Streamz Documentation

Release 0.0.1

Matthew Rocklin

Feb 01, 2019

Contents

Streamz helps you build pipelines to manage continuous streams of data. It is simple to use in simple cases, but also supports complex pipelines that involve branching, joining, flow control, feedback, back pressure, and so on.

Optionally, Streamz can also work with Pandas dataframes to provide sensible streaming operations on continuous tabular data.

To learn more about how to use streams, visit *Core documentation*.

CHAPTER 1

Motivation

Continuous data streams arise in many applications like the following:

1. Log processing from web servers
2. Scientific instrument data like telemetry or image processing pipelines
3. Financial time series
4. Machine learning pipelines for real-time and on-line learning
5. ...

Sometimes these pipelines are very simple, with a linear sequence of processing steps:

And sometimes these pipelines are more complex, involving branching, look-back periods, feedback into earlier stages, and more.

Streamz endeavors to be simple in simple cases, while also being powerful enough to let you define custom and powerful pipelines for your application.

1.1 Why not Python generator expressions?

Python users often manage continuous sequences of data with iterators or generator expressions.

```
def fib():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

sequence = (f(n) for n in fib())
```

However iterators become challenging when you want to fork them or control the flow of data. Typically people rely on tools like `itertools.tee`, and `zip`.

```
x1, x2 = itertools.tee(x, 2)
y1 = map(f, x1)
y2 = map(g, x2)
```

However this quickly become cumbersome, especially when building complex pipelines.

CHAPTER 2

Related Work

Streamz is similar to reactive programming systems like RxPY or big data streaming systems like Apache Flink, Apache Beam or Apache Spark Streaming.

2.1 Core Streams

This document takes you through how to build basic streams and push data through them. We start with map and accumulate, talk about emitting data, then discuss flow control and finally back pressure. Examples are used throughout.

2.1.1 Map, emit, and sink

`Stream.emit`(x[, asynchronous])	Push data into the stream at this point
`map`(upstream, func, *args, **kwargs)	Apply a function to every element in the stream
`sink`(upstream, func, *args, **kwargs)	Apply a function on every element

You can create a basic pipeline by instantiating the `Streamz` object and then using methods like `map`, `accumulate`, and `sink`.

```
from streamz import Stream

def increment(x):
    return x + 1

source = Stream()
source.map(increment).sink(print)
```

The `map` and `sink` methods both take a function and apply that function to every element in the stream. The `map` method returns a new stream with the modified elements while `sink` is typically used at the end of a stream for final actions.

To push data through our pipeline we call `emit`

```
>>> source.emit(1)
2
>>> source.emit(2)
3
>>> source.emit(10)
11
```

As we can see, whenever we push data in at the source, our pipeline calls `increment` on that data, and then calls `print` on that data, resulting in incremented results being printed to the screen.

Often we call `emit` from some other continuous process, like reading lines from a file

```
import json

data = []

source = Stream()
source.map(json.loads).sink(data.append)

for line in open('myfile.json'):
    source.emit(line)
```

2.1.2 Accumulating State

accumulate(upstream, func[, start, ...])	Accumulate results with previous state

Map and sink both pass data directly through a stream. One piece of data comes in, either one or zero pieces go out. Accumulate allows you to track some state within the pipeline. It takes an accumulation function that takes the previous state, the new element, and then returns a new state and a new element to emit. In the following example we make an accumulator that keeps a running total of the elements seen so far.

```
def add(x, y):
    return x + y

source = Stream()
source.accumulate(add).sink(print)
```

```
>>> source.emit(1)
1
>>> source.emit(2)
3
>>> source.emit(3)
6
>>> source.emit(4)
10
```

The accumulation function above is particularly simple, the state that we store and the value that we emit are the same. In more complex situations we might want to keep around different state than we emit. For example lets count the number of distinct elements that we have seen so far.

```
def num_distinct(state, new):
    state.add(new)
    return state, len(state)
```

(continues on next page)

(continued from previous page)

```
source = Stream()
source.accumulate(num_distinct, returns_state=True, start=set()).sink(print)

>>> source.emit('cat')
1
>>> source.emit('dog')
2
>>> source.emit('cat')
2
>>> source.emit('mouse')
3
```

Accumulators allow us to build many interesting operations.

2.1.3 Flow Control

buffer(upstream, n, **kwargs)	Allow results to pile up at this point in the stream
flatten([upstream, upstreams, stream_name, ...])	Flatten streams of lists or iterables into a stream of elements
partition(upstream, n, **kwargs)	Partition stream into tuples of equal size
sliding_window(upstream, n, **kwargs)	Produce overlapping tuples of size n
union(*upstreams, **kwargs)	Combine multiple streams into one
unique(upstream[, history, key])	Avoid sending through repeated elements

You can batch and slice streams into streams of batches in various ways with operations like `partition`, `buffer`, and `sliding_window`

```
source = Stream()
source.sliding_window(3).sink(print)

>>> source.emit(1)
>>> source.emit(2)
>>> source.emit(3)
(1, 2, 3)
>>> source.emit(4)
(2, 3, 4)
>>> source.emit(5)
(3, 4, 5)
```

2.1.4 Branching and Joining

combine_latest(*upstreams, **kwargs)	Combine multiple streams together to a stream of tuples
zip(*upstreams, **kwargs)	Combine streams together into a stream of tuples
zip_latest(lossless, *upstreams, **kwargs)	Combine multiple streams together to a stream of tuples

You can branch multiple streams off of a single stream. Elements that go into the input will pass through to both output streams.

```
def increment(x):
    return x + 1
```

(continues on next page)

(continued from previous page)

```
def decrement(x):
    return x - 1

source = Stream()
a = source.map(increment).sink(print)
b = source.map(decrement).sink(print)
b.visualize(rankdir='LR')
```

```
>>> source.emit(1)
0
2
>>> source.emit(10)
9
11
```

Similarly you can also combine multiple streams together with operations like `zip`, which emits once both streams have provided a new element, or `combine_latest` which emits when either stream has provided a new element.

```
source = Stream()
a = source.map(increment)
b = source.map(decrement)
c = a.zip(b).map(sum).sink(print)

>>> source.emit(10)
20  # 9 + 11
```

This branching and combining is where Python iterators break down, and projects like `streamz` start becoming valuable.

2.1.5 Processing Time and Back Pressure

delay(upstream, interval, **kwargs)	Add a time delay to results
rate_limit(upstream, interval, **kwargs)	Limit the flow of data
timed_window(upstream, interval, **kwargs)	Emit a tuple of collected results every interval

Time-based flow control depends on having an active Tornado event loop. Tornado is active by default within a Jupyter notebook, but otherwise you will need to learn at least a little about asynchronous programming in Python to use these features. Learning async programming is not mandatory, the rest of the project will work fine without Tornado.

You can control the flow of data through your stream over time. For example you may want to batch all elements that have arrived in the last minute, or slow down the flow of data through sensitive parts of the pipeline, particularly when they may be writing to slow resources like databases.

Streamz helps you do these operations both with operations like `delay`, `rate_limit`, and `timed_window`, and also by passing Tornado futures back through the pipeline. As data moves forward through the pipeline, futures that signal work completed move backwards. In this way you can reliably avoid buildup of data in slower parts of your pipeline.

Lets consider the following example that reads JSON data from a file and inserts it into a database using an async-aware insertion function.

```
async def write_to_database(...):
    ...

# build pipeline
source = Source()
source.map(json.loads).sink(write_to_database)

async def process_file(fn):
    with open(fn) as f:
        for line in f:
            await source.emit(line)  # wait for pipeline to clear
```

As we call the `write_to_database` function on our parsed JSON data it produces a future for us to signal that the writing process has finished. Streamz will ensure that this future is passed all the way back to the `source.emit` call, so that user code at the start of our pipeline can await on it. This allows us to avoid buildup even in very large and complex streams. We always pass futures back to ensure responsiveness.

But wait, maybe we don't mind having a few messages in memory at once, this will help steady the flow of data so that we can continue to work even if our sources or sinks become less productive for brief periods. We might add a `buffer` just before writing to the database.

```
source.map(json.loads).buffer(100).sink(write_to_database)
```

And if we are pulling from an API with known limits then we might want to introduce artificial rate limits at 10ms.

```
source.rate_limit(0.010).map(json.loads).buffer(100).sink(write_to_database)
```

Operations like these (and more) allow us to shape the flow of data through our pipelines.

2.1.6 Modifying and Cleaning up Streams

When you call `Stream` you create a stream. When you call any method on a `Stream`, like `Stream.map`, you also create a stream. All operations can be chained together. Additionally, as discussed in the section on Branching, you can split multiple streams off of any point. Streams will pass their outputs on to all downstream streams so that anyone can hook in at any point, and get a full view of what that stream is producing.

If you delete a part of a stream then it will stop getting data. Streamz follows normal Python garbage collection semantics so once all references to a stream have been lost those operations will no longer occur. The one counter example to this is `sink`, which is intended to be used with side effects and will stick around even without a reference.

Note: Sink streams store themselves in `streamz.core._global_sinks`. You can remove them permanently by clearing that collection.

```
>>> source.map(print)       # this doesn't do anything
>>> source.sink(print)      # this stays active even without a reference
>>> s = source.map(print)   # this works too because we have a handle to s
```

2.1.7 Recursion and Feedback

By connecting sources to sinks you can create feedback loops. As an example, here is a tiny web crawler:

```
from streamz import Stream
source = Stream()

pages = source.unique()
content = pages.map(requests.get).map(lambda x: x.content)
links = content.map(get_list_of_links).flatten()
links.sink(source.emit)  # pipe new links back into pages

pages.sink(print)

>>> source.emit('http://github.com')
http://github.com
http://github.com/features
http://github.com/business
http://github.com/explore
http://github.com/pricing
...
```

2.1.8 Performance

Streamz adds microsecond overhead to normal Python operations.

```
from streamz import Stream

source = Stream()

def inc(x):
    return x + 1

source.sink(inc)

In [5]: %timeit source.emit(1)
100000 loops, best of 3: 3.19 µs per loop

In [6]: %timeit inc(1)
10000000 loops, best of 3: 91.5 ns per loop
```

You may want to avoid pushing millions of individual elements per second through a stream. However, you can avoid performance issues by collecting lots of data into single elements, for example by pushing through Pandas dataframes instead of individual integers and strings. This will be faster regardless, just because projects like NumPy and Pandas can be much faster than Python generally.

In the following example we pass filenames through a stream, convert them to Pandas dataframes, and then map pandas-level functions on those dataframes. For operations like this Streamz adds virtually no overhead.

```
source = Stream()
s = source.map(pd.read_csv).map(lambda df: df.value.sum()).accumulate(add)

for fn in glob('data/2017-*-*.csv'):
    source.emit(fn)
```

Streams provides higher level APIs for situations just like this one. You may want to read further about *collections*

2.2 DataFrames

When handling large volumes of streaming tabular data it is often more efficient to pass around larger Pandas dataframes with many rows each rather than pass around individual Python tuples or dicts. Handling and computing on data with Pandas can be much faster than operating on Python objects.

So one could imagine building streaming dataframe pipelines using the `.map` and `.accumulate` streaming operators with functions that consume and produce Pandas dataframes as in the following example:

```
from streamz import Stream

def query(df):
    return df[df.name == 'Alice']

def aggregate(acc, df):
    return acc + df.amount.sum()

stream = Stream()
stream.map(query).accumulate(aggregate, start=0)
```

This is fine, and straightforward to do if you understand `streamz.core` , Pandas, and have some skill with developing algorithms.

2.2.1 Streaming Dataframes

The `streamz.dataframe` module provides a streaming dataframe object that implements many of these algorithms for you. It provides a Pandas-like interface on streaming data. Our example above is rewritten below using streaming dataframes:

```
import pandas as pd
from streamz.dataframe import DataFrame

example = pd.DataFrame({'name': [], 'amount': []})
sdf = DataFrame(stream, example=example)

sdf[sdf.name == 'Alice'].amount.sum()
```

The two examples are identical in terms of performance and execution. The resulting streaming dataframe contains a `.stream` attribute which is equivalent to the `stream` produced in the first example. Streaming dataframes are only syntactic sugar on core streams.

2.2.2 Supported Operations

Streaming dataframes support the following classes of operations

- Elementwise operations like `df.x + 1`
- Filtering like `df[df.name == 'Alice']`
- Column addition like `df['z'] = df.x + df.y`
- Reductions like `df.amount.mean()`
- Groupby-aggregations like `df.groupby(df.name).amount.mean()`
- Windowed aggregations (fixed length) like `df.window(n=100).amount.sum()`

- Windowed aggregations (index valued) like `df.window(value='2h').amount.sum()`
- Windowed groupby aggregations like `df.window(value='2h').groupby('name').amount.sum()`

2.2.3 DataFrame Aggregations

Dataframe aggregations are composed of an aggregation (like sum, mean, ...) and a windowing scheme (fixed sized windows, index-valued, all time, ...)

Aggregations

Streaming Dataframe aggregations are built from three methods

- `initial`: Creates initial state given an empty example dataframe
- `on_new`: Updates state and produces new result to emit given new data
- `on_old`: Updates state and produces new result to emit given decayed data

So a simple implementation of `sum` as an aggregation might look like the following:

```
from streamz.dataframe import Aggregation

class Mean(Aggregation):
    def initial(self, new):
        state = new.iloc[:0].sum(), new.iloc[:0].count()
        return state

    def on_new(self, state, new):
        total, count = state
        total = total + new.sum()
        count = count + new.count()
        new_state = (total, count)
        new_value = total / count
        return new_state, new_value

    def on_old(self, state, old):
        total, count = state
        total = total - new.sum()    # switch + for - here
        count = count - new.count() # switch + for - here
        new_state = (total, count)
        new_value = total / count
        return new_state, new_value
```

These aggregations can then used in a variety of different windowing schemes with the `aggregate` method as follows:

```
df.aggregate(Mean())

df.window(n=100).aggregate(Mean())

df.window(value='60s').aggregate(Mean())
```

whose job it is to deliver new and old data to your aggregation for processing.

Windowing Schemes

Different windowing schemes like fixed sized windows (last 100 elements) or value-indexed windows (last two hours of data) will track newly arrived and decaying data and call these methods accordingly. The mechanism to track data arriving and leaving is kept orthogonal from the aggregations themselves. These windowing schemes include the following:

1. All previous data. Only `initial` and `on_new` are called, `on_old` is never called.

```
>>> df.sum()
```

2. The previous `n` elements

```
>>> df.window(n=100).sum()
```

3. An index range, like a time range for a datetime index

```
>>> df.window(value='2h').sum()
```

   Although this can be done for any range on any type of index, time is just a common case.

Windowing schemes generally maintain a deque of historical values within accumulated state. As new data comes in they inspect that state and eject data that no longer falls within the window.

Grouping

Groupby aggregations also maintain historical data on the grouper and perform a parallel aggregation on the number of times any key has been seen, removing that key once it is no longer present.

2.2.4 Dask

In all cases, dataframe operations are only implemented with the `.map` and `.accumulate` operators, and so are equally compatible with core `Stream` and `DaskStream` objects.

2.2.5 Not Yet Supported

Streaming dataframes algorithms do not currently pay special attention to data arriving out-of-order.

2.3 Dask Integration

The `streamz.dask` module contains a Dask-powered implementation of the core Stream object. This is a drop-in implementation, but uses Dask for execution and so can scale to a multicore machine or a distributed cluster.

2.3.1 Quickstart

Installation

First install dask and dask.distributed:

```
conda install dask
or
pip install dask[complete] --upgrade
```

You may also want to install Bokeh for web diagnostics:

```
conda install -c bokeh bokeh
or
pip install bokeh --upgrade
```

Start Local Dask Client

Then start a local Dask cluster

```
from dask.distributed import Client
client = Client()
```

This operates on a local processes or threads. If you have Bokeh installed then this will also start a diagnostics web server at http://localhost:8787/status which you may want to open to get a real-time view of execution.

Sequential Execution

`Stream.emit`(x[, asynchronous])	Push data into the stream at this point
map(upstream, func, *args, **kwargs)	Apply a function to every element in the stream
sink(upstream, func, *args, **kwargs)	Apply a function on every element

Before we build a parallel stream, lets build a sequential stream that maps a simple function across data, and then prints those results. We use the core `Stream` object.

```
from time import sleep

def inc(x):
    sleep(1)  # simulate actual work
    return x + 1

from streamz import Stream

source = Stream()
source.map(inc).sink(print)

for i in range(10):
    source.emit(i)
```

This should take ten seconds we call the `inc` function ten times sequentially.

Parallel Execution

`scatter`(*args, **kwargs)	Convert local stream to Dask Stream
buffer(upstream, n, **kwargs)	Allow results to pile up at this point in the stream

gather([upstream, upstreams, stream_name, ...])	Wait on and gather results from DaskStream to local Stream

That example ran sequentially under normal execution, now we use `.scatter()` to convert our stream into a DaskStream and `.gather()` to convert back.

```
source = Stream()
source.scatter().map(inc).buffer(8).gather().sink(print)

for i in range(10):
    source.emit(i)
```

You may want to look at http://localhost:8787/status during execution to get a sense of the parallel execution.

This should have run much more quickly depending on how many cores you have on your machine. We added a few extra nodes to our stream, lets look at what they did.

- `scatter`: Converted our Stream into a DaskStream. The elements that we emitted into our source were sent to the Dask client, and the subsequent `map` call used that client's cores to perform the computations.
- `gather`: Converted our DaskStream back into a Stream, pulling data on our Dask client back to our local stream
- `buffer(5)`: Normally gather would exert back pressure so that the source would not accept new data until results finished and were pulled back to the local stream. This back-pressure would limit parallelism. To counteract this we add a buffer of size eight to allow eight unfinished futures to build up in the pipeline before we start to apply back-pressure to `source.emit`.

2.4 Collections

Streamz high-level collection APIs are built on top of `streamz.core`, and bring special consideration to certain types of data:

1. `streamz.batch`: supports streams of lists of Python objects like tuples or dictionaries
2. `streamz.dataframe`: supports streams of Pandas dataframes or Pandas series

These high-level APIs help us handle common situations in data processing. They help us implement complex algorithms and also improve efficiency.

These APIs are built on the streamz core operations (map, accumulate, buffer, timed_window, ...) which provide the building blocks to build complex pipelines but offer no help with what those functions should be. The higher-level APIs help to fill in this gap for common situations.

2.4.1 Conversion

`Stream.to_batch`(**kwargs)	Convert a stream of lists to a Batch
`Stream.to_dataframe`(example)	Convert a stream of Pandas dataframes to a DataFrame

You can convert from core Stream objects to Batch, and DataFrame objects using the `.to_batch` and `.to_dataframe` methods. In each case we assume that the stream is a stream of batches (lists or tuples) or a list of Pandas dataframes.

```
>>> batch = stream.to_batch()
>>> sdf = stream.to_dataframe()
```

To convert back from a Batch or a DataFrame to a `core.Stream` you can access the `.stream` property.

```
>>> stream = sdf.stream
>>> stream = batch.stream
```

2.4.2 Example

We create a stream and connect it to a file object

```
file = ...  # filename or file-like object
from streamz import Stream

source = Stream.from_textfile(file)
```

Our file produces line-delimited JSON serialized data on which we want to call `json.loads` to parse into dictionaries.

To reduce overhead we first batch our records up into 100-line batches and turn this into a Batch object. We provide our Batch object an example element that it will use to help it determine metadata.

```
example = [{'name': 'Alice', 'x': 1, 'y': 2}]
lines = source.partition(100).to_batch(example=example)  # batches of 100 elements
records = lines.map(json.loads)  # convert lines to text.
```

We could have done the `.map(json.loads)` command on the original stream, but this way reduce overhead by applying this function to lists of items, rather than one item at a time.

Now we convert these batches of records into pandas dataframes and do some basic filtering and groupby-aggregations.

```
sdf = records.to_dataframe()
sdf = sdf[sdf.name == "Alice"]
sdf = sdf.groupby(sdf.x).y.mean()
```

The DataFrames satisfy a subset of the Pandas API, but now rather than operate on the data directly, they set up a pipeline to compute the data in an online fashion.

Finally we convert this back to a stream and push the results into a fixed-size deque.

```
from collections import deque
d = deque(maxlen=10)

sdf.stream.sink(d.append)
```

See *Collections API* for more information.

2.5 API

2.5.1 Stream

Stream([upstream, upstreams, stream_name, ...])	A Stream is an infinite sequence of data

accumulate(upstream, func[, start, ...])	Accumulate results with previous state
buffer(upstream, n, **kwargs)	Allow results to pile up at this point in the stream
collect(upstream[, cache])	Hold elements in a cache and emit them as a collection when flushed.
combine_latest(*upstreams, **kwargs)	Combine multiple streams together to a stream of tuples
Stream.connect(downstream)	Connect this stream to a downstream element.
delay(upstream, interval, **kwargs)	Add a time delay to results
Stream.destroy([streams])	Disconnect this stream from any upstream sources
Stream.disconnect(downstream)	Disconnect this stream to a downstream element.
filter(upstream, predicate, **kwargs)	Only pass through elements that satisfy the predicate
flatten([upstream, upstreams, stream_name, ...])	Flatten streams of lists or iterables into a stream of elements
map(upstream, func, *args, **kwargs)	Apply a function to every element in the stream
partition(upstream, n, **kwargs)	Partition stream into tuples of equal size
rate_limit(upstream, interval, **kwargs)	Limit the flow of data
scatter(*args, **kwargs)	Convert local stream to Dask Stream
sink(upstream, func, *args, **kwargs)	Apply a function on every element
sliding_window(upstream, n, **kwargs)	Produce overlapping tuples of size n
starmap(upstream, func, *args, **kwargs)	Apply a function to every element in the stream, splayed out
timed_window(upstream, interval, **kwargs)	Emit a tuple of collected results every interval
union(*upstreams, **kwargs)	Combine multiple streams into one
unique(upstream[, history, key])	Avoid sending through repeated elements
pluck(upstream, pick, **kwargs)	Select elements from elements in the stream.
zip(*upstreams, **kwargs)	Combine streams together into a stream of tuples
zip_latest(lossless, *upstreams, **kwargs)	Combine multiple streams together to a stream of tuples

2.5.2 Sources

filenames(path[, poll_interval, start])	Stream over filenames in a directory
from_kafka(topics, consumer_params[, ...])	Accepts messages from Kafka
from_textfile(f[, poll_interval, delimiter, ...])	Stream data from a text file

2.5.3 DaskStream

DaskStream(*args, **kwargs)	A Parallel stream using Dask
gather([upstream, upstreams, stream_name, ...])	Wait on and gather results from DaskStream to local Stream

2.5.4 Definitions

streamz.**accumulate**(*upstream*, *func*, *start='--no-default--'*, *returns_state=False*, ***kwargs*)

Accumulate results with previous state

This performs running or cumulative reductions, applying the function to the previous total and the new element. The function should take two arguments, the previous accumulated state and the next element and it should return a new accumulated state.

Parameters

func: callable

start: object Initial value. Defaults to the first submitted element

returns_state: boolean If true then func should return both the state and the value to emit If false then both values are the same, and func returns one value

****kwargs:** Keyword arguments to pass to func

Examples

```
>>> source = Stream()
>>> source.accumulate(lambda acc, x: acc + x).sink(print)
>>> for i in range(5):
...     source.emit(i)
1
3
6
10
```

streamz.**buffer**(*upstream*, *n*, ***kwargs*)

Allow results to pile up at this point in the stream

This allows results to buffer in place at various points in the stream. This can help to smooth flow through the system when backpressure is applied.

streamz.**collect**(*upstream*, *cache=None*, ***kwargs*)

Hold elements in a cache and emit them as a collection when flushed.

Examples

```
>>> source1 = Stream()
>>> source2 = Stream()
>>> collector = collect(source1)
>>> collector.sink(print)
>>> source2.sink(collector.flush)
>>> source1.emit(1)
>>> source1.emit(2)
>>> source2.emit('anything')  # flushes collector
...
[1, 2]
```

streamz.**combine_latest**(**upstreams*, ***kwargs*)

Combine multiple streams together to a stream of tuples

This will emit a new tuple of all of the most recent elements seen from any stream.

Parameters

emit_on [stream or list of streams or None] only emit upon update of the streams listed. If None, emit on update from any stream

See also:

zip

streamz.**delay**(*upstream*, *interval*, ***kwargs*)
Add a time delay to results

streamz.**filter**(*upstream*, *predicate*, ***kwargs*)
Only pass through elements that satisfy the predicate

Parameters

predicate [function] The predicate. Should return True or False, where True means that the predicate is satisfied.

Examples

```
>>> source = Stream()
>>> source.filter(lambda x: x % 2 == 0).sink(print)
>>> for i in range(5):
...     source.emit(i)
0
2
4
```

streamz.**flatten**(*upstream=None*, *upstreams=None*, *stream_name=None*, *loop=None*, *asynchronous=None*, *ensure_io_loop=False*)
Flatten streams of lists or iterables into a stream of elements

See also:

partition

Examples

```
>>> source = Stream()
>>> source.flatten().sink(print)
>>> for x in [[1, 2, 3], [4, 5], [6, 7, 7]]:
...     source.emit(x)
1
2
3
4
5
6
7
```

streamz.**map**(*upstream*, *func*, **args*, ***kwargs*)
Apply a function to every element in the stream

Parameters

func: callable

***args :** The arguments to pass to the function.

****kwargs:** Keyword arguments to pass to func

Examples

```
>>> source = Stream()
>>> source.map(lambda x: 2*x).sink(print)
>>> for i in range(5):
...     source.emit(i)
0
2
4
6
8
```

streamz.**partition**(*upstream*, *n*, ***kwargs*)
Partition stream into tuples of equal size

Examples

```
>>> source = Stream()
>>> source.partition(3).sink(print)
>>> for i in range(10):
...     source.emit(i)
(0, 1, 2)
(3, 4, 5)
(6, 7, 8)
```

streamz.**rate_limit**(*upstream*, *interval*, ***kwargs*)
Limit the flow of data

This stops two elements of streaming through in an interval shorter than the provided value.

Parameters

interval: float Time in seconds

streamz.**sink**(*upstream*, *func*, **args*, ***kwargs*)
Apply a function on every element

See also:

map, Stream.sink_to_list

Examples

```
>>> source = Stream()
>>> L = list()
>>> source.sink(L.append)
>>> source.sink(print)
>>> source.sink(print)
>>> source.emit(123)
123
123
>>> L
[123]
```

streamz.**sliding_window**(*upstream*, *n*, ***kwargs*)
Produce overlapping tuples of size n

Examples

```
>>> source = Stream()
>>> source.sliding_window(3).sink(print)
>>> for i in range(8):
...     source.emit(i)
(0, 1, 2)
(1, 2, 3)
(2, 3, 4)
(3, 4, 5)
(4, 5, 6)
(5, 6, 7)
```

streamz.**Stream**(*upstream=None*, *upstreams=None*, *stream_name=None*, *loop=None*, *asynchronous=None*, *ensure_io_loop=False*)

A Stream is an infinite sequence of data

Streams subscribe to each other passing and transforming data between them. A Stream object listens for updates from upstream, reacts to these updates, and then emits more data to flow downstream to all Stream objects that subscribe to it. Downstream Stream objects may connect at any point of a Stream graph to get a full view of the data coming off of that point to do with as they will.

Parameters

asynchronous: boolean or None Whether or not this stream will be used in asynchronous functions or normal Python functions. Leave as None if you don't know. True will cause operations like emit to return awaitable Futures False will use an Event loop in another thread (starts it if necessary)

ensure_io_loop: boolean Ensure that some IOLoop will be created. If asynchronous is None or False then this will be in a separate thread, otherwise it will be IOLoop.current

Examples

```
>>> def inc(x):
...     return x + 1
```

```
>>> source = Stream()  # Create a stream object
>>> s = source.map(inc).map(str)  # Subscribe to make new streams
>>> s.sink(print)  # take an action whenever an element reaches the end
```

```
>>> L = list()
>>> s.sink(L.append)  # or take multiple actions (streams can branch)
```

```
>>> for i in range(5):
...     source.emit(i)  # push data in at the source
'1'
'2'
'3'
'4'
'5'
>>> L  # and the actions happen at the sinks
['1', '2', '3', '4', '5']
```

streamz.**timed_window**(*upstream*, *interval*, ***kwargs*)

Emit a tuple of collected results every interval

Every `interval` seconds this emits a tuple of all of the results seen so far. This can help to batch data coming off of a high-volume stream.

`streamz.`**`union`**(**upstreams*, ***kwargs*)
Combine multiple streams into one

Every element from any of the upstreams streams will immediately flow into the output stream. They will not be combined with elements from other streams.

See also:

`Stream.zip`, `Stream.combine_latest`

`streamz.`**`unique`**(*upstream*, *history=None*, *key=<function identity>*, ***kwargs*)
Avoid sending through repeated elements

This deduplicates a stream so that only new elements pass through. You can control how much of a history is stored with the `history=` parameter. For example setting `history=1` avoids sending through elements when one is repeated right after the other.

Examples

```
>>> source = Stream()
>>> source.unique(history=1).sink(print)
>>> for x in [1, 1, 2, 2, 2, 1, 3]:
...     source.emit(x)
1
2
1
3
```

`streamz.`**`pluck`**(*upstream*, *pick*, ***kwargs*)
Select elements from elements in the stream.

Parameters

pluck [object, list] The element(s) to pick from the incoming element in the stream If an instance of list, will pick multiple elements.

Examples

```
>>> source = Stream()
>>> source.pluck([0, 3]).sink(print)
>>> for x in [[1, 2, 3, 4], [4, 5, 6, 7], [8, 9, 10, 11]]:
...     source.emit(x)
(1, 4)
(4, 7)
(8, 11)
```

```
>>> source = Stream()
>>> source.pluck('name').sink(print)
>>> for x in [{'name': 'Alice', 'x': 123}, {'name': 'Bob', 'x': 456}]:
...     source.emit(x)
'Alice'
'Bob'
```

streamz.**zip**(**upstreams*, ***kwargs*)
Combine streams together into a stream of tuples

We emit a new tuple once all streams have produce a new tuple.

See also:

combine_latest, *zip_latest*

streamz.**zip_latest**(*lossless*, **upstreams*, ***kwargs*)
Combine multiple streams together to a stream of tuples

The stream which this is called from is lossless. All elements from the lossless stream are emitted reguardless of when they came in. This will emit a new tuple consisting of an element from the lossless stream paired with the latest elements from the other streams. Elements are only emitted when an element on the lossless stream are received, similar to `combine_latest` with the `emit_on` flag.

See also:

`Stream.combine_latest`, `Stream.zip`

streamz.**filenames**(*path*, *poll_interval=0.1*, *start=False*, ***kwargs*)
Stream over filenames in a directory

Parameters

path: string Directory path or globstring over which to search for files

poll_interval: Number Seconds between checking path

start: bool (False) Whether to start running immediately; otherwise call stream.start() explicitly.

Examples

```
>>> source = Stream.filenames('path/to/dir')
>>> source = Stream.filenames('path/to/*.csv', poll_interval=0.500)
```

streamz.**from_kafka**(*topics*, *consumer_params*, *poll_interval=0.1*, *start=False*, ***kwargs*)
Accepts messages from Kafka

Uses the confluent-kafka library, https://docs.confluent.io/current/clients/confluent-kafka-python/

Parameters

topics: list of str Labels of Kafka topics to consume from

consumer_params: dict Settings to set up the stream, see https://docs.confluent.io/current/clients/confluent-kafka-python/#configuration https://github.com/edenhill/librdkafka/blob/master/CONFIGURATION.md Examples: bootstrap.servers: Connection string(s) (host:port) by which to reach Kafka group.id: Identity of the consumer. If multiple sources share the same

group, each message will be passed to only one of them.

poll_interval: number Seconds that elapse between polling Kafka for new messages

start: bool (False) Whether to start polling upon instantiation

streamz.**from_textfile**(*f*, *poll_interval=0.1*, *delimiter='\n'*, *start=False*, ***kwargs*)
Stream data from a text file

Parameters

f: file or string

poll_interval: Number Interval to poll file for new data in seconds

delimiter: str (“

“)

Character(s) to use to split the data into parts

start: bool (False) Whether to start running immediately; otherwise call stream.start() explicitly.

Returns

Stream

`streamz.dask.`**`DaskStream`**(**args*, ***kwargs*)

A Parallel stream using Dask

This object is fully compliant with the `streamz.core.Stream` object but uses a Dask client for execution. Operations like `map` and `accumulate` submit functions to run on the Dask instance using `dask.distributed.Client.submit` and pass around Dask futures. Time-based operations like `timed_window`, buffer, and so on operate as normal.

Typically one transfers between normal Stream and DaskStream objects using the `Stream.scatter()` and `DaskStream.gather()` methods.

See also:

`dask.distributed.Client`

Examples

```
>>> from dask.distributed import Client
>>> client = Client()
```

```
>>> from streamz import Stream
>>> source = Stream()
>>> source.scatter().map(func).accumulate(binop).gather().sink(...)
```

`streamz.dask.`**`gather`**(*upstream=None*, *upstreams=None*, *stream_name=None*, *loop=None*, *asynchronous=None*, *ensure_io_loop=False*)

Wait on and gather results from DaskStream to local Stream

This waits on every result in the stream and then gathers that result back to the local stream. Warning, this can restrict parallelism. It is common to combine a `gather()` node with a `buffer()` to allow unfinished futures to pile up.

See also:

`buffer`, `scatter`

Examples

```
>>> local_stream = dask_stream.buffer(20).gather()
```

2.6 Collections API

2.6.1 Collections

Streaming([stream, example, stream_type])	Superclass for streaming collections
Streaming.map_partitions(func, *args, **kwargs)	Map a function across all batch elements of this stream
Streaming.accumulate_partitions(func, *args, ...)	Accumulate a function with state across batch elements
Streaming.verify(x)	Verify elements that pass through this stream

2.6.2 Batch

Batch([stream, example])	A Stream of tuples or lists
Batch.filter(predicate)	Filter elements by a predicate
Batch.map(func, **kwargs)	Map a function across all elements
Batch.pluck(ind)	Pick a field out of all elements
Batch.to_dataframe()	Convert to a streaming dataframe
Batch.to_stream()	Concatenate batches and return base Stream

2.6.3 Dataframes

DataFrame(*args, **kwargs)	A Streaming dataframe
DataFrame.groupby(other)	Groupby aggreagtions
DataFrame.rolling(window[, min_periods])	Compute rolling aggregations
DataFrame.assign(**kwargs)	Assign new columns to this dataframe
DataFrame.sum()	Sum frame
DataFrame.mean()	Average
DataFrame.cumsum()	Cumulative sum
DataFrame.cumprod()	Cumulative product
DataFrame.cummin()	Cumulative minimum
DataFrame.cummax()	Cumulative maximum

GroupBy(root, grouper[, index])	Groupby aggregations on streaming dataframes
GroupBy.count()	Groupby-count
GroupBy.mean()	Groupby-mean
GroupBy.size()	Groupby-size
GroupBy.std([ddof])	Groupby-std
GroupBy.sum()	Groupby-sum
GroupBy.var([ddof])	Groupby-variance

Rolling(sdf, window, min_periods)	Rolling aggregations
Rolling.aggregate(*args, **kwargs)	Rolling aggregation
Rolling.count(*args, **kwargs)	Rolling count
Rolling.max()	Rolling maximum
Rolling.mean()	Rolling mean

Continued on next page

Table 18 – continued from previous page

Rolling.median()	Rolling median
Rolling.min()	Rolling minimum
Rolling.quantile(*args, **kwargs)	Rolling quantile
Rolling.std(*args, **kwargs)	Rolling standard deviation
Rolling.sum()	Rolling sum
Rolling.var(*args, **kwargs)	Rolling variance

DataFrame.window([n, value])	Sliding window operations
Window.apply(func)	Apply an arbitrary function over each window of data
Window.count()	Count elements within window
Window.groupby(other)	Groupby-aggregations within window
Window.sum()	Sum elements within window
Window.size	Number of elements within window
Window.std([ddof])	Compute standard deviation of elements within window
Window.var([ddof])	Compute variance of elements within window

Rolling.aggregate(*args, **kwargs)	Rolling aggregation
Rolling.count(*args, **kwargs)	Rolling count
Rolling.max()	Rolling maximum
Rolling.mean()	Rolling mean
Rolling.median()	Rolling median
Rolling.min()	Rolling minimum
Rolling.quantile(*args, **kwargs)	Rolling quantile
Rolling.std(*args, **kwargs)	Rolling standard deviation
Rolling.sum()	Rolling sum
Rolling.var(*args, **kwargs)	Rolling variance

Random([freq, interval, dask])	A streaming dataframe of random data

2.6.4 Details

class streamz.collection.**Streaming**(*stream=None*, *example=None*, *stream_type=None*)

Superclass for streaming collections

Do not create this class directly, use one of the subclasses instead.

Parameters

stream: streamz.Stream

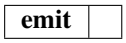
example: object An object to represent an example element of this stream

See also:

streamz.dataframe.StreamingDataFrame, streamz.dataframe.StreamingBatch

Methods

accumulate_partitions(func, *args, **kwargs)	Accumulate a function with state across batch elements
map_partitions(func, *args, **kwargs)	Map a function across all batch elements of this stream
verify(x)	Verify elements that pass through this stream



accumulate_partitions (*func*, **args*, ***kwargs*)
Accumulate a function with state across batch elements

See also:

Streaming.map_partitions

static map_partitions (*func*, **args*, ***kwargs*)
Map a function across all batch elements of this stream

The output stream type will be determined by the action of that function on the example

See also:

Streaming.accumulate_partitions

verify (*x*)
Verify elements that pass through this stream

class streamz.batch.**Batch** (*stream=None*, *example=None*)
A Stream of tuples or lists

This streaming collection manages batches of Python objects such as lists of text or dictionaries. By batching many elements together we reduce overhead from Python.

This library is typically used at the early stages of data ingestion before handing off to streaming dataframes

Examples

```
>>> text = Streaming.from_file(myfile)
>>> b = text.partition(100).map(json.loads)
```

Methods

accumulate_partitions(func, *args, **kwargs)	Accumulate a function with state across batch elements
filter(predicate)	Filter elements by a predicate
map(func, **kwargs)	Map a function across all elements
map_partitions(func, *args, **kwargs)	Map a function across all batch elements of this stream
pluck(ind)	Pick a field out of all elements
sum()	Sum elements
to_dataframe()	Convert to a streaming dataframe
to_stream()	Concatenate batches and return base Stream
verify(x)	Verify elements that pass through this stream

emit	

accumulate_partitions (*func*, **args*, ***kwargs*)
Accumulate a function with state across batch elements

See also:

`Streaming.map_partitions`

filter (*predicate*)
Filter elements by a predicate

map (*func*, ***kwargs*)
Map a function across all elements

static map_partitions (*func*, **args*, ***kwargs*)
Map a function across all batch elements of this stream

The output stream type will be determined by the action of that function on the example

See also:

`Streaming.accumulate_partitions`

pluck (*ind*)
Pick a field out of all elements

sum ()
Sum elements

to_dataframe ()
Convert to a streaming dataframe

This calls `pd.DataFrame` on all list-elements of this stream

to_stream ()
Concatenate batches and return base Stream

Returned stream will be composed of single elements

verify (*x*)
Verify elements that pass through this stream

class streamz.dataframe.**DataFrame** (**args*, ***kwargs*)
A Streaming dataframe

This is a logical collection over a stream of Pandas dataframes. Operations on this object will translate to the appropriate operations on the underlying Pandas dataframes.

See also:

`Series`

Attributes

columns

dtypes

index

size size of frame

Methods

accumulate_partitions(func, *args, **kwargs)	Accumulate a function with state across batch elements
assign(**kwargs)	Assign new columns to this dataframe
count()	Count of frame
cummax()	Cumulative maximum
cummin()	Cumulative minimum
cumprod()	Cumulative product
cumsum()	Cumulative sum
groupby(other)	Groupby aggreagtions
map_partitions(func, *args, **kwargs)	Map a function across all batch elements of this stream
mean()	Average
reset_index()	Reset Index
rolling(window[, min_periods])	Compute rolling aggregations
round([decimals])	Round elements in frame
set_index(index, **kwargs)	Set Index
sum()	Sum frame
tail([n])	Round elements in frame
to_frame()	Convert to a streaming dataframe
verify(x)	Verify consistency of elements that pass through this stream
window([n, value])	Sliding window operations

aggregate	
astype	
emit	
map	
query	

accumulate_partitions (*func*, **args*, ***kwargs*)
Accumulate a function with state across batch elements

See also:

`Streaming.map_partitions`

assign (***kwargs*)
Assign new columns to this dataframe

Alternatively use setitem syntax

Examples

```
>>> sdf = sdf.assign(z=sdf.x + sdf.y)
>>> sdf['z'] = sdf.x + sdf.y
```

count ()
Count of frame

cummax ()
Cumulative maximum

cummin()
Cumulative minimum

cumprod()
Cumulative product

cumsum()
Cumulative sum

groupby(*other*)
Groupby aggreagtions

static map_partitions(*func*, **args*, ***kwargs*)
Map a function across all batch elements of this stream

The output stream type will be determined by the action of that function on the example

See also:

`Streaming.accumulate_partitions`

mean()
Average

reset_index()
Reset Index

rolling(*window*, *min_periods=1*)
Compute rolling aggregations

When followed by an aggregation method like `sum`, `mean`, or `std` this produces a new Streaming dataframe whose values are aggregated over that window.

The window parameter can be either a number of rows or a timedelta like ‘‘”2 minutes”‘ in which case the index should be a datetime index.

This operates by keeping enough of a backlog of records to maintain an accurate stream. It performs a copy at every added dataframe. Because of this it may be slow if the rolling window is much larger than the average stream element.

Parameters

window: int or timedelta Window over which to roll

Returns

Rolling object

See also:

DataFrame.window more generic window operations

round(*decimals=0*)
Round elements in frame

set_index(*index*, ***kwargs*)
Set Index

size
size of frame

sum()
Sum frame

tail(*n=5*)
Round elements in frame

to_frame()
Convert to a streaming dataframe

verify(*x*)
Verify consistency of elements that pass through this stream

window(*n=None*, *value=None*)
Sliding window operations

Windowed operations are defined over a sliding window of data, either with a fixed number of elements:

```
>>> df.window(n=10).sum()  # sum of the last ten elements
```

or over an index value range (index must be monotonic):

```
>>> df.window(value='2h').mean()  # average over the last two hours
```

Windowed dataframes support all normal arithmetic, aggregations, and groupby-aggregations.

See also:

DataFrame.rolling mimic's Pandas rolling aggregations

Examples

```
>>> df.window(n=10).std()
>>> df.window(value='2h').count()
```

```
>>> w = df.window(n=100)
>>> w.groupby(w.name).amount.sum()
>>> w.groupby(w.x % 10).y.var()
```

class streamz.dataframe.**Rolling**(*sdf*, *window*, *min_periods*)
Rolling aggregations

This intermediate class enables rolling aggregations across either a fixed number of rows or a time window.

Examples

```
>>> sdf.rolling(10).x.mean()
>>> sdf.rolling('100ms').x.mean()
```

Methods

aggregate(*args, **kwargs)	Rolling aggregation
count(*args, **kwargs)	Rolling count
max()	Rolling maximum
mean()	Rolling mean
median()	Rolling median

Continued on next page

Table 25 – continued from previous page

min()	Rolling minimum
quantile(*args, **kwargs)	Rolling quantile
std(*args, **kwargs)	Rolling standard deviation
sum()	Rolling sum
var(*args, **kwargs)	Rolling variance

aggregate(**args*, ***kwargs*)
Rolling aggregation

count(**args*, ***kwargs*)
Rolling count

max()
Rolling maximum

mean()
Rolling mean

median()
Rolling median

min()
Rolling minimum

quantile(**args*, ***kwargs*)
Rolling quantile

std(**args*, ***kwargs*)
Rolling standard deviation

sum()
Rolling sum

var(**args*, ***kwargs*)
Rolling variance

class streamz.dataframe.**Window**(*sdf*, *n=None*, *value=None*)
Windowed aggregations

This provides a set of aggregations that can be applied over a sliding window of data.

See also:

DataFrame.window contains full docstring

Attributes

columns

dtypes

example

index

size Number of elements within window

Methods

apply(func)	Apply an arbitrary function over each window of data
count()	Count elements within window
groupby(other)	Groupby-aggregations within window
mean()	Average elements within window
std([ddof])	Compute standard deviation of elements within window
sum()	Sum elements within window
value_counts()	Count groups of elements within window
var([ddof])	Compute variance of elements within window

aggregate	
full	
map_partitions	
reset_index	

apply(*func*)
Apply an arbitrary function over each window of data

count()
Count elements within window

groupby(*other*)
Groupby-aggregations within window

mean()
Average elements within window

size
Number of elements within window

std(*ddof=1*)
Compute standard deviation of elements within window

sum()
Sum elements within window

value_counts()
Count groups of elements within window

var(*ddof=1*)
Compute variance of elements within window

class streamz.dataframe.**GroupBy**(*root*, *grouper*, *index=None*)
Groupby aggregations on streaming dataframes

Methods

count()	Groupby-count
mean()	Groupby-mean
size()	Groupby-size
std([ddof])	Groupby-std
sum()	Groupby-sum

Continued on next page

Table 27 – continued from previous page

var([ddof])	Groupby-variance

count()
Groupby-count

mean()
Groupby-mean

size()
Groupby-size

std(*ddof=1*)
Groupby-std

sum()
Groupby-sum

var(*ddof=1*)
Groupby-variance

class streamz.dataframe.**Random**(*freq='100ms'*, *interval='500ms'*, *dask=False*)
A streaming dataframe of random data

The x column is uniformly distributed. The y column is poisson distributed. The z column is normally distributed.

This class is experimental and will likely be removed in the future

Parameters

freq: timedelta The time interval between records

interval: timedelta The time interval between new dataframes, should be significantly larger than freq

Attributes

columns

dtypes

index

size size of frame

Methods

accumulate_partitions(func, *args, **kwargs)	Accumulate a function with state across batch elements
assign(**kwargs)	Assign new columns to this dataframe
count()	Count of frame
cummax()	Cumulative maximum
cummin()	Cumulative minimum
cumprod()	Cumulative product
cumsum()	Cumulative sum
groupby(other)	Groupby aggreagtions
map_partitions(func, *args, **kwargs)	Map a function across all batch elements of this stream

Continued on next page

Table 28 – continued from previous page

mean()	Average
reset_index()	Reset Index
rolling(window[, min_periods])	Compute rolling aggregations
round([decimals])	Round elements in frame
set_index(index, **kwargs)	Set Index
sum()	Sum frame
tail([n])	Round elements in frame
to_frame()	Convert to a streaming dataframe
verify(x)	Verify consistency of elements that pass through this stream
window([n, value])	Sliding window operations

aggregate	
astype	
emit	
map	
query	
stop	

2.7 Asynchronous Computation

This section is only relevant if you want to use time-based functionality. If you are only using operations like map and accumulate then you can safely skip this section.

When using time-based flow control like `rate_limit`, `delay`, or `timed_window` Streamz relies on the Tornado framework for concurrency. This allows us to handle many concurrent operations cheaply and consistently within a single thread. However, this also adds complexity and requires some understanding of asynchronous programming. There are a few different ways to use Streamz with a Tornado event loop.

We give a few examples below that all do the same thing, but with different styles. In each case we use the following toy functions:

```
from tornado import gen
import time

def increment(x):
    """ A blocking increment function

    Simulates a computational function that was not designed to work
    asynchronously
    """
    time.sleep(0.1)
    return x + 1

@gen.coroutine
def write(x):
    """ A non-blocking write function

    Simulates writing to a database asynchronously
    """
    yield gen.sleep(0.2)
    print(x)
```

2.7.1 Within the Event Loop

You may have an application that runs strictly within an event loop.

```
from streamz import Stream
from tornado.ioloop import IOLoop

@gen.coroutine
def f():
    source = Stream(asynchronous=True)  # tell the stream we're working asynchronously
    source.map(increment).rate_limit(0.500).sink(write)

    for x in range(10):
        yield source.emit(x)

IOLoop().run_sync(f)
```

We call Stream with the `asynchronous=True` keyword, informing it that it should expect to operate within an event loop. This ensures that calls to `emit` return Tornado futures rather than block. We wait on results using `yield`.

```
yield source.emit(x)  # waits until the pipeline is ready
```

This would also work with async-await syntax in Python 3

```
from streamz import Stream
from tornado.ioloop import IOLoop

async def f():
    source = Stream(asynchronous=True)  # tell the stream we're working asynchronously
    source.map(increment).rate_limit(0.500).sink(write)

    for x in range(10):
        await source.emit(x)

IOLoop().run_sync(f)
```

2.7.2 Event Loop on a Separate Thread

Sometimes the event loop runs on a separate thread. This is common when you want to support interactive workloads (the user needs their own thread for interaction) or when using Dask (next section).

```
from streamz import Stream

source = Stream(asynchronous=False)  # starts IOLoop in separate thread
source.map(increment).rate_limit('500ms').sink(write)

for x in range(10):
    source.emit(x)
```

In this case we pass `asynchronous=False` to inform the stream that it is expected to perform time-based computation (our write function is a coroutine) but that it should not expect to run in an event loop, and so needs to start its own in a separate thread. Now when we call `source.emit` normally without using `yield` or `await` the emit call blocks, waiting on a coroutine to finish within the IOLoop.

All functions here happen on the IOLoop. This is good for consistency, but can cause other concurrent applications to become unresponsive if your functions (like `increment`) block for long periods of time. You might address this by using Dask (see below) which will offload these computations onto separate threads or processes.

2.7.3 Using Dask

Dask is a parallel computing library that uses Tornado for concurrency and threads for computation. The `DaskStream` object is a drop-in replacement for `Stream` (mostly). Typically we create a Dask client, and then `scatter` a local Stream to become a DaskStream.

```
from dask.distributed import Client
client = Client(processes=False)  # starts thread pool, IOLoop in separate thread

from streamz import Stream
source = Stream()
(source.scatter()        # scatter local elements to cluster, creating a DaskStream
       .map(increment)   # map a function remotely
       .buffer(5)        # allow five futures to stay on the cluster at any time
       .gather()         # bring results back to local process
       .sink(write))     # call write locally

for x in range(10):
    source.emit(x)
```

This operates very much like the synchronous case in terms of coding style (no `@gen.coroutine` or `yield`) but does computations on separate threads. This also provides parallelism and access to a dashboard at http://localhost:8787/status .

2.7.4 Asynchronous Dask

Dask can also operate within an event loop if preferred. Here you can get the non-blocking operation within an event loop while also offloading computations to separate threads.

```
from dask.distributed import Client
from tornado.ioloop import IOLoop

async def f():
    client = await Client(processes=False, asynchronous=True)
    source = Stream(asynchronous=True)
    source.scatter().map(increment).rate_limit('500ms').gather().sink(write)

    for x in range(10):
        await source.emit(x)

IOLoop().run_sync(f)
```

Index

S

T

U

V

W

Z