

Enabling real-time insights through stream processing in Python



Materials



Point72/csp

Adam Glustein

PyData London: June 15, 2024

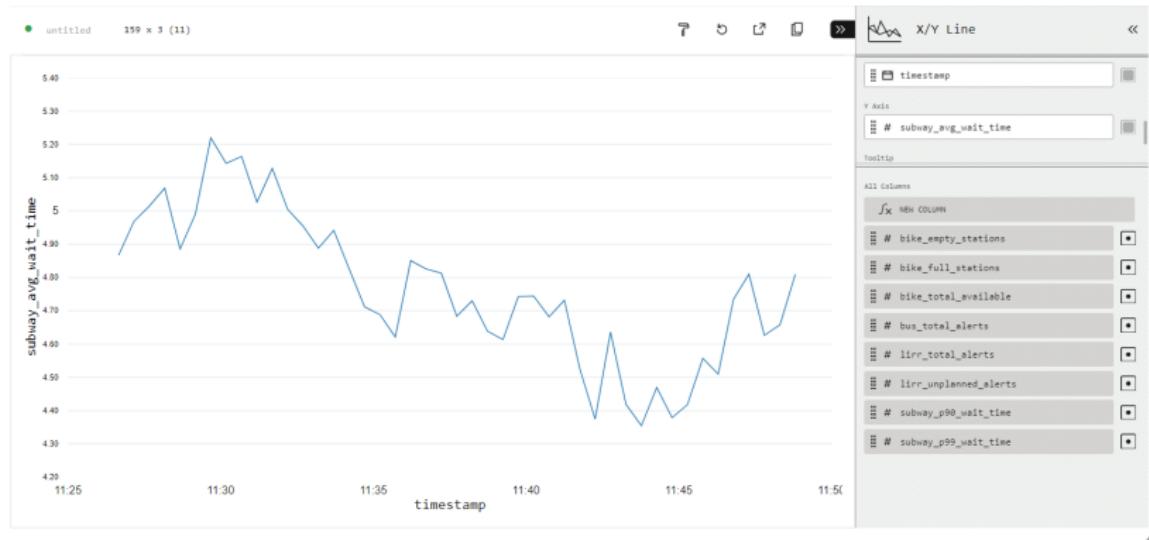
What you will learn in this talk

- what **stream processing** is
- where stream processing can be most useful
- how to create a streaming application using **csp**



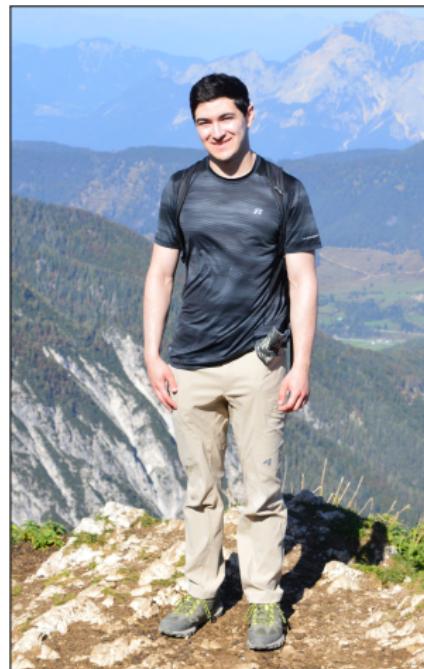
What we will build in this talk

- Live, interactive dashboard for **transit** data



A bit about myself

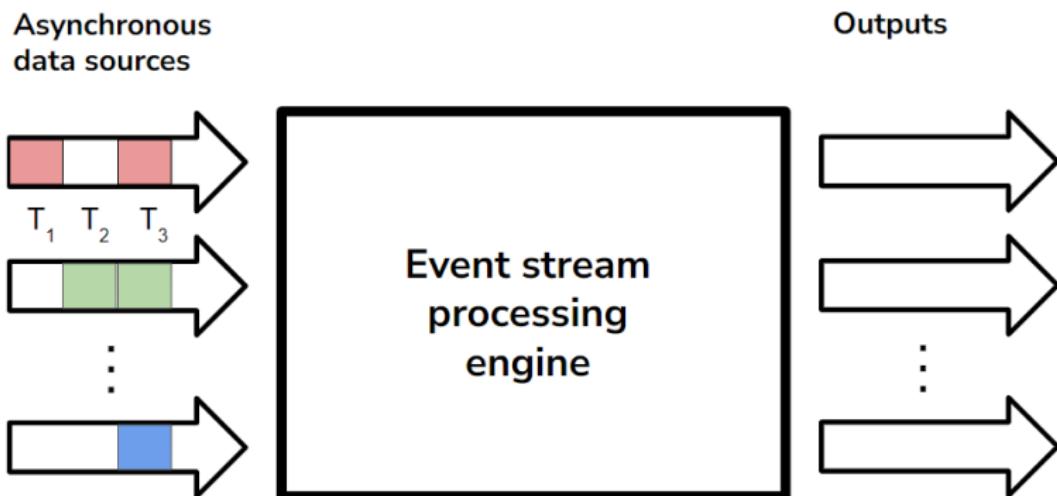
- Quantitative Developer at **Point72** in New York, USA
- Contributor to csp since 2022



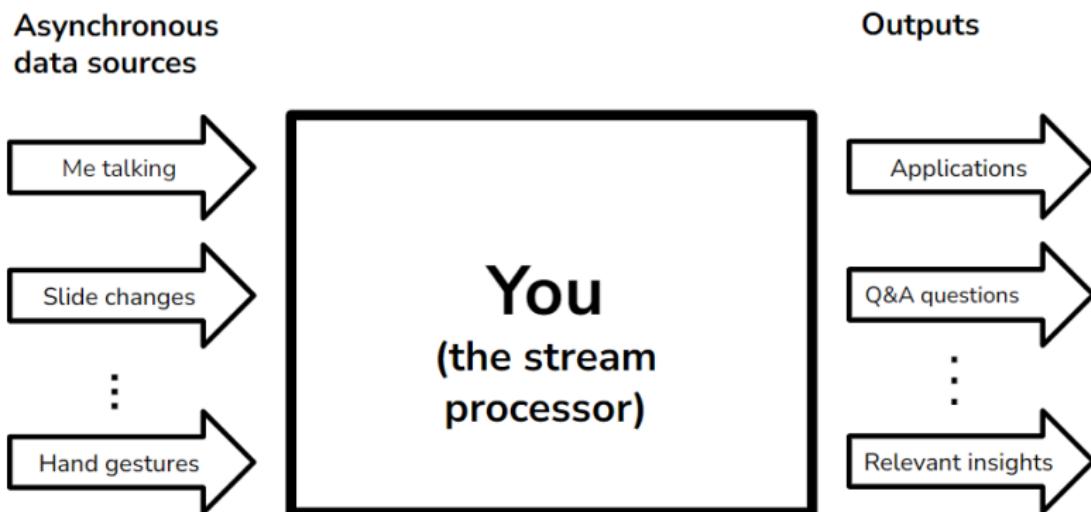
Event stream processing

- React to multiple realtime **events** that occur at unpredictable times
- Humans are naturally good at this!

Event stream processing



Event stream processing



What does a good stream processor need to do?

- ① **Coordinate** multiple input streams
- ② **Transform** data efficiently
- ③ **Convert** outputs to useful data formats

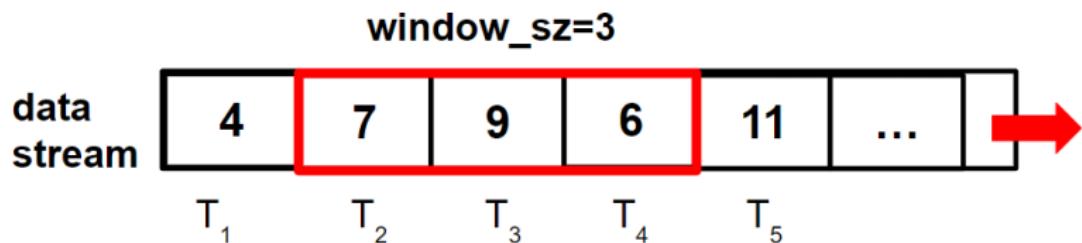
Also needs to work the same in **historical** and **real-time** modes.

Coordinate multiple input feeds

- Key challenges: order and synchronization
 - Internal vs. external timestamps
- Real-time sources: **Kafka**, web endpoints...
- Historical sources: **Parquet**, **polars**...

Transform data efficiently

- Aggregations, filtering, sampling
- **Rolling window** statistics (i.e moving averages)



- Bespoke logic (i.e. calling a pre-trained model)

Convert to useful output

- Same idea as ingesting useful input
- Ensure **predictive** output is timely

Current stream processing ecosystem



bytewax

- Efficient Rust implementation
- Wide support for data formats (*connectors*)

Current stream processing ecosystem



- Stateful, fault-tolerant
- *Watermarks*: complex but highly flexible

Current stream processing ecosystem



- Great for highly parallel tasks
- Multi-language support (Python, Java, Go, etc.)

Current stream processing ecosystem



Pure Python and still widely used



Can be used for simple pipelines or integrated with another framework



Strengths

- Easy to get started
- No infrastructure constraints
- Seamless historical to real-time transition
- Efficient C++ runtime

Current Limitations

- No parallel execution
- Python API only

Brief walk-through of CSP

I said stream processors need to do three things:

- **Coordinate** input data streams - **input adapters**
- **Transform** data - **csp nodes**
- **Convert** to useful output - **output adapters**

Put these together and you have a **csp graph**

Hello World in CSP

```
1 @csp.graph
2 def hello_world():
3     hw = csp.const("hElLo wOrLd") # input adapter
4     low = csp.apply(hw, lambda x: x.title(), str) # node
5     csp.print("Message", low) # output adapter
6
7 csp.show_graph(hello_world)
```



Hello World in CSP

```
1 from datetime import datetime, timedelta  
2  
3 csp.run(hello_world, starttime=datetime.utcnow(),  
4         endtime=timedelta(seconds=1), realtime=True)
```

Output

```
1 2024-05-27 13:10:33.160102 Message:Hello World
```

We can also run in the past

- CSP keeps its own internal *engine time*
- Seamless transition to playback mode

```
1 csp.run(hello_world, starttime=datetime(2022,1,1),  
2         endtime=timedelta(seconds=1), realtime=False)
```

Output

```
1 2022-01-01 00:00:00 Message:Hello World
```

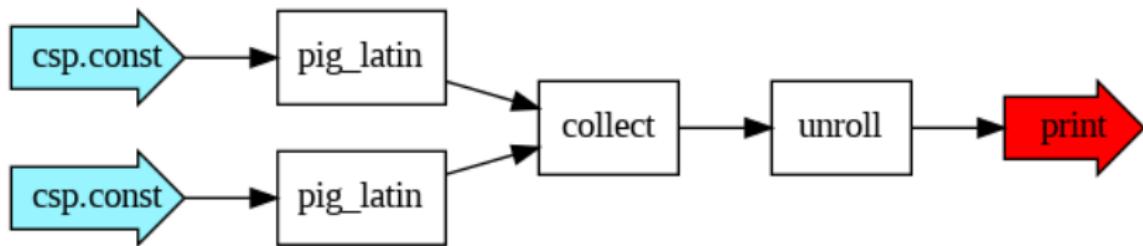
Writing your own CSP nodes

- Nodes transform **time series** data

```
1 from csp import ts
2
3 @csp.node
4 def pig_latin(text: ts[str]) -> ts[str]:
5     is_vowel = lambda c : c in "aeiou"
6     assert len(text) > 2
7     if not is_vowel(text[0]):
8         if is_vowel(text[1]):
9             pl = text[1:]+text[0]
10        else:
11            pl = text[2:]+text[:2]
12        else:
13            pl = text+"w"
14
15    return pl+"ay"
```

Writing your own CSP nodes

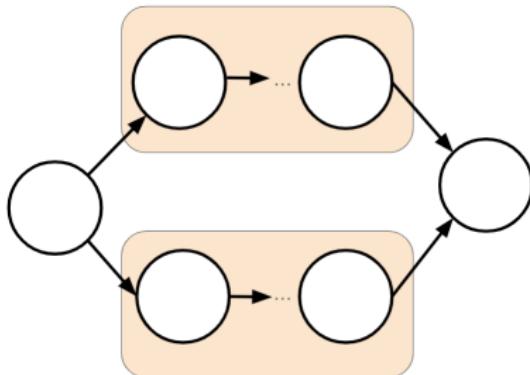
```
1 @csp.graph
2 def ellohay_orldway():
3     hello = csp.const("hello")
4     world = csp.const("world")
5     latin = [pig_latin(hello), pig_latin(world)]
6     csp.print("Pig Latin", csp.flatten(latin))
7
8 csp.show_graph(ellohay_orldway)
```



Nodes and graphs

- Nodes are **atomic** units of computation
- Graphs are collections of nodes

```
1 @csp.graph
2 def subgraph(x: ts[T]):
3     ...
4
5 @csp.graph
6 def main_graph():
7     for source in
8         data_sources:
9             subgraph(source)
10            ...
```



Nodes and graphs

Nodes can maintain a per-instance **state** and schedule internal events

```
1 @csp.node
2 def poisson_counter(lam: float) -> ts[int]:
3     with csp.alarms(): # An internal time-series
4         a = csp.alarm(int)
5     with csp.state(): # State stored per node instance
6         s_cnt = 0
7     with csp.start(): # Block runs once at graph start
8         delay = np.random.exponential(lam)
9         csp.schedule_alarm(a, timedelta(seconds=delay), 1)
10
11    if csp.ticked(a):
12        s_cnt += 1
13        new_delay = np.random.exponential(lam)
14        csp.schedule_alarm(a,
15            timedelta(seconds=new_delay), 1)
16
17    return s_cnt
```

- Historical replay is key for research/testing
- Real deployment uses **live** data
 - Financial markets
 - Transportation networks
 - Climate analysis
 - Many more!

A dashboard for transit methods in New York City

- Real-time feeds available:
 - Bus and commuter rail alerts (from the MTA)
 - Bike share station status (from CitiBike)
 - Subway train positions/delays (from the MTA)
- These are **asynchronous, real-time** data sources

Link to live demo



github.com/AdamGlustein/csp-pydata-london-2024/

Summary

- Stream processing is a powerful technique for gaining insights from **live** data
- Common problem: moving from historical to real-time execution
- **Composable stream processing** (csp) allows for a seamless transition

Q & A



Presentation materials



CSP Project