers.

---

arrow_windows(*start: int*, *end: int*, *width: int*, *version: int = 0*, *auto_retry=False*, *retries=5*, *retry_delay=3*, *retry_backoff=4*) → Table

Read arbitrarily-sized windows of data from BTrDB.

Parameters

- **start** (*int or datetime like object, required*) – The start time in nanoseconds for the range to be queried. (see *btrdb.utils.timez.to_nanoseconds()* for valid input types)
- **end** (*int or datetime like object, required*) – The end time in nanoseconds for the range to be queried. (see *btrdb.utils.timez.to_nanoseconds()* for valid input types)
- **width** (*int, required*) – The number of nanoseconds in each window.
- **version** (*int, default=0, optional*) – The version of the stream to query.
- **auto_retry** (*bool, default: False*) – Whether to retry this request in the event of an error
- **retries** (*int, default: 5*) – Number of times to retry this request if there is an error. Will be ignored if auto_retry is False
- **retry_delay** (*int, default: 3*) – initial time to wait before retrying function call if there is an error. Will be ignored if auto_retry is False
- **retry_backoff** (*int, default: 4*) – Exponential factor by which the backoff increases between retries. Will be ignored if auto_retry is False

Returns
Returns a pyarrow Table containing windows of data.

Return type
pyarrow.Table

Note: `windows` returns arbitrary precision windows from `BTrDB`. It is slower than `aligned_windows`, but still significantly faster than RawValues. Each returned window will be `width` nanoseconds long. `start` is inclusive, but `end` is exclusive (e.g if end < start+width you will get no results). That is, results will be returned for all windows that start at a time less than the `end` timestamp. If (`end` - `start`) is not a multiple of `width`, then `end` will be decreased to the greatest value less than `end` such that (`end` - `start`) is a multiple of *width* (i.e., we set `end = start + width * floordiv(end - start, width)`. The `depth` parameter previously available has been deprecated. The only valid value for `depth` is now 0.

Note: This method is available for commercial customers with arrow-enabled servers.

`property btrdb`

Returns the stream's BTrDB object.

Parameters

None

Returns

BTrDB
The BTrDB database object.

Examples

```
>>> import btrdb
>>> conn = btrdb.connect()
>>> stream = conn.stream_from_uuid("...")
>>> btrdb_obj = stream.btrdb
>>> btrdb_obj
<btrdb.conn.BTrDB object at 0x...>
```

`property collection`

Returns the collection of the stream. It may require a round trip to the server depending on how the stream was acquired.

Parameters
None

Returns
the collection of the stream

Return type
str

Examples

```
>>> import btrdb
>>> conn = btrdb.connect()
>>> stream = conn.stream_from_uuid("...")
>>> stream.collection
'foo/bar'
```

count(*start=-115292150460684697*6, end=3458764513820540927, pointwidth=62, precise=False, version=0*)

Compute the total number of points in the stream

Counts the number of points in the specified window and version. By default, returns the latest total count of points in the stream.

Parameters

- **start** (*int or datetime like object, default: MINIMUM_TIME*) – The start time in nanoseconds for the range to be queried. (see *btrdb.utils.timez.to_nanoseconds()* for valid input types)
- **end** (*int or datetime like object, default: MAXIMUM_TIME*) – The end time in nanoseconds for the range to be queried. (see *btrdb.utils.timez.to_nanoseconds()* for valid input types)
- **pointwidth** (*int, default: 62*) – Specify the number of ns between data points (2**pointwidth). If the value is too large for the time window than the next smallest, appropriate pointwidth will be used.
- **precise** (*bool, default: False*) – Forces the use of a windows query instead of aligned_windows to determine exact count down to the nanosecond. This will be some amount slower than the aligned_windows version.
- **version** (*int, default: 0*) – Version of the stream to query

Returns

The total number of points in the stream for the specified window.

Return type

int

Note: This helper method sums the counts of all StatPoints returned by `aligned_windows`. Because of this, note that the start and end timestamps may be adjusted if they are not powers of 2. For smaller windows of time, you may also need to adjust the `pointwidth` to ensure that the count granularity is captured appropriately.

Alternatively you can set the `precise` argument to `True` which will give an exact count to the nanosecond but may be slower to execute.

Examples

```
>>> import btrdb
>>> conn = btrdb.connect()
>>> stream = conn.stream_from_uuid("...")
>>> stream.count()
1234
>>> stream.count(start=150000000000000000, end=160368000000000000,␣
↪pointwidth=55)
567
>>> stream.count(start=150000000000000000, end=160368000000000000,␣
↪precise=True)
789
```

current(*version=0*, *auto_retry=False*, *retries=5*, *retry_delay=3*, *retry_backoff=4*)

Returns the point that is closest to the current timestamp, e.g. the latest point in the stream up until now. Note that no future values will be returned. Returns None if errors occur during lookup or there are no values before now.

Parameters

- **version** (*int, default: 0*) – Specify the version of the stream to query; if zero, queries the latest stream state rather than pinning to a version.
- **auto_retry** (*bool, default: False*) – Whether to retry this request in the event of an error
- **retries** (*int, default: 5*) – Number of times to retry this request if there is an error. Will be ignored if auto_retry is False
- **retry_delay** (*int, default: 3*) – initial time to wait before retrying function call if there is an error. Will be ignored if auto_retry is False
- **retry_backoff** (*int, default: 4*) – Exponential factor by which the backoff increases between retries. Will be ignored if auto_retry is False

Returns

The last data point in the stream up until now and the version of the stream the value was retrieved at (tuple(RawPoint, int)).

Return type

tuple

delete(*start*, *end*, *auto_retry=False*, *retries=5*, *retry_delay=3*, *retry_backoff=4*)

"Delete" all points between [start, end)

"Delete" all points between start (inclusive) and end (exclusive), both in nanoseconds.

Note: As BTrDB has persistent multiversioning, the deleted points will still exist as part of an older version of the stream.

Parameters

- **start** (*int or datetime like object*) – The start time in nanoseconds for the range to be deleted. (see :func:btrdb.utils.timez.to_nanoseconds for valid input types)

- **end** (*int or datetime like object*) – The end time in nanoseconds for the range to be deleted. (see :func:`btrdb.utils.timez.to_nanoseconds` for valid input types)
- **auto_retry** (*bool, default: False*) – Whether to retry this request in the event of an error
- **retries** (*int, default: 5*) – Number of times to retry this request if there is an error. Will be ignored if auto_retry is False
- **retry_delay** (*int, default: 3*) – initial time to wait before retrying function call if there is an error. Will be ignored if auto_retry is False
- **retry_backoff** (*int, default: 4*) – Exponential factor by which the backoff increases between retries. Will be ignored if auto_retry is False

Returns

The version of the new stream created

Return type

int

Examples

```
>>> import btrdb
>>> conn = btrdb.connect()
>>> stream = conn.stream_from_uuid("...")
>>> start = 1500000000000000000
>>> end = 1500000001000000000
>>> stream.delete(start, end)
1234
>>> stream.count(start=start, end=end)
0
```

earliest(*version=0*, *auto_retry=False*, *retries=5*, *retry_delay=3*, *retry_backoff=4*)

Returns the first point of data in the stream. Returns None if error encountered during lookup or no values in stream.

Parameters

- **version** (*int, default: 0*) – Specify the version of the stream to query; if zero, queries the latest stream state rather than pinning to a version.
- **auto_retry** (*bool, default: False*) – Whether to retry this request in the event of an error
- **retries** (*int, default: 5*) – Number of times to retry this request if there is an error. Will be ignored if auto_retry is False
- **retry_delay** (*int, default: 3*) – initial time to wait before retrying function call if there is an error. Will be ignored if auto_retry is False
- **retry_backoff** (*int, default: 4*) – Exponential factor by which the backoff increases between retries. Will be ignored if auto_retry is False

Returns

The first data point in the stream and the version of the stream the value was retrieved at (tuple(RawPoint, int)).

Return type
tuple

Examples

Get the earliest point for a stream using `version 0`.

```
>>> stream.earliest(version=0)
(<btrdb.point.RawPoint at 0x...>, 1234567)
```

Extract just the `RawPoint` data.

```
>>> pt, _ = stream.earliest(version=0)
>>> print(pt.time, pt.value)
1547241923338098176 123.7
```

exists()

Check if stream exists

Exists returns true if the stream exists. This is essential after using StreamFromUUID as the stream may not exist, causing a 404 error on later stream operations. Any operation that returns a stream from collection and tags will have ensured the stream exists already.

Parameters
None

Returns
Indicates whether stream is extant in the BTrDB server.

Return type
bool

Examples

```
>>> import btrdb
>>> conn = btrdb.connect()
>>> stream = conn.stream_from_uuid("...")
>>> stream.uuid
UUID('...')
>>> stream.exists()
True
```

flush(*auto_retry=False*, *retries=5*, *retry_delay=3*, *retry_backoff=4*)

Flush writes the stream buffers out to persistent storage.

Parameters

- **auto_retry** (*bool, default: False*) – Whether to retry this request in the event of an error
- **retries** (*int, default: 5*) – Number of times to retry this request if there is an error. Will be ignored if auto_retry is False
- **retry_delay** (*int, default: 3*) – initial time to wait before retrying function call if there is an error. Will be ignored if auto_retry is False

- **retry_backoff** (*int, default: 4*) – Exponential factor by which the backoff increases between retries. Will be ignored if auto_retry is False

insert(*data*, *merge='never'*)

Insert new data in the form (time, value) into the series.

Inserts a list of new (time, value) tuples into the series. The tuples in the list need not be sorted by time. If the arrays are larger than appropriate, this function will automatically chunk the inserts. As a consequence, the insert is not necessarily atomic, but can be used with a very large array.

Parameters

- **data** (*list[tuple[int, float]]*) – A list of tuples in which each tuple contains a time (int) and value (float) for insertion to the database
- **merge** (*str*) –

  A string describing the merge policy. Valid policies are:

  - 'never': the default, no points are merged
  - 'equal': points are deduplicated if the time and value are equal
  - 'retain': if two points have the same timestamp, the old one is kept
  - 'replace': if two points have the same timestamp, the new one is kept

Returns

The version of the stream after inserting new points.

Return type

int

Examples

```
>>> import btrdb
>>> conn = btrdb.connect()
>>> stream = conn.stream_from_uuid("...")
>>> data = [(1500000000000000000, 1.0), (1500000000100000000, 2.0)]
>>> stream.insert(data)
1234
>>> stream.insert(data, merge="replace")
1235
```

latest(*version=0*, *auto_retry=False*, *retries=5*, *retry_delay=3*, *retry_backoff=4*)

Returns last point of data in the stream. Returns None if error encountered during lookup or no values in stream.

Parameters

- **version** (*int, default: 0*) – Specify the version of the stream to query; if zero, queries the latest stream state rather than pinning to a version.
- **auto_retry** (*bool, default: False*) – Whether to retry this request in the event of an error
- **retries** (*int, default: 5*) – Number of times to retry this request if there is an error. Will be ignored if auto_retry is False
- **retry_delay** (*int, default: 3*) – initial time to wait before retrying function call if there is an error. Will be ignored if auto_retry is False

- **retry_backoff** (*int, default: 4*) – Exponential factor by which the backoff increases between retries. Will be ignored if auto_retry is False

Returns
The last data point in the stream and the version of the stream the value was retrieved at (tuple(RawPoint, int)).

Return type
tuple

Examples

Get the latest point for a stream using `version 0`.

```
>>> stream.latest(version=0)
(<btrdb.point.RawPoint at 0x...>, 1234567)
```

Extract just the `RawPoint` data.

```
>>> pt, _ = stream.latest(version=0)
>>> print(pt.time, pt.value)
1547241923338098176 123.7
```

property name

Returns the stream's name which is parsed from the stream tags. This may require a round trip to the server depending on how the stream was acquired.

Returns
The name of the stream.

Return type
str

Examples

```
>>> import btrdb
>>> conn = btrdb.connect()
>>> stream = conn.stream_from_uuid("...")
>>> stream.name
'foo'
```

nearest(*time*, *version*, *backward=False*, *auto_retry=False*, *retries=5*, *retry_delay=3*, *retry_backoff=4*)

Finds the closest point in the stream to a specified time.

Return the point nearest to the specified `time` in nanoseconds since Epoch in the stream with `version` while specifying whether to search forward or backward in time. If `backward` is `false`, the returned point will be `>= time`. If `backward` is `true`, the returned point will be `< time`. The `version` of the `stream` used to satisfy the query is returned.

Parameters

- **time** (*int or datetime like object*) – The time (in nanoseconds since Epoch) to search near (see :func:`btrdb.utils.timez.to_nanoseconds` for valid input types)
- **version** (*int*) – Version of the stream to use in search

- **backward** (*boolean*) – True to search backwards from time, else false for forward
- **auto_retry** (*bool, default: False*) – Whether to retry this request in the event of an error
- **retries** (*int, default: 5*) – Number of times to retry this request if there is an error. Will be ignored if auto_retry is False
- **retry_delay** (*int, default: 3*) – initial time to wait before retrying function call if there is an error. Will be ignored if auto_retry is False
- **retry_backoff** (*int, default: 4*) – Exponential factor by which the backoff increases between retries. Will be ignored if auto_retry is False

Returns
The closest data point in the stream and the version of the stream the value was retrieved at (tuple(RawPoint, int)).

Return type
tuple

obliterate(*auto_retry=False*, *retries=5*, *retry_delay=3*, *retry_backoff=4*)

Obliterate deletes a stream from the BTrDB server. An exception will be raised if the stream could not be found.

Parameters

- **auto_retry** (*bool, default: False*) – Whether to retry this request in the event of an error
- **retries** (*int, default: 5*) – Number of times to retry this request if there is an error. Will be ignored if auto_retry is False
- **retry_delay** (*int, default: 3*) – initial time to wait before retrying function call if there is an error. Will be ignored if auto_retry is False
- **retry_backoff** (*int, default: 4*) – Exponential factor by which the backoff increases between retries. Will be ignored if auto_retry is False

Raises
BTrDBError **[404] stream does not exist** – The stream could not be found in BTrDB

refresh_metadata()

Refreshes the locally cached metadata for a stream from the server.

Queries the BTrDB server for all stream metadata including collection, annotation, and tags. This method requires a round trip to the server.

tags(*refresh=False*, *auto_retry=False*, *retries=5*, *retry_delay=3*, *retry_backoff=4*)

Returns the stream's tags.

Tags returns the tags of the stream. It may require a round trip to the server depending on how the stream was acquired.

Parameters

- **refresh** (*bool, default: False*) – Indicates whether a round trip to the server should be implemented regardless of whether there is a local copy.
- **auto_retry** (*bool, default: False*) – Whether to retry this request in the event of an error

- **retries** (*int, default: 5*) – Number of times to retry this request if there is an error. Will be ignored if auto_retry is False
- **retry_delay** (*int, default: 3*) – initial time to wait before retrying function call if there is an error. Will be ignored if auto_retry is False
- **retry_backoff** (*int, default: 4*) – Exponential factor by which the backoff increases between retries. Will be ignored if auto_retry is False

Returns
A dictionary containing the tags.

Return type
dict

property unit

Returns the stream's unit which is parsed from the stream tags. This may require a round trip to the server depending on how the stream was acquired.

Returns
The unit for values of the stream.

Return type
str

Examples

```
>>> import btrdb
>>> conn = btrdb.connect()
>>> stream = conn.stream_from_uuid("...")
>>> stream.unit
'volts'
```

update(*tags=None*, *annotations=None*, *collection=None*, *encoder=<class 'btrdb.utils.conversion.AnnotationEncoder'>*, *replace=False*, *auto_retry=False*, *retries=5*, *retry_delay=3*, *retry_backoff=4*)

Updates metadata including tags, annotations, and collection as an UPSERT operation.

By default, the update will only affect the keys and values in the specified tags and annotations dictionaries, inserting them if they don't exist, or updating the value for the key if it does exist. If any of the update arguments (i.e. tags, annotations, collection) are None, they will remain unchanged in the database.

To delete either tags or annotations, you must specify exactly which keys and values you want set for the field and set `replace=True`.

This ensures that all the keys and values for the annotations are preserved except for the key to be deleted.

Parameters

- **tags** (*dict, optional*) – Specify the tag key/value pairs as a dictionary to upsert or replace. If None, the tags will remain unchanged in the database.
- **annotations** (*dict, optional*) – Specify the annotations key/value pairs as a dictionary to upsert or replace. If None, the annotations will remain unchanged in the database.
- **collection** (*str, optional*) – Specify a new collection for the stream. If None, the collection will remain unchanged.

- **encoder** (*json.JSONEncoder or None*) – JSON encoder class to use for annotation serialization. Set to `None` to prevent JSON encoding of the annotations.
- **replace** (*bool, default: False*) – Replace all annotations or tags with the specified dictionaries instead of performing the normal upsert operation. Specifying `True` is the only way to remove annotation keys.
- **auto_retry** (*bool, default: False*) – Whether to retry this request in the event of an error
- **retries** (*int, default: 5*) – Number of times to retry this request if there is an error. Will be ignored if auto_retry is False
- **retry_delay** (*int, default: 3*) – initial time to wait before retrying function call if there is an error. Will be ignored if auto_retry is False
- **retry_backoff** (*int, default: 4*) – Exponential factor by which the backoff increases between retries. Will be ignored if auto_retry is False

Returns
The version of the metadata (separate from the version of the data) also known as the "property version".

Return type
int

Examples

```
>>> annotations, _ = stream.anotations()
>>> del annotations["key_to_delete"]
>>> stream.update(annotations=annotations, replace=True)
12345
>>> annotations, _ = stream.annotations()
>>> "key_to_delete" in annotations
False
```

property uuid

Returns the stream's UUID. The stream may or may not exist yet, depending on how the stream object was obtained.

Returns
The unique identifier of the stream.

Return type
UUID

See also:

`stream.exists`

values(*start*, *end*, *version=0*, *auto_retry=False*, *retries=5*, *retry_delay=3*, *retry_backoff=4*)

Read raw values from BTrDB between time [a, b) in nanoseconds.

RawValues queries BTrDB for the raw time series data points between *start* and *end* time, both in nanoseconds since the Epoch for the specified stream *version*.

Parameters

- **start** (*int or datetime like object*) – The start time in nanoseconds for the range to be queried. (see *btrdb.utils.timez.to_nanoseconds()* for valid input types)
- **end** (*int or datetime like object*) – The end time in nanoseconds for the range to be queried. (see *btrdb.utils.timez.to_nanoseconds()* for valid input types)
- **version** (*int*) – The version of the stream to be queried
- **auto_retry** (*bool, default: False*) – Whether to retry this request in the event of an error
- **retries** (*int, default: 5*) – Number of times to retry this request if there is an error. Will be ignored if auto_retry is False
- **retry_delay** (*int, default: 3*) – initial time to wait before retrying function call if there is an error. Will be ignored if auto_retry is False
- **retry_backoff** (*int, default: 4*) – Exponential factor by which the backoff increases between retries. Will be ignored if auto_retry is False

Returns

Returns a list of tuples containing a RawPoint and the stream version (list(tuple(RawPoint,int))).

Return type

list

Note: Note that the raw data points are the original values at the sensor's native sampling rate (assuming the time series represents measurements from a sensor). This is the lowest level of data with the finest time granularity. In the tree data structure of BTrDB, this data is stored in the vector nodes.

---

version(*auto_retry=False*, *retries=5*, *retry_delay=3*, *retry_backoff=4*)

Returns the current data version of the stream.

> **Warning:** Version returns the current data version of the stream. This is not cached, it queries each time. Take care that you do not introduce races in your code by assuming this function will always return the same value.

Parameters

- **auto_retry** (*bool, default: False*) – Whether to retry this request in the event of an error
- **retries** (*int, default: 5*) – Number of times to retry this request if there is an error. Will be ignored if auto_retry is False
- **retry_delay** (*int, default: 3*) – initial time to wait before retrying function call if there is an error. Will be ignored if auto_retry is False
- **retry_backoff** (*int, default: 4*) – Exponential factor by which the backoff increases between retries. Will be ignored if auto_retry is False

Returns

The version of the stream.

Return type

int

windows(*start*, *end*, *width*, *depth=0*, *version=0*, *auto_retry=False*, *retries=5*, *retry_delay=3*, *retry_backoff=4*)

Read arbitrarily-sized windows of data from BTrDB. StatPoint objects will be returned representing the data for each window.

Parameters

- **start** (*int or datetime like object*) – The start time in nanoseconds for the range to be queried. (see *btrdb.utils.timez.to_nanoseconds()* for valid input types)
- **end** (*int or datetime like object*) – The end time in nanoseconds for the range to be queried. (see *btrdb.utils.timez.to_nanoseconds()* for valid input types)
- **width** (*int*) – The number of nanoseconds in each window.
- **version** (*int*) – The version of the stream to query.
- **auto_retry** (*bool, default: False*) – Whether to retry this request in the event of an error
- **retries** (*int, default: 5*) – Number of times to retry this request if there is an error. Will be ignored if auto_retry is False
- **retry_delay** (*int, default: 3*) – initial time to wait before retrying function call if there is an error. Will be ignored if auto_retry is False
- **retry_backoff** (*int, default: 4*) – Exponential factor by which the backoff increases between retries. Will be ignored if auto_retry is False

Returns

Returns a tuple containing windows of data. Each window is a tuple containing data tuples. Each data tuple contains a StatPoint and the stream version (tuple(tuple(StatPoint, int), ...)).

Return type

tuple

Note: `windows` returns arbitrary precision windows from BTrDB. It is slower than `aligned_windows`, but can be significantly faster than raw value queries (`values`). Each returned window will be `width` nanoseconds long. `start` is inclusive, but `end` is exclusive (e.g if `end < start+width` you will get no results). That is, results will be returned for all windows that start at a time less than the end timestamp. If (`end - start`) is not a multiple of `width`, then `end` will be decreased to the greatest value less than `end` such that (`end - start`) is a multiple of `width` (i.e., we set `end = start + width * floordiv(end - start, width)`. The `depth` parameter previously available has been deprecated. The only valid value for `depth` is now `0`.

---

class btrdb.stream.**StreamSet**(*streams: List[*Stream*]*)

Bases: `StreamSetBase`, `StreamSetTransformer`

Public class for a collection of streams

Attributes

allow_window

Methods

aligned_windows(pointwidth)	Stores the request for an aligned windowing operation when the query is eventually materialized.
arrow_insert(data_map[, merge])	Insert new data in the form (time, value) into their mapped streams using pyarrow tables.
arrow_to_arrow_table()	Return a pyarrow table of data.
arrow_to_dataframe([agg, name_callable])	Returns a Pandas DataFrame object indexed by time and using the values of a stream for each column.
arrow_to_dict([agg, name_callable])	Returns a list of dicts for each time code with the appropriate stream data attached.
arrow_to_numpy([agg])	Return a multidimensional array in the numpy format.
arrow_to_polars([agg, name_callable])	Returns a Polars DataFrame object with time as a column and the values of a stream for each additional column from an arrow table.
arrow_to_series([agg, name_callable])	Returns a list of Pandas Series objects indexed by time
arrow_values()	Return a pyarrow table of stream values based on the streamset parameters.
clone()	Returns a deep copy of the object.
count([precise])	Compute the total number of points in the streams using filters.
current()	Returns the points of data in the streams closest to the current timestamp.
earliest()	Returns earliest points of data in streams using available filters.
filter([start, end, collection, name, unit, ...])	Provides a new StreamSet instance containing stored query parameters and stream objects that match filtering criteria.
index(value, [start, [stop]])	Raises ValueError if the value is not present.
insert(data_map[, merge])	Insert new data in the form (time, value) into their mapped streams.
latest()	Returns latest points of data in the streams using available filters.
pin_versions([versions])	Saves the stream versions that future materializations should use.
rows()	Returns a materialized list of tuples where each tuple contains the points from each stream at a unique time.
to_array([agg])	Returns a multidimensional numpy array (similar to a list of lists) containing point classes.
to_csv(fobj[, dialect, fieldnames, agg, ...])	Saves stream data as a CSV file.
to_dataframe([agg, name_callable])	Returns a Pandas DataFrame object indexed by time and using the values of a stream for each column.
to_dict([agg, name_callable])	Returns a list of OrderedDict for each time code with the appropriate stream data attached.
to_polars([agg, name_callable])	Returns a Polars DataFrame object with time as a column and the values of a stream for each additional column.
to_series([datetime64_index, agg, name_callable])	Returns a list of Pandas Series objects indexed by time
to_table([agg, name_callable])	Returns string representation of the data in tabular form using the tabulate library.
values()	Returns a fully materialized list of lists for the stream values/points
~~*values_iter*()~~	~~Must return context object which would then close~~ server cursor on __exit__
versions()	Returns a dict of the stream versions.
windows(width[, depth])	Stores the request for a windowing operation when the query is eventually materialized.

aligned_windows(*pointwidth*)

Stores the request for an aligned windowing operation when the query is eventually materialized.

Parameters
pointwidth (*int*) – The length of each returned window as computed by 2^pointwidth.

Returns
Returns self

Return type
StreamSet

Note: `aligned_windows` reads power-of-two aligned windows from BTrDB. It is faster than `windows()`. Each returned window will be 2^``pointwidth`` nanoseconds long, beginning at `start`. Note that start is inclusive, but end is exclusive. That is, results will be returned for all windows that start in the interval `[start, end)`. If `end < start``+2^``pointwidth` you will not get any results. If `start` and `end` are not powers of two, the bottom `pointwidth` bits will be cleared. Each window will contain statistical summaries of the window. Statistical points with `count` == 0 will be omitted.

Examples

```
>>> import btrdb
>>> conn = btrdb.connect()
>>> stream1 = conn.stream_from_uuid("...")
>>> stream2 = conn.stream_from_uuid("...")
>>> streamset = btrdb.stream.StreamSet([stream1, stream2])
>>> streamset.aligned_windows(pointwidth=30)
<StreamSet ...>
>>> streamset.windows(width=1000000000)
Traceback (most recent call last):
  ...
btrdb.exceptions.InvalidOperation: A window operation is already requested
```

arrow_insert(*data_map: dict*, *merge: str = 'never'*) → dict

Insert new data in the form (time, value) into their mapped streams using pyarrow tables.

The times in the arrow table need not be sorted by time. If the point counts are larger than appropriate, this function will automatically chunk the inserts. As a consequence, the insert is not necessarily atomic, but can be used with a very large array.

Parameters

- **data_map** (*dict[uuid, pyarrow.Table]*) – A dictionary keyed on stream uuids and mapped to pyarrow tables with a schema of time:Timestamp[ns, tz=UTC], value:float64. This schema will be validated and converted if necessary.
- **merge** (*str*) –

  A string describing the merge policy. Valid policies are:

  - 'never': the default, no points are merged
  - 'equal': points are deduplicated if the time and value are equal
  - 'retain': if two points have the same timestamp, the old one is kept
  - 'replace': if two points have the same timestamp, the new one is kept

Notes

BTrDB expects datetimes to be in UTC+0.

This method is available for commercial customers with arrow-enabled servers.

Returns
The versions of the stream after inserting new points.

Return type
dict[uuid, int]

arrow_to_arrow_table()

Return a pyarrow table of data.

Note: This method is available for commercial customers with arrow-enabled servers.

arrow_to_dataframe(*agg=None*, *name_callable=None*) → DataFrame

Returns a Pandas DataFrame object indexed by time and using the values of a stream for each column.

Parameters

- **agg** (*List[str], default: ["mean"]*) – Specify the StatPoint fields (e.g. aggregating function) to create the dataframe from. Must be one or more of "min", "mean", "max", "count", "stddev", or "all". This argument is ignored if not using StatPoints.
- **name_callable** (*lambda, default: lambda s: s.collection + "/" + s.name*) – Specify a callable that can be used to determine the series name given a Stream object.

Note: This method is available for commercial customers with arrow-enabled servers.

Examples

```
>>> conn = btrdb.connect()
>>> s1 = conn.stream_from_uuid('c9fd8735-5ec5-4141-9a51-d23e1b2dfa42')
>>> s2 = conn.stream_from_uuid('9173fa70-87ab-4ac8-ac08-4fd63b910cae'
>>> streamset = btrdb.stream.StreamSet([s1,s2])
>>> streamset.filter(start=1500000000000000000, end=1500000000900000001).arrow_
↪to_dataframe()
                                        new/stream/collection/foo       new/
↪stream/bar
   time
   2017-07-14 02:40:00+00:00                                  1.0             1.0
   2017-07-14 02:40:00.100000+00:00                           2.0             2.0
   2017-07-14 02:40:00.200000+00:00                           3.0             3.0
   2017-07-14 02:40:00.300000+00:00                           4.0             4.0
   2017-07-14 02:40:00.400000+00:00                           5.0             5.0
   2017-07-14 02:40:00.500000+00:00                           6.0             6.0
   2017-07-14 02:40:00.600000+00:00                           7.0             7.0
   2017-07-14 02:40:00.700000+00:00                           8.0             8.0
```

(continues on next page)

(continued from previous page)

```
    2017-07-14 02:40:00.800000+00:00                                  9.0                 9.0
    2017-07-14 02:40:00.900000+00:00                                 10.0                10.0
```

Use the stream uuids as their column names instead, using a lambda function.

```
>>> streamset.filter(start=1500000000000000000, end=1500000000900000001)
...           .arrow_to_dataframe(
...              name_callable=lambda s: str(s.uuid)
...           )
                                  c9fd8735-5ec5-4141-9a51-d23e1b2dfa42 ␣
↪9173fa70-87ab-4ac8-ac08-4fd63b910cae
    time
    2017-07-14 02:40:00+00:00                                         1.0    ␣
↪                                1.0
    2017-07-14 02:40:00.100000+00:00                                  2.0    ␣
↪                                2.0
    2017-07-14 02:40:00.200000+00:00                                  3.0    ␣
↪                                3.0
    2017-07-14 02:40:00.300000+00:00                                  4.0    ␣
↪                                4.0
    2017-07-14 02:40:00.400000+00:00                                  5.0    ␣
↪                                5.0
    2017-07-14 02:40:00.500000+00:00                                  6.0    ␣
↪                                6.0
    2017-07-14 02:40:00.600000+00:00                                  7.0    ␣
↪                                7.0
    2017-07-14 02:40:00.700000+00:00                                  8.0    ␣
↪                                8.0
    2017-07-14 02:40:00.800000+00:00                                  9.0    ␣
↪                                9.0
    2017-07-14 02:40:00.900000+00:00                                 10.0    ␣
↪                               10.0
```

A window query, with a window width of 0.4 seconds, and only showing the `mean` statpoint.

```
>>> streamset.filter(start=1500000000000000000, end=1500000000900000001)
...           .windows(width=int(0.4*10**9))
...           .arrow_to_dataframe(agg=["mean"])
                                          new/stream/collection/foo/mean ␣
↪new/stream/bar/mean
    time
    2017-07-14 02:40:00+00:00                                        2.5    ␣
↪       2.5
    2017-07-14 02:40:00.400000+00:00                                 6.5    ␣
↪       6.5
```

A window query, with a window width of 0.4 seconds, and only showing the `mean` and `count` statpoints.

```
>>> streamset.filter(start=1500000000000000000, end=1500000000900000001)
...           .windows(width=int(0.4*10**9))
...           .arrow_to_dataframe(agg=["mean", "count"])
                             new/stream/collection/foo/mean  new/stream/
↪collection/foo/count  new/stream/bar/mean  new/stream/bar/count
```

(continues on next page)

(continued from previous page)

```
    time
    2017-07-14 02:40:00+00:00               2.5                                 ␣
↪4                  2.5                      4
    2017-07-14 02:40:00.400000+00:00        6.5                                 ␣
↪4                  6.5                      4
```

arrow_to_dict(*agg=None*, *name_callable=None*)

Returns a list of dicts for each time code with the appropriate stream data attached.

Parameters

- **agg** (*List[str], default: ["mean"]*) – Specify the StatPoint field or fields (e.g. aggregating function) to constrain dict keys. Must be one or more of "min", "mean", "max", "count", or "stddev". This argument is ignored if RawPoint values are passed into the function.
- **name_callable** (*lambda, default: lambda s: s.collection + "/" + s.name*) – Specify a callable that can be used to determine the series name given a Stream object.

Note: This method is available for commercial customers with arrow-enabled servers.

arrow_to_numpy(*agg=None*)

Return a multidimensional array in the numpy format.

Parameters

agg (*List[str], default: ["mean"]*) – Specify the StatPoint field or fields (e.g. aggregating function) to return for the arrays. Must be one or more of "min", "mean", "max", "count", or "stddev". This argument is ignored if RawPoint values are passed into the function.

Note: This method first converts to a pandas data frame then to a numpy array.

Note: This method is available for commercial customers with arrow-enabled servers.

arrow_to_polars(*agg=None*, *name_callable=None*)

Returns a Polars DataFrame object with time as a column and the values of a stream for each additional column from an arrow table.

Parameters

- **agg** (*List[str], default: ["mean"]*) – Specify the StatPoint field or fields (e.g. aggregating function) to create the dataframe from. Must be one or multiple of "min", "mean", "max", "count", "stddev", or "all". This argument is ignored if not using StatPoints.
- **name_callable** (*lambda, default: lambda s: s.collection + "/" + s.name*) – Specify a callable that can be used to determine the series name given a Stream object.

Note: This method is available for commercial customers with arrow-enabled servers.

arrow_to_series(*agg=None*, *name_callable=None*)

Returns a list of Pandas Series objects indexed by time

Parameters

- **agg** (*List[str], default: ["mean"]*) – Specify the StatPoint field or fields (e.g. aggregating function) to create the Series from. Must be one or more of "min", "mean", "max", "count", or "stddev". This argument is ignored if RawPoint values are passed into the function.
- **name_callable** (*lambda, default: lambda s: s.collection + "/" + s.name*) – Specify a callable that can be used to determine the series name given a Stream object.

Return type

List[pandas.Series]

Note: This method is available for commercial customers with arrow-enabled servers.

Note: If you are not performing a `window` or `aligned_window` query, the `agg` parameter will be ignored.

Examples

Return a list of series of raw data per stream.

```
>>> conn = btrdb.connect()
>>> s1 = conn.stream_from_uuid('c9fd8735-5ec5-4141-9a51-d23e1b2dfa42')
>>> s2 = conn.stream_from_uuid('9173fa70-87ab-4ac8-ac08-4fd63b910cae'
>>> streamset = btrdb.stream.StreamSet([s1,s2])
>>> streamset.filter(start=1500000000000000000, end=1500000000900000001).arrow_
↪to_series(agg=None)
    [time
    2017-07-14 02:40:00+00:00            1.0
    2017-07-14 02:40:00.100000+00:00     2.0
    2017-07-14 02:40:00.200000+00:00     3.0
    2017-07-14 02:40:00.300000+00:00     4.0
    2017-07-14 02:40:00.400000+00:00     5.0
    2017-07-14 02:40:00.500000+00:00     6.0
    2017-07-14 02:40:00.600000+00:00     7.0
    2017-07-14 02:40:00.700000+00:00     8.0
    2017-07-14 02:40:00.800000+00:00     9.0
    2017-07-14 02:40:00.900000+00:00    10.0
    Name: new/stream/collection/foo, dtype: double[pyarrow],
    time
    2017-07-14 02:40:00+00:00            1.0
    2017-07-14 02:40:00.100000+00:00     2.0
    2017-07-14 02:40:00.200000+00:00     3.0
    2017-07-14 02:40:00.300000+00:00     4.0
    2017-07-14 02:40:00.400000+00:00     5.0
    2017-07-14 02:40:00.500000+00:00     6.0
    2017-07-14 02:40:00.600000+00:00     7.0
```

(continues on next page)

(continued from previous page)

```
    2017-07-14 02:40:00.700000+00:00     8.0
    2017-07-14 02:40:00.800000+00:00     9.0
    2017-07-14 02:40:00.900000+00:00    10.0
    Name: new/stream/bar, dtype: double[pyarrow]]
```

A window query of 0.5seconds long.

```
>>> streamset.filter(start=1500000000000000000,␣
↪end=1500000000900000001)
...           .windows(width=int(0.5 * 10**9))
...           .arrow_to_series(agg=["mean", "count"])
    [time
    2017-07-14 02:40:00+00:00           2.5
    2017-07-14 02:40:00.400000+00:00    6.5
    Name: new/stream/collection/foo/mean, dtype: double[pyarrow],
...
    time
    2017-07-14 02:40:00+00:00           4
    2017-07-14 02:40:00.400000+00:00    4
    Name: new/stream/collection/foo/count, dtype: uint64[pyarrow],
...
    time
    2017-07-14 02:40:00+00:00           2.5
    2017-07-14 02:40:00.400000+00:00    6.5
    Name: new/stream/bar/mean, dtype: double[pyarrow],
...
    time
    2017-07-14 02:40:00+00:00           4
    2017-07-14 02:40:00.400000+00:00    4
    Name: new/stream/bar/count, dtype: uint64[pyarrow]]
```

arrow_values()

Return a pyarrow table of stream values based on the streamset parameters.

This data will be sorted by the 'time' column.

Note: This method is available for commercial customers with arrow-enabled servers.

clone()

Returns a deep copy of the object. Attributes that cannot be copied will be referenced to both objects.

Parameters
None

Returns
Returns a new copy of the instance

Return type
StreamSet

count(*precise: bool = False*)

Compute the total number of points in the streams using filters.

Computes the total number of points across all streams using the specified filters. By default, this returns the latest total count of all points in the streams. The count is modified by start and end filters or by pinning versions.

Parameters

precise (*bool, default = False*) – Use statpoint counts using aligned_windows which trades accuracy for speed.

Returns

- *int* – The total number of points in all streams for the specified filters.
- .. *note::* – Note that this helper method sums the counts of all StatPoints returned by `aligned_windows`. Because of this the start and end timestamps may be adjusted if they are not powers of 2.

Examples

```
>>> import btrdb
>>> conn = btrdb.connect()
>>> stream1 = conn.stream_from_uuid("...")
>>> stream2 = conn.stream_from_uuid("...")
>>> streamset = btrdb.stream.StreamSet([stream1, stream2])
>>> streamset.count()
2345
>>> filtered_streamset = streamset.filter(start=150000000000000000,␣
↪end=150000000010000000)
>>> filtered_streamset.count(precise=True)
734
>>> streamset.filter(start=150000000000000000, end=150000000010000000).
↪count(precise=True)
734
```

current()

Returns the points of data in the streams closest to the current timestamp. If the current timestamp is outside the filtered range of data, a ValueError is raised.

Returns

The latest points of data found among all streams

Return type

tuple

Examples

```
>>> import btrdb
>>> conn = btrdb.connect()
>>> stream1 = conn.stream_from_uuid("...")
>>> stream2 = conn.stream_from_uuid("...")
>>> streamset = btrdb.stream.StreamSet([stream1, stream2])
>>> streamset.current()
(<RawPoint ...>, <RawPoint ...>)
```

earliest()

Returns earliest points of data in streams using available filters.

Parameters
None

Returns
The earliest points of data found among all streams

Return type
tuple

Examples

```
>>> import btrdb
>>> conn = btrdb.connect()
>>> stream1 = conn.stream_from_uuid("...")
>>> stream2 = conn.stream_from_uuid("...")
>>> streamset = btrdb.stream.StreamSet([stream1, stream2])
>>> streamset.earliest()
(<RawPoint ...>, <RawPoint ...>)
```

filter(*start=None*, *end=None*, *collection=None*, *name=None*, *unit=None*, *tags=None*, *annotations=None*, *sampling_frequency=None*, *schema=None*)

Provides a new StreamSet instance containing stored query parameters and stream objects that match filtering criteria.

The collection, name, and unit arguments will be used to select streams from the original StreamSet object. If a string is supplied, then a case-insensitive exact match is used to select streams. Otherwise, you may supply a compiled regex pattern that will be used with *re.search*.

The tags and annotations arguments expect dictionaries for the desired key/value pairs. Any stream in the original instance that has the exact key/values will be included in the new StreamSet instance.

Parameters

- **start** (*int or datetime like object*) – the inclusive start of the query (see *btrdb.utils.timez.to_nanoseconds()* for valid input types)
- **end** (*int or datetime like object*) – the exclusive end of the query (see *btrdb.utils.timez.to_nanoseconds()* for valid input types)
- **collection** (*str or regex*) – string for exact (case-insensitive) matching of collection when filtering streams or a compiled regex expression for re.search of stream collections.
- **name** (*str or regex*) – string for exact (case-insensitive) matching of name when filtering streams or a compiled regex expression for re.search of stream names.
- **unit** (*str or regex*) – string for exact (case-insensitive) matching of unit when filtering streams or a compiled regex expression for re.search of stream units.
- **tags** (*dict*) – key/value pairs for filtering streams based on tags
- **annotations** (*dict*) – key/value pairs for filtering streams based on annotations
- **sampling_frequency** (*float*) – The sampling frequency of the data streams in Hz, set this if you want timesnapped values.

- **schema** (*pyarrow.Schema*) – Optional arrow schema the server will cast the returned data to before sending it over the network. You can use this to change the timestamp format, column names or data sizes.

Returns
a new instance cloned from the original with filters applied

Return type
StreamSet

Note: If you set `sampling_frequency` to a non-zero value, the stream data returned will be aligned to a grid of timestamps based on the period of the sampling frequency. For example, a sampling rate of 30hz will have a sampling period of 1/30hz -> ~33_333_333 ns per sample. Leave `sampling_frequency` as `None`, or set to `0` to prevent time alignment. You should **not** use aligned data for frequency-based analysis.

Examples

create a streamset and apply a few filters

```
>>> streamset = btrdb.stream.StreamSet(list_of_streams)
>>> print(f"Total streams: {len(streamset)}")
Total streams: 89
```

```
>>> streamset.filter(units="Volts")
>>> print(f"Total streams: {len(streamset)}")
Total streams: 89
```

```
>>> filtered_streamset = streamset.filter(units="Volts")
>>> print(f"Total streams: {len(filtered_streamset)}")
Total streams: 23
```

```
>>> multiple_filters_streamset = (streamset.filter(unit="Volts")
>>>                                .filter(name="Sensor 1")
>>>                                .filter(annotations={"phase":"A"})
>>>                              )
>>> print(f"Total streams: {len(multiple_filters_streamset)}")
Total streams: 1
```

index(*value*[, *start*[, *stop*]]) → integer -- return first index of value.

Raises ValueError if the value is not present.

Supporting start and stop arguments is optional, but recommended.

insert(*data_map: dict*, *merge: str = 'never'*) → dict

Insert new data in the form (time, value) into their mapped streams.

The times in the dataframe need not be sorted by time. If the point counts are larger than appropriate, this function will automatically chunk the inserts. As a consequence, the insert is not necessarily atomic, but can be used with a very large array.

Parameters

- **data_map** (*dict[uuid, pandas.DataFrame]*) – A dictionary mapping stream uuids to insert data into and their value as a pandas dataframe containing two columns,

one named "time" which contains int64 utc+0 timestamps and a "value" column containing float64 measurements. These columns will be typecast into these types.

- **merge** (*str*) –

  A string describing the merge policy. Valid policies are:

  - 'never': the default, no points are merged
  - 'equal': points are deduplicated if the time and value are equal
  - 'retain': if two points have the same timestamp, the old one is kept
  - 'replace': if two points have the same timestamp, the new one is kept

Returns
The versions of the stream after inserting new points.

Return type
dict[uuid, int]

Note: You MUST convert your datetimes into UTC+0 **yourself**. BTrDB expects UTC+0 datetimes.

Examples

```
>>> import pandas as pd
>>> import btrdb
>>> conn = btrdb.connect()
>>> stream1 = conn.stream_from_uuid("...")
>>> stream2 = conn.stream_from_uuid("...")
>>> streamset = btrdb.stream.StreamSet([stream1, stream2])
>>> data_map = {
...     stream1.uuid: pd.DataFrame({'time': [1500000000000000000,␣
↪1500000000100000000], 'value': [1.0, 2.0]}),
...     stream2.uuid: pd.DataFrame({'time': [1500000000000000000,␣
↪1500000000100000000], 'value': [3.0, 4.0]})
... }
>>> streamset.insert(data_map)
{UUID('...'): 1234, UUID('...'): 5678}
>>> streamset.insert(data_map, merge='replace')
{UUID('...'): 1235, UUID('...'): 5679}
```

`latest()`

Returns latest points of data in the streams using available filters.

Parameters
None

Returns
The latest points of data found among all streams

Return type
tuple

Examples

```
>>> import btrdb
>>> conn = btrdb.connect()
>>> stream1 = conn.stream_from_uuid("...")
>>> stream2 = conn.stream_from_uuid("...")
>>> streamset = btrdb.stream.StreamSet([stream1, stream2])
>>> streamset.earliest()
(<RawPoint ...>, <RawPoint ...>)
```

pin_versions(*versions=None*)

Saves the stream versions that future materializations should use. If no pin is requested then the first materialization will automatically pin the return versions. Versions can also be supplied through a `dict` object with key:UUID, value:stream.version().

Parameters

versions (*dict[UUID: int]*) – A dict containing the stream UUID and version ints as key/values

Returns

Returns self

Return type

StreamSet

Examples

```
>>> version_map = {s.uuid: 0 for s in streamset}
>>> pinned_streamset = streamset.pin_versions(versions=version_map)
```

rows()

Returns a materialized list of tuples where each tuple contains the points from each stream at a unique time. If a stream has no value for that time than None is provided instead of a point object.

Parameters

None

Returns

A list of tuples containing a RawPoint (or StatPoint) and the stream version (list(tuple(RawPoint, int))).

Return type

list

Examples

```
>>> for row in streams.rows():
>>>     print(row)
(None, RawPoint(1500000000000000000, 1.0), RawPoint(1500000000000000000, 1.0),␣
↪RawPoint(1500000000000000000, 1.0))
(RawPoint(1500000000100000000, 2.0), None, RawPoint(1500000000100000000, 2.0),␣
↪RawPoint(1500000000100000000, 2.0))
(None, RawPoint(1500000000200000000, 3.0), None, RawPoint(1500000000200000000,␣
```

(continues on next page)

(continued from previous page)

```
↪3.0))
(RawPoint(1500000000300000000, 4.0), None, RawPoint(1500000000300000000, 4.0),␣
↪RawPoint(1500000000300000000, 4.0))
(None, RawPoint(1500000000400000000, 5.0), RawPoint(1500000000400000000, 5.0),␣
↪RawPoint(1500000000400000000, 5.0))
(RawPoint(1500000000500000000, 6.0), None, None, RawPoint(1500000000500000000,␣
↪6.0))
(None, RawPoint(1500000000600000000, 7.0), RawPoint(1500000000600000000, 7.0),␣
↪RawPoint(1500000000600000000, 7.0))
(RawPoint(1500000000700000000, 8.0), None, RawPoint(1500000000700000000, 8.0),␣
↪RawPoint(1500000000700000000, 8.0))
(None, RawPoint(1500000000800000000, 9.0), RawPoint(1500000000800000000, 9.0),␣
↪RawPoint(1500000000800000000, 9.0))
(RawPoint(1500000000900000000, 10.0), None, RawPoint(1500000000900000000, 10.
↪0), RawPoint(1500000000900000000, 10.0))
```

to_array(*agg='mean'*)

Returns a multidimensional numpy array (similar to a list of lists) containing point classes.

Parameters

agg (*str, default: "mean"*) – Specify the StatPoint field (e.g. aggregating function) to return for the arrays. Must be one of "min", "mean", "max", "count", or "stddev". This argument is ignored if RawPoint values are passed into the function.

Note: This method does **not** use the `arrow` -accelerated endpoints for faster and more efficient data retrieval.

---

to_csv(*fobj*, *dialect=None*, *fieldnames=None*, *agg='mean'*, *name_callable=None*)

Saves stream data as a CSV file.

Parameters

- **fobj** (*str or file-like object*) – Path to use for saving CSV file or a file-like object to use to write to.
- **dialect** (*csv.Dialect*) – CSV dialect object from Python csv module. See Python's csv module for more information.
- **fieldnames** (*sequence*) – A sequence of strings to use as fieldnames in the CSV header. See Python's csv module for more information.
- **agg** (*str, default: "mean"*) – Specify the StatPoint field (e.g. aggregating function) to return when limiting results. Must be one of "min", "mean", "max", "count", or "stddev". This argument is ignored if RawPoint values are passed into the function.
- **name_callable** (*lambda, default: lambda s: s.collection + "/" + s.name*) – Specify a callable that can be used to determine the series name given a Stream object.

Note: This method does **not** use the `arrow` -accelerated endpoints for faster and more efficient data retrieval.

---

to_dataframe(*agg='mean'*, *name_callable=None*)

Returns a Pandas DataFrame object indexed by time and using the values of a stream for each column.

Parameters

- **agg** (*str, default: "mean"*) – Specify the StatPoint field (e.g. aggregating function) to create the Series from. Must be one of "min", "mean", "max", "count", "stddev", or "all". This argument is ignored if not using StatPoints.
- **name_callable** (*lambda, default: lambda s: s.collection + "/" + s.name*) – Specify a callable that can be used to determine the series name given a Stream object. This is not compatible with agg == "all" at this time

---

Note: This method does **not** use the `arrow` -accelerated endpoints for faster and more efficient data retrieval.

`to_dict`(*agg='mean'*, *name_callable=None*)

Returns a list of OrderedDict for each time code with the appropriate stream data attached.

Parameters

- **agg** (*str, default: "mean"*) – Specify the StatPoint field (e.g. aggregating function) to constrain dict keys. Must be one of "min", "mean", "max", "count", or "stddev". This argument is ignored if RawPoint values are passed into the function.
- **name_callable** (*lambda, default: lambda s: s.collection + "/" + s.name*) – Specify a callable that can be used to determine the series name given a Stream object.

Note: This method does **not** use the `arrow` -accelerated endpoints for faster and more efficient data retrieval.

`to_polars`(*agg='mean'*, *name_callable=None*)

Returns a Polars DataFrame object with time as a column and the values of a stream for each additional column.

Parameters

- **agg** (*str, default: "mean"*) – Specify the StatPoint field (e.g. aggregating function) to create the Series from. Must be one of "min", "mean", "max", "count", "stddev", or "all". This argument is ignored if not using StatPoints.
- **name_callable** (*lambda, default: lambda s: s.collection + "/" + s.name*) – Specify a callable that can be used to determine the series name given a Stream object. This is not compatible with agg == "all" at this time

---

Note: This method does **not** use the `arrow` -accelerated endpoints for faster and more efficient data retrieval.

`to_series`(*datetime64_index=True*, *agg='mean'*, *name_callable=None*)

Returns a list of Pandas Series objects indexed by time

Parameters

- **datetime64_index** (*bool*) – Directs function to convert Series index to np.datetime64[ns] or leave as np.int64.

- **agg** (*str, default: "mean"*) – Specify the StatPoint field (e.g. aggregating function) to create the Series from. Must be one of "min", "mean", "max", "count", or "stddev". This argument is ignored if RawPoint values are passed into the function.
- **name_callable** (*lambda, default: lambda s: s.collection + "/" + s.name*) – Specify a callable that can be used to determine the series name given a Stream object.

Note: This method does **not** use the `arrow` -accelerated endpoints for faster and more efficient data retrieval.

`to_table`(*agg='mean'*, *name_callable=None*)

Returns string representation of the data in tabular form using the tabulate library.

Parameters

- **agg** (*str, default: "mean"*) – Specify the StatPoint field (e.g. aggregating function) to create the Series from. Must be one of "min", "mean", "max", "count", or "stddev". This argument is ignored if RawPoint values are passed into the function.
- **name_callable** (*lambda, default: lambda s: s.collection + "/" + s.name*) – Specify a callable that can be used to determine the column name given a Stream object.

Note: This method does **not** use the `arrow` -accelerated endpoints for faster and more efficient data retrieval.

`values`()

Returns a fully materialized list of lists for the stream values/points

`versions`()

Returns a dict of the stream versions. These versions are the pinned values if previously pinned or the latest stream versions if not pinned.

Parameters
`None`

Returns
A dict containing the stream UUID and version ints as key/values

Return type
dict

Examples

A pinned vs non-pinned streamset

```
>>> streamset = btrdb.stream.StreamSet([stream1, stream2])
>>> version_map = {s.uuid: 0 for s streamset}
>>> pinned_streamset = streamset.pin_versions(versions=version_map)
>>> pinned_streamset.versions()
{UUID('fa42f64a-a851-408f-aa7e-88a85b3d295c'): 0, UUID('18e5527a-ed13-424d-
↪bb97-3e06a763609e'): 0}
>>> streamset.versions()
```

(continues on next page)

(continued from previous page)

```
{UUID('fa42f64a-a851-408f-aa7e-88a85b3d295c'): 34532, UUID('18e5527a-ed13-424d-
↪bb97-3e06a763609e'): 12345}
```

windows(*width*, *depth=0*)

Stores the request for a windowing operation when the query is eventually materialized.

Parameters

- **width** (*int*) – The number of nanoseconds to use for each window size.
- **depth** (*int*) – The requested accuracy of the data up to 2^depth nanoseconds. A depth of 0 is accurate to the nanosecond. This is now the only valid value for depth.

Returns

Returns self

Return type

StreamSet

Note: `windows` returns arbitrary precision windows from BTrDB. It is slower than `aligned_windows`, but can be significantly faster than values. Each returned window will be `width` nanoseconds long. `start` is inclusive, but `end` is exclusive (`[start, end)`) `(e.g. if end < start+width you will get no results). That is, results will be returned for all windows that start at a time less than the end timestamp. If (``end - start)` is not a multiple of `width`, then `end` will be decreased to the greatest value less than `end` such that (end - start) is a multiple of width (i.e., we set end = start + width * floordiv(end - start, width)). The `depth` parameter previously available has been deprecated. The only valid value for `depth` is now `0`.

Examples

```
>>> import btrdb
>>> conn = btrdb.connect()
>>> stream1 = conn.stream_from_uuid("...")
>>> stream2 = conn.stream_from_uuid("...")
>>> streamset = btrdb.stream.StreamSet([stream1, stream2])
>>> streamset.windows(width=1000000000)
<StreamSet ...>
>>> streamset.windows(width=1000000000, depth=0)
<StreamSet ...>
>>> streamset.aligned_windows(pointwidth=30)
Traceback (most recent call last):
  ...
btrdb.exceptions.InvalidOperation: A window operation is already requested
```

1.6.4 btrdb.point

Module for Point classes

class btrdb.point.**RawPoint**(*time*, *value*)

A point of data representing a single position within a time series. Each point contains a read-only time and value attribute.

Parameters

- **time** (*int*) – The time portion of a single value in the time series in nanoseconds since the Unix epoch.
- **value** (*float*) – The value of a time series at a single point in time.

Attributes

time
The time portion of a data point in nanoseconds since the Unix epoch.

value
The value portion of a data point as a float object.

Methods

from_proto
from_proto_list
to_proto
to_proto_list

property time

The time portion of a data point in nanoseconds since the Unix epoch.

property value

The value portion of a data point as a float object.

class btrdb.point.**StatPoint**(*time*, *minv*, *meanv*, *maxv*, *count*, *stddev*)

An aggregated data point representing a summary or rollup of one or more points of data within a single time series.

This aggregation point provides for the min, mean, max, count, and standard deviation of all data values it spans. It is returned by windowing queries such as *windows* or *aligned_windows*.

Parameters

- **time** (*int*) – The start time of the window which spans the aggregated values. Represented in nanoseconds since the Unix epoch.
- **min** (*float*) – The minimum value in a time series within a specified range of time.
- **mean** (*float*) – The mean value in a time series within a specified range of time.
- **max** (*float*) – The maximum value in a time series within a specified range of time.
- **count** (*float*) – The number of values in a time series within a specified range of time.
- **stddev** (*float*) – The standard deviation of values in a time series within a specified range of time.

Notes

This object may also be treated as a tuple by referencing the values according to position.

```
// returns time
val = point[0]

// returns standard deviation
val = point[5]
```

Attributes

count
The number of values within the time series for a range of time

max
The maximum value of the time series within a range of time

mean
The mean value of the time series within a range of time

min
The minimum value of the time series within a range of time

stddev
The standard deviation of the values of a time series within a range of time

time
The starting time of the time series within the stat point

Methods

from_proto
from_proto_list

property count
The number of values within the time series for a range of time

property max
The maximum value of the time series within a range of time

property mean
The mean value of the time series within a range of time

property min
The minimum value of the time series within a range of time

property stddev
The standard deviation of the values of a time series within a range of time

property time
The starting time of the time series within the stat point

1.6.5 btrdb.exceptions

exception btrdb.exceptions.**BTrDBError**

The primary exception for BTrDB errors.

exception btrdb.exceptions.**ConnectionError**

Raised when an error occurrs while trying to establish a connection with BTrDB.

exception btrdb.exceptions.**StreamNotFoundError**

Raised when attempting to perform an operation on a stream that does not exist in the specified BTrDB allocation.

exception btrdb.exceptions.**CredentialsFileNotFound**

Raised when a credentials file could not be found.

exception btrdb.exceptions.**ProfileNotFound**

Raised when a requested profile could not be found in the credentials file.

exception btrdb.exceptions.**BTRDBServerError**

Raised when an error occurs with btrdb-server.

exception btrdb.exceptions.**BTRDBTypeError**

Raised when attempting to perform an operation with an invalid type.

exception btrdb.exceptions.**InvalidOperation**

Raised when an invalid BTrDB operation has been requested.

exception btrdb.exceptions.**StreamExists**

Raised when create() has been attempted and the uuid already exists.

exception btrdb.exceptions.**AmbiguousStream**

Raised when create() has been attempted and uuid is different, but collection and tags already exist

exception btrdb.exceptions.**PermissionDenied**

Raised when user does not have permission to perform an operation.

exception btrdb.exceptions.**BTRDBValueError**

Raised when an invalid value has been passed to a BTrDB operation.

exception btrdb.exceptions.**InvalidCollection**

Raised when a collection name is invalid. It is either too long or not a valid string.

exception btrdb.exceptions.**InvalidTagKey**

Raised when a tag key is invalid. Must be one of (“name”, “unit”, “ingress”, “distiller”).

exception btrdb.exceptions.**InvalidTagValue**

Raised when a tag value is invalid. It is either too long or not a valid string.

exception btrdb.exceptions.**InvalidTimeRange**

Raised when insert data contains a timestamp outside the range of (btrdb.MINIMUM_TIME, btrdb.MAXIMUM_TIME)

exception btrdb.exceptions.**InvalidPointWidth**

Raised when attempting to use a pointwidth that is not a whole number between 0 and 64 (exclusive).

exception btrdb.exceptions.**BadValue**

Raised when attempting to insert data that contains non-float values such as None.

exception btrdb.exceptions.**RecycledUUID**

Raised when attempting to create a stream with a uuid that matches a previously deleted stream.

exception btrdb.exceptions.**BadSQLValue**

Raised when invalid parameters have been passed to metadata db.

exception btrdb.exceptions.**VersionNotAvailable**

Raised when querying a stream at a pruned or otherwise invalid version number.

exception btrdb.exceptions.**NoSuchPoint**

Raised when asking for next/previous point and there isn't one.

1.6.6 btrdb.transformers

A number of transformation and serialization functions have been developed so you can use the data in the format of your choice. These functions are provided in the `StreamSet` class. Value transformation utilities

btrdb.transformers.**arrow_to_dict**(*streamset*, *agg=None*, *name_callable=None*)

Returns a list of dicts for each time code with the appropriate stream data attached.

Parameters

- **agg** (*List[str], default: ["mean"]*) – Specify the StatPoint field or fields (e.g. aggregating function) to constrain dict keys. Must be one or more of "min", "mean", "max", "count", or "stddev". This argument is ignored if RawPoint values are passed into the function.
- **name_callable** (*lambda, default: lambda s: s.collection + "/" + s.name*) – Specify a callable that can be used to determine the series name given a Stream object.

Note: This method is available for commercial customers with arrow-enabled servers.

---

btrdb.transformers.**arrow_to_numpy**(*streamset*, *agg=None*)

Return a multidimensional array in the numpy format.

Parameters

agg (*List[str], default: ["mean"]*) – Specify the StatPoint field or fields (e.g. aggregating function) to return for the arrays. Must be one or more of "min", "mean", "max", "count", or "stddev". This argument is ignored if RawPoint values are passed into the function.

Note: This method first converts to a pandas data frame then to a numpy array.

Note: This method is available for commercial customers with arrow-enabled servers.

---

btrdb.transformers.**arrow_to_series**(*streamset*, *agg=None*, *name_callable=None*)

Returns a list of Pandas Series objects indexed by time

Parameters

- **agg** (*List[str], default: ["mean"]*) – Specify the StatPoint field or fields (e.g. aggregating function) to create the Series from. Must be one or more of "min", "mean",

"max", "count", or "stddev". This argument is ignored if RawPoint values are passed into the function.

- **name_callable** (*lambda, default: lambda s: s.collection + "/" + s.name*) – Specify a callable that can be used to determine the series name given a Stream object.

Return type
List[pandas.Series]

Note: This method is available for commercial customers with arrow-enabled servers.

Note: If you are not performing a `window` or `aligned_window` query, the `agg` parameter will be ignored.

Examples

Return a list of series of raw data per stream.

```
>>> conn = btrdb.connect()
>>> s1 = conn.stream_from_uuid('c9fd8735-5ec5-4141-9a51-d23e1b2dfa42')
>>> s2 = conn.stream_from_uuid('9173fa70-87ab-4ac8-ac08-4fd63b910cae'
>>> streamset = btrdb.stream.StreamSet([s1,s2])
>>> streamset.filter(start=1500000000000000000, end=1500000000900000001).arrow_to_
↪series(agg=None)
    [time
    2017-07-14 02:40:00+00:00            1.0
    2017-07-14 02:40:00.100000+00:00     2.0
    2017-07-14 02:40:00.200000+00:00     3.0
    2017-07-14 02:40:00.300000+00:00     4.0
    2017-07-14 02:40:00.400000+00:00     5.0
    2017-07-14 02:40:00.500000+00:00     6.0
    2017-07-14 02:40:00.600000+00:00     7.0
    2017-07-14 02:40:00.700000+00:00     8.0
    2017-07-14 02:40:00.800000+00:00     9.0
    2017-07-14 02:40:00.900000+00:00    10.0
    Name: new/stream/collection/foo, dtype: double[pyarrow],
    time
    2017-07-14 02:40:00+00:00            1.0
    2017-07-14 02:40:00.100000+00:00     2.0
    2017-07-14 02:40:00.200000+00:00     3.0
    2017-07-14 02:40:00.300000+00:00     4.0
    2017-07-14 02:40:00.400000+00:00     5.0
    2017-07-14 02:40:00.500000+00:00     6.0
    2017-07-14 02:40:00.600000+00:00     7.0
    2017-07-14 02:40:00.700000+00:00     8.0
    2017-07-14 02:40:00.800000+00:00     9.0
    2017-07-14 02:40:00.900000+00:00    10.0
    Name: new/stream/bar, dtype: double[pyarrow]]
```

A window query of 0.5seconds long.

```
>>> streamset.filter(start=1500000000000000000, end=1500000000900000001)
...          .windows(width=int(0.5 * 10**9))
...          .arrow_to_series(agg=["mean", "count"])
    [time
    2017-07-14 02:40:00+00:00           2.5
    2017-07-14 02:40:00.400000+00:00    6.5
    Name: new/stream/collection/foo/mean, dtype: double[pyarrow],
...
    time
    2017-07-14 02:40:00+00:00           4
    2017-07-14 02:40:00.400000+00:00    4
    Name: new/stream/collection/foo/count, dtype: uint64[pyarrow],
...
    time
    2017-07-14 02:40:00+00:00           2.5
    2017-07-14 02:40:00.400000+00:00    6.5
    Name: new/stream/bar/mean, dtype: double[pyarrow],
...
    time
    2017-07-14 02:40:00+00:00           4
    2017-07-14 02:40:00.400000+00:00    4
    Name: new/stream/bar/count, dtype: uint64[pyarrow]]
```

btrdb.transformers.**arrow_to_dataframe**(*streamset*, *agg=None*, *name_callable=None*) → DataFrame

Returns a Pandas DataFrame object indexed by time and using the values of a stream for each column.

Parameters

- **agg** (*List[str], default: ["mean"]*) – Specify the StatPoint fields (e.g. aggregating function) to create the dataframe from. Must be one or more of "min", "mean", "max", "count", "stddev", or "all". This argument is ignored if not using StatPoints.
- **name_callable** (*lambda, default: lambda s: s.collection + "/" + s.name*) – Specify a callable that can be used to determine the series name given a Stream object.

Note: This method is available for commercial customers with arrow-enabled servers.

Examples

```
>>> conn = btrdb.connect()
>>> s1 = conn.stream_from_uuid('c9fd8735-5ec5-4141-9a51-d23e1b2dfa42')
>>> s2 = conn.stream_from_uuid('9173fa70-87ab-4ac8-ac08-4fd63b910cae'
>>> streamset = btrdb.stream.StreamSet([s1,s2])
>>> streamset.filter(start=1500000000000000000, end=1500000000900000001).arrow_to_
↪dataframe()
                                    new/stream/collection/foo       new/stream/
↪bar
    time
    2017-07-14 02:40:00+00:00                                1.0             1.0
    2017-07-14 02:40:00.100000+00:00                         2.0             2.0
```

(continues on next page)

(continued from previous page)

```
    2017-07-14 02:40:00.200000+00:00                    3.0             3.0
    2017-07-14 02:40:00.300000+00:00                    4.0             4.0
    2017-07-14 02:40:00.400000+00:00                    5.0             5.0
    2017-07-14 02:40:00.500000+00:00                    6.0             6.0
    2017-07-14 02:40:00.600000+00:00                    7.0             7.0
    2017-07-14 02:40:00.700000+00:00                    8.0             8.0
    2017-07-14 02:40:00.800000+00:00                    9.0             9.0
    2017-07-14 02:40:00.900000+00:00                   10.0            10.0
```

Use the stream uuids as their column names instead, using a lambda function.

```
>>> streamset.filter(start=1500000000000000000, end=1500000000900000001)
...           .arrow_to_dataframe(
...              name_callable=lambda s: str(s.uuid)
...           )
                                c9fd8735-5ec5-4141-9a51-d23e1b2dfa42  9173fa70-
↪87ab-4ac8-ac08-4fd63b910cae
    time
    2017-07-14 02:40:00+00:00                                   1.0          ␣
↪                          1.0
    2017-07-14 02:40:00.100000+00:00                            2.0          ␣
↪                          2.0
    2017-07-14 02:40:00.200000+00:00                            3.0          ␣
↪                          3.0
    2017-07-14 02:40:00.300000+00:00                            4.0          ␣
↪                          4.0
    2017-07-14 02:40:00.400000+00:00                            5.0          ␣
↪                          5.0
    2017-07-14 02:40:00.500000+00:00                            6.0          ␣
↪                          6.0
    2017-07-14 02:40:00.600000+00:00                            7.0          ␣
↪                          7.0
    2017-07-14 02:40:00.700000+00:00                            8.0          ␣
↪                          8.0
    2017-07-14 02:40:00.800000+00:00                            9.0          ␣
↪                          9.0
    2017-07-14 02:40:00.900000+00:00                           10.0          ␣
↪                         10.0
```

A window query, with a window width of 0.4 seconds, and only showing the `mean` statpoint.

```
>>> streamset.filter(start=1500000000000000000, end=1500000000900000001)
...           .windows(width=int(0.4*10**9))
...           .arrow_to_dataframe(agg=["mean"])
                                          new/stream/collection/foo/mean  new/
↪stream/bar/mean
    time
    2017-07-14 02:40:00+00:00                                    2.5          ␣
↪  2.5
    2017-07-14 02:40:00.400000+00:00                             6.5          ␣
↪  6.5
```

A window query, with a window width of 0.4 seconds, and only showing the `mean` and `count` statpoints.

```
>>> streamset.filter(start=1500000000000000000, end=1500000000900000001)
...          .windows(width=int(0.4*10**9))
...          .arrow_to_dataframe(agg=["mean", "count"])
                            new/stream/collection/foo/mean  new/stream/collection/
↪foo/count  new/stream/bar/mean  new/stream/bar/count
    time
    2017-07-14 02:40:00+00:00               2.5                                 4  ␣
↪              2.5                       4
    2017-07-14 02:40:00.400000+00:00        6.5                                 4  ␣
↪              6.5                       4
```

btrdb.transformers.**arrow_to_polars**(*streamset*, *agg=None*, *name_callable=None*)

Returns a Polars DataFrame object with time as a column and the values of a stream for each additional column from an arrow table.

Parameters

- **agg** (*List[str], default: ["mean"]*) – Specify the StatPoint field or fields (e.g. aggregating function) to create the dataframe from. Must be one or multiple of "min", "mean", "max", "count", "stddev", or "all". This argument is ignored if not using StatPoints.
- **name_callable** (*lambda, default: lambda s: s.collection + "/" + s.name*) – Specify a callable that can be used to determine the series name given a Stream object.

Note: This method is available for commercial customers with arrow-enabled servers.

---

btrdb.transformers.**arrow_to_arrow_table**(*streamset*)

Return a pyarrow table of data.

Note: This method is available for commercial customers with arrow-enabled servers.

---

btrdb.transformers.**to_dict**(*streamset*, *agg='mean'*, *name_callable=None*)

Returns a list of OrderedDict for each time code with the appropriate stream data attached.

Parameters

- **agg** (*str, default: "mean"*) – Specify the StatPoint field (e.g. aggregating function) to constrain dict keys. Must be one of "min", "mean", "max", "count", or "stddev". This argument is ignored if RawPoint values are passed into the function.
- **name_callable** (*lambda, default: lambda s: s.collection + "/" + s.name*) – Specify a callable that can be used to determine the series name given a Stream object.

Note: This method does **not** use the `arrow` -accelerated endpoints for faster and more efficient data retrieval.

---

btrdb.transformers.**to_array**(*streamset*, *agg='mean'*)

Returns a multidimensional numpy array (similar to a list of lists) containing point classes.

Parameters

agg (*str, default: "mean"*) – Specify the StatPoint field (e.g. aggregating function)

to return for the arrays. Must be one of "min", "mean", "max", "count", or "stddev". This argument is ignored if RawPoint values are passed into the function.

Note: This method does **not** use the `arrow` -accelerated endpoints for faster and more efficient data retrieval.

`btrdb.transformers.to_polars(`*streamset*, *agg='mean'*, *name_callable=None*`)`

Returns a Polars DataFrame object with time as a column and the values of a stream for each additional column.

Parameters

- **agg** (*str, default: "mean"*) – Specify the StatPoint field (e.g. aggregating function) to create the Series from. Must be one of "min", "mean", "max", "count", "stddev", or "all". This argument is ignored if not using StatPoints.
- **name_callable** (*lambda, default: lambda s: s.collection + "/" + s.name*) – Specify a callable that can be used to determine the series name given a Stream object. This is not compatible with agg == "all" at this time

---

Note: This method does **not** use the `arrow` -accelerated endpoints for faster and more efficient data retrieval.

`btrdb.transformers.to_series(`*streamset*, *datetime64_index=True*, *agg='mean'*, *name_callable=None*`)`

Returns a list of Pandas Series objects indexed by time

Parameters

- **datetime64_index** (*bool*) – Directs function to convert Series index to np.datetime64[ns] or leave as np.int64.
- **agg** (*str, default: "mean"*) – Specify the StatPoint field (e.g. aggregating function) to create the Series from. Must be one of "min", "mean", "max", "count", or "stddev". This argument is ignored if RawPoint values are passed into the function.
- **name_callable** (*lambda, default: lambda s: s.collection + "/" + s.name*) – Specify a callable that can be used to determine the series name given a Stream object.

Note: This method does **not** use the `arrow` -accelerated endpoints for faster and more efficient data retrieval.

`btrdb.transformers.to_dataframe(`*streamset*, *agg='mean'*, *name_callable=None*`)`

Returns a Pandas DataFrame object indexed by time and using the values of a stream for each column.

Parameters

- **agg** (*str, default: "mean"*) – Specify the StatPoint field (e.g. aggregating function) to create the Series from. Must be one of "min", "mean", "max", "count", "stddev", or "all". This argument is ignored if not using StatPoints.
- **name_callable** (*lambda, default: lambda s: s.collection + "/" + s.name*) – Specify a callable that can be used to determine the series name given a Stream object. This is not compatible with agg == "all" at this time

---

Note: This method does **not** use the `arrow` -accelerated endpoints for faster and more efficient data retrieval.

---

btrdb.transformers.**to_csv**(*streamset*, *fobj*, *dialect=None*, *fieldnames=None*, *agg='mean'*, *name_callable=None*)

Saves stream data as a CSV file.

Parameters

- **fobj** (*str or file-like object*) – Path to use for saving CSV file or a file-like object to use to write to.
- **dialect** (*csv.Dialect*) – CSV dialect object from Python csv module. See Python's csv module for more information.
- **fieldnames** (*sequence*) – A sequence of strings to use as fieldnames in the CSV header. See Python's csv module for more information.
- **agg** (*str, default: "mean"*) – Specify the StatPoint field (e.g. aggregating function) to return when limiting results. Must be one of "min", "mean", "max", "count", or "stddev". This argument is ignored if RawPoint values are passed into the function.
- **name_callable** (*lambda, default: lambda s: s.collection + "/" + s.name*) – Specify a callable that can be used to determine the series name given a Stream object.

Note: This method does **not** use the `arrow` -accelerated endpoints for faster and more efficient data retrieval.

1.6.7 btrdb.utils.timez

Time related utilities

btrdb.utils.timez.**currently_as_ns**()

Returns the current UTC time as nanoseconds since epoch

btrdb.utils.timez.**datetime_to_ns**(*dt*)

Converts a datetime object to nanoseconds since epoch. If a timezone aware object is received then it will be converted to UTC.

Parameters
dt (*datetime*)

Returns
nanoseconds

Return type
int

btrdb.utils.timez.**ns_delta**(*days=0*, *hours=0*, *minutes=0*, *seconds=0*, *milliseconds=0*, *microseconds=0*, *nanoseconds=0*)

Similar to `timedelta`, `ns_delta` represents a span of time but as the total number of nanoseconds.

Parameters

- **days** (*int, float, decimal.Decimal*) – days (as 24 hours) to convert to nanoseconds
- **hours** (*int, float, decimal.Decimal*) – hours to convert to nanoseconds
- **minutes** (*int, float, decimal.Decimal*) – minutes to convert to nanoseconds
- **seconds** (*int, float, decimal.Decimal*) – seconds to convert to nanoseconds

- **milliseconds** (*int, float, decimal.Decimal*) – milliseconds to convert to nanoseconds
- **microseconds** (*int, float, decimal.Decimal*) – microseconds to convert to nanoseconds
- **nanoseconds** (*int*) – nanoseconds to add to the time span

Returns
amount of time in nanoseconds

Return type
int

Examples

1 minute time delta should be 60 billion nanoseconds

```
>>> deltaT = ns_delta(minutes=1)
>>> deltaT == 1 * 60 * 10**9 # 1 minute * 60 seconds * 1billion nanoseconds / second
True
```

btrdb.utils.timez.**ns_to_datetime**(*ns*)

Converts nanoseconds to a naive datetime object (UTC+0)

Parameters
ns (*int*) – nanoseconds since epoch

Returns
nanoseconds since epoch as a datetime object

Return type
datetime

btrdb.utils.timez.**to_nanoseconds**(*val*)

Converts datetime, datetime64, float, str (RFC 2822) to nanoseconds. If a datetime-like object is received then nanoseconds since epoch is returned.

Parameters
val (*datetime, datetime64, float, str*) – an object to convert to nanoseconds

Returns
object converted to nanoseconds

Return type
int

Notes

The following string formats are supported for conversion.

Format String	Description
%Y-%m-%d %H:%M:%S.%f%z	RFC3339 format
%Y-%m-%d %H:%M:%S.%f	RFC3339 with UTC default timezone
%Y-%m-%dT%H:%M:%S.%fZ	JSON encoding, UTC timezone
%Y-%m-%dT%H:%M:%SZ	JSON encoding, UTC timezone, without s
%Y-%m-%dT%H:%M:%S.%f%z	JSON-like encoding
%Y-%m-%dT%H:%M:%S.%f	JSON-like encoding, UTC default timezone
%Y-%m-%d %H:%M:%S%z	human readable date time with TZ
%Y-%m-%d %H:%M:%S	human readable date time UTC default
%Y-%m-%d	midnight at a particular date

1.7 Changelog

1.7.1 5.32.0

What's Changed

- Initial docstring overhaul and a new test for better documentation and test coverage. by @JustinGilmer in #82
- Test new join logic for improved data loading for windowed queries. by @JustinGilmer in #80
- Improve `arrow_to_dataframe` function for handling large amounts of columns, enhancing performance and usability. by @Jefflinf in #73
- Expand testing to include Python 3.11, ensuring compatibility and stability. by @JustinGilmer in #74
- Update exception handling to better support `RpcErrors`, improving error management and debugging. by @JustinGilmer in #72
- Introduce an option for specifying the schema of the returned raw data, allowing for more flexibility in data handling. by @andrewchambers in #51
- Remove non-required dependencies and migrate to 'data' optional dependency for a lighter package and easier installation. by @JustinGilmer in #71
- New method to get first and last timestamps from `aligned_windows`, enhancing data analysis capabilities. by @Jefflinf in #70
- Add `to_timedelta` method for `pointwidth` class, providing more options for time-based data manipulation. by @Jefflinf in #69

Fixed

- Fix `NoneType` error for `earliest/latest` for empty streams, ensuring reliability and error handling. by @Jefflinf in #64
- Correct integration tests where the time column is not automatically set as the index, improving test accuracy and reliability. by @JustinGilmer in #56

Deprecated

- FutureWarning for `streams_in_collection` to return `StreamSet` in the future, preparing users for upcoming API changes. by @Jefflinf in #60

Full Changelog: GitHub compare view

1.7.2 5.31.0

What's Changed

- Have release script update pyproject.toml file by @youngale-pingthings in https://github.com/PingThingsIO/btrdb-python/pull/48
- Provide option to sort the arrow tables by @justinGilmer in https://github.com/PingThingsIO/btrdb-python/pull/47
- Remove 4MB limit for gRPC message payloads by @justinGilmer in https://github.com/PingThingsIO/btrdb-python/pull/49
- Update documentation for arrow methods by @justinGilmer in https://github.com/PingThingsIO/btrdb-python/pull/50
- Update from staging by @justinGilmer in https://github.com/PingThingsIO/btrdb-python/pull/54
- Sort tables by time by default for any `pyarrow` tables. by @justinGilmer in
- Fix deprecation warnings for pip installations. by @jleifnf in

Full Changelog: GitHub compare view

1.7.3 5.30.2

What's Changed

- Update readthedocs to new yaml for testing. by @justinGilmer in https://github.com/PingThingsIO/btrdb-python/pull/40
- Converting pandas index takes very long, fix this in arrow_table. by @justinGilmer in https://github.com/PingThingsIO/btrdb-python/pull/41

Full Changelog: 5.30.2

1.7.4 5.30.1

What's Changed

- Small version bump for pypi release

Full Changelog: 5.30.1

1.7.5 5.30.0

What's Changed

- Merge Arrow support into Main for Release by @youngale-pingthings in https://github.com/PingThingsIO/btrdb-python/pull/37
  - This PR contains many changes that support the commercial Arrow data fetches and inserts
  - `arrow_` prefixed methods for `Stream` Objects:
    - `insert, aligned_windows, windows, values`
  - `arrow_` prefixed methods for StreamSet` objects:
    - `insert, values, to_dataframe, to_polars, to_arrow_table, to_numpy, to_dict, to_series`
- Justin gilmer patch 1 by @justinGilmer in https://github.com/PingThingsIO/btrdb-python/pull/39

Full Changelog: 5.30.0

1.7.6 5.28.1

What's Changed

- Upgrade ray versions by @jleifnf in https://github.com/PingThingsIO/btrdb-python/pull/15
- Release v5.28.1 and Update Python by @youngale-pingthings in https://github.com/PingThingsIO/btrdb-python/pull/17

New Contributors

- @jleifnf made their first contribution in https://github.com/PingThingsIO/btrdb-python/pull/15

Full Changelog: 5.28.1

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

b

INDEX