Straightforward stream processing with CSP

Materials

Adam Glustein

PyData NYC

Point72/csp

November 7th, 2024

What you will learn

- Basics of stream processing
- Challenges posed by **real-time** data pipelines
  - Debugging and unit testing
  - Performance tuning
  - Interactive execution (ipython)
- How **CSP** removes some of these obstacles

The main goal of this talk

- Add a new tool to your data science toolbox

The law of the instrument (corollary)

If the only tool you have is *pandas*, then all data looks like a *DataFrame*.

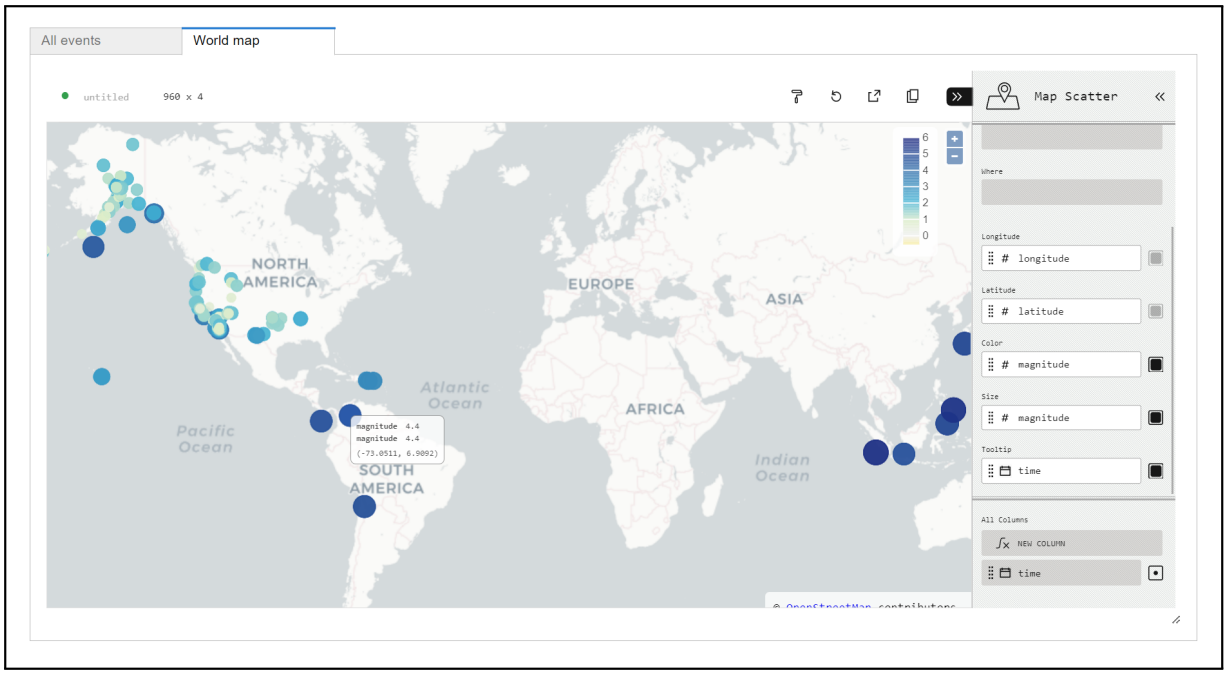
- Know when to identify situations where stream processing makes sense for you

A bit about myself

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

Composable Stream Processing (CSP)

- First open-source project from Point72
- Many internal (and now external!) contributors
- Primary goal: improve **developer experience** when building real-time data applications



What is developer experience?

Definition

(Developer Experience) How easy it is to build a robust application using a tool or framework

- Inherently subjective!
- Common criteria that *most* people agree on
  1. How much code do I have to write?
  2. How certain can I be in my app's behavior?
  3. How easy it is to integrate with other tools?

Static vs. dynamic data

- Two different types of *data*

Static Data

- All entries known up-front

Dynamic Data

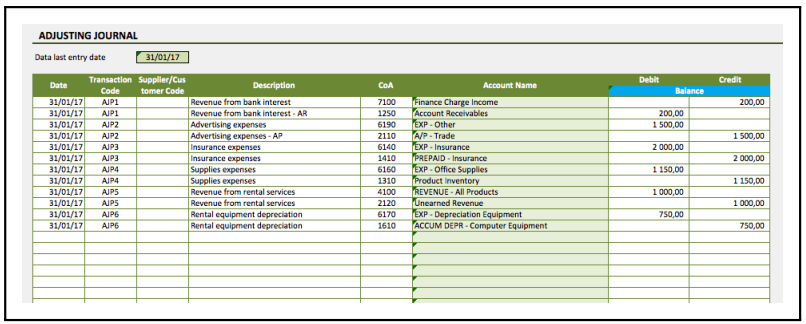
- Data updates during application runtime

Batch vs. stream processing

- Two different methods of *data processing*

Batch Processing

- Periodically process *batches* of data
- Simpler and less compute



ADJUSTING JOURNAL

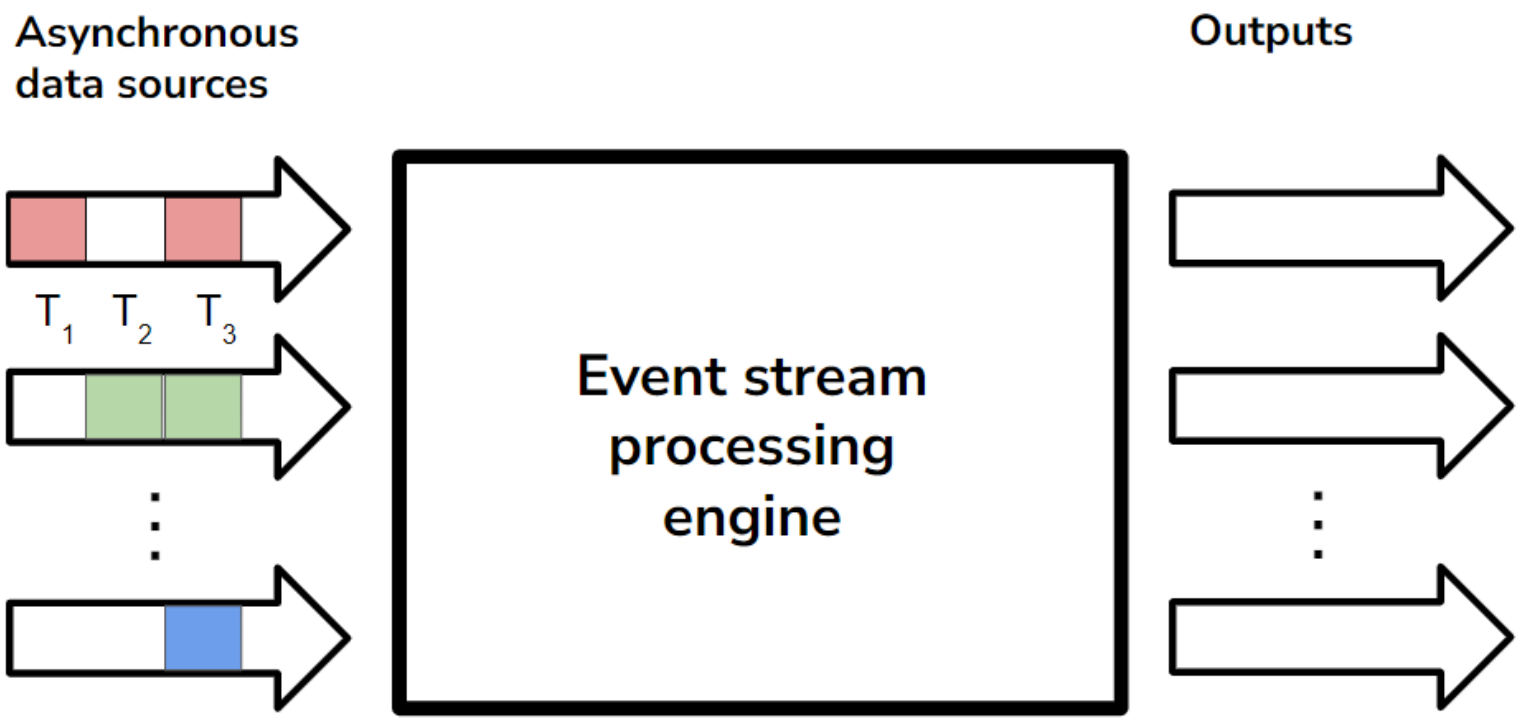
Data last entry date 31/01/17

Date	Transaction Code	Supplier/Customer Code	Description	CoA	Account Name	Debit	Credit	Balance
31/01/17	AJP1		Revenue from bank interest	7100	Finance Charge Income		200,00	
31/01/17	AJP1		Revenue from bank interest - AR	1250	Account Receivables	200,00		
31/01/17	AJP2		Advertising expenses	6190	EXP - Other	1 500,00		
31/01/17	AJP2		Advertising expenses - AP	2110	A/P - Trade		1 500,00	
31/01/17	AJP3		Insurance expenses	6140	EXP - Insurance	2 000,00		
31/01/17	AJP3		Insurance expenses	1410	PREPAID - Insurance		2 000,00	
31/01/17	AJP4		Supplies expenses	6160	EXP - Office Supplies	1 150,00		
31/01/17	AJP4		Supplies expenses	1310	Product Inventory		1 150,00	
31/01/17	AJP5		Revenue from rental services	4100	REVENUE - All Products	1 000,00		
31/01/17	AJP5		Revenue from rental services	2120	Unearned Revenue		1 000,00	
31/01/17	AJP6		Rental equipment depreciation	6170	EXP - Depreciation Equipment	750,00		
31/01/17	AJP6		Rental equipment depreciation	1610	ACCUM DEPR - Computer Equipment		750,00	

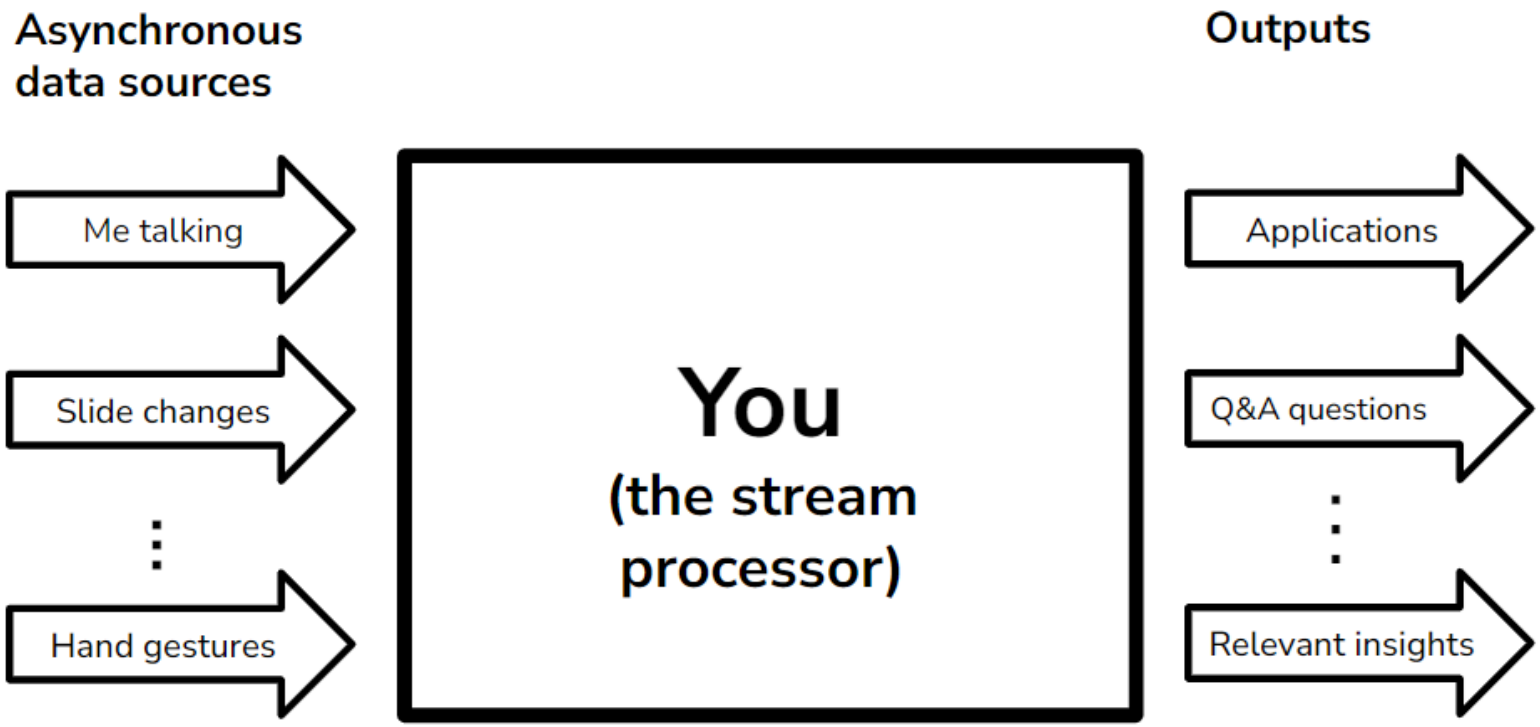
Stream Processing

- Process data immediately
- Provides immediate results

Stream processing



Stream processing



A semi-complete data toolbox

- Well-known tools for static data


- Distributed batch processing tools

- Unified (batch and stream) platforms

- Pure stream processing tools

What about CSP?

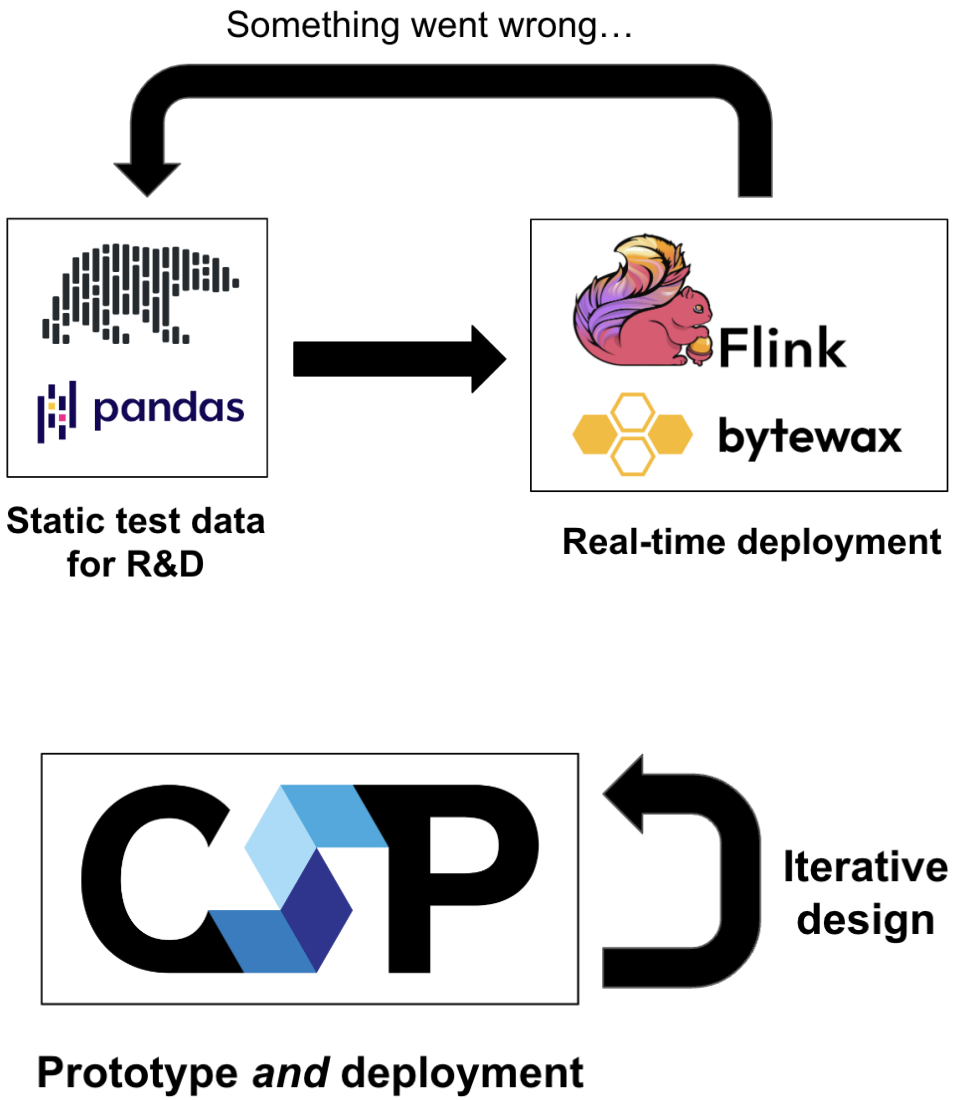
 is a stream processing library for *both* static and dynamic data

Key feature: seamless transition between historical and real-time workflows

Why CSP integrates static dataflows

- Streaming data is inherently dynamic
- However, static data is much easier to work with
  - Quick prototyping
  - Backtesting on historical data
  - Verification before deployment
  - Stable CI tests and benchmarks

The CSP development cycle



Why use a different framework for research?

- Static data libraries have a **great** developer experience!
- Streaming data libraries are focused on deployment, not development

Example 1: Reading a CSV file

pyflink

```
t_env.create_temporary_table(
    "source",
    TableDescriptor.for_connector("filesystem")
    .schema(Schema.new_builder()
    .column("word", DataTypes.STRING())
    .build())
    .option("path", input_path)
    .format("csv")
    .build())
tab = t_env.from_path("source")
```

pandas

```
df = pd.read_csv(input_path)
```

Example 2: Writing to stdout

pyflink

```
t_env.create_temporary_table(
    "sink",
    TableDescriptor.for_connector("print")
    .schema(Schema.new_builder()
    .column("word", DataTypes.STRING())
    .column("count", DataTypes.BIGINT())
    .build())
    .build())
```

pandas

```
print(df)
```

CSP's solution

- Make a *streaming data* library with a similar developer experience to a *static data* library

Reading a CSV file

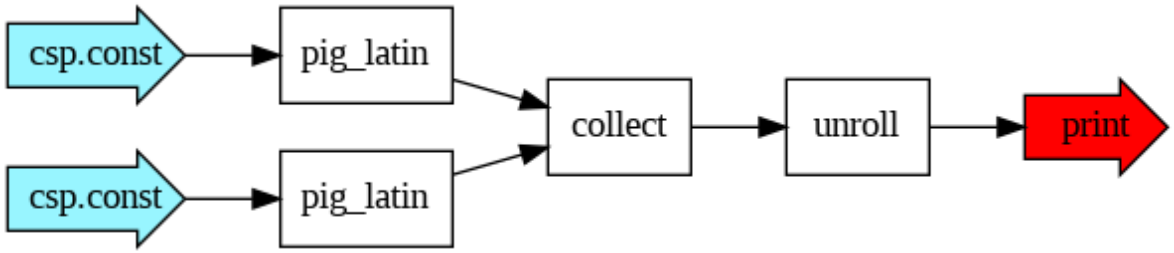
```
reader = CSVReader("data.csv", time_format)
stream = reader.subscribe_all(str)
```

Writing to stdout

```
csp.print(stream)
```

Brief walkthrough of CSP

- CSP is a **computation graph** framework



1. Input adapters bring data into the graph
2. Nodes transform data
3. Output adapters send data out of the graph

- `csp.graph`: collection of these 3 elements

Hello World in CSP

```
@csp.graph
def hello_world():
  hw = csp.const("hElLo wOrLd") # input adapter
  fmt = csp.apply(hw, lambda x: x.title(), str) # node
  csp.print("Message", fmt) # output adapter

csp.show_graph(hello_world)
```

Hello World in CSP

```
from datetime import datetime, timedelta

csp.run(hello_world, starttime=datetime.utcnow(),
    endtime=timedelta(seconds=1), realtime=True)
```

Output

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

Hello World in CSP

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

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

Output

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

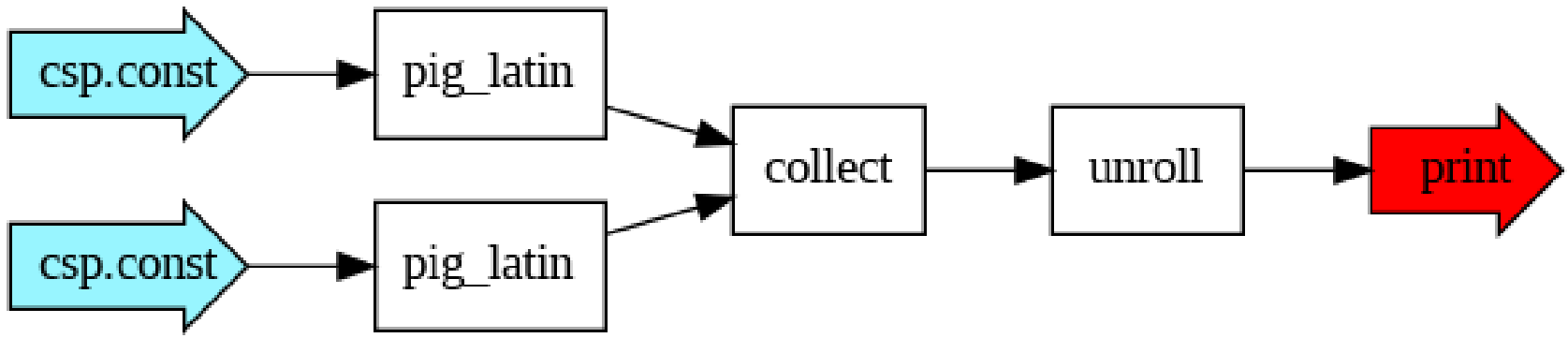
Writing your own CSP nodes

- Nodes transform **time series** data

```
from csp import ts

@csp.node
def pig_latin(text: ts[str]) -> ts[str]:
  is_vowel = lambda c : c in "aeiou"
  assert len(text) > 2
  if not is_vowel(text[0]):
    if is_vowel(text[1]):
      pl = text[1:]+text[0]
    else:
      pl = text[2:]+text[:2]
  else:
    pl = text+"w"

  return pl+"ay"
```

Writing your own CSP nodes

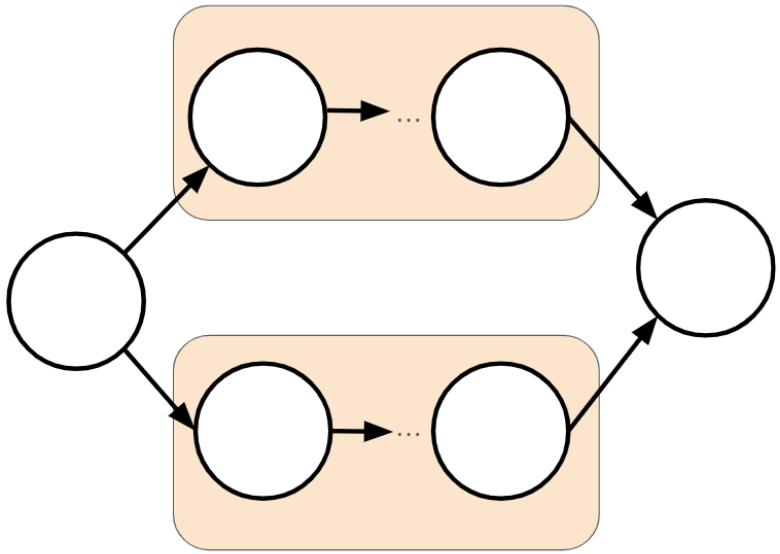
```python
@csp.graph
def ellohay_orldway():
  hello = csp.const("hello")
  world = csp.const("world")
  latin = [pig_latin(hello), pig_latin(world)]
  csp.print("Pig Latin", csp.flatten(latin))

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
      data_sources:
8     subgraph(source)
9     ...
```



Pre-written nodes and adapters

- CSP engine is written in **C++** for performance
- Includes an extensive library of optimized nodes (control flow, statistics etc.)
- Input and output adapters for common formats (Parquet, Kafka, Perspective)

Current limitations of CSP

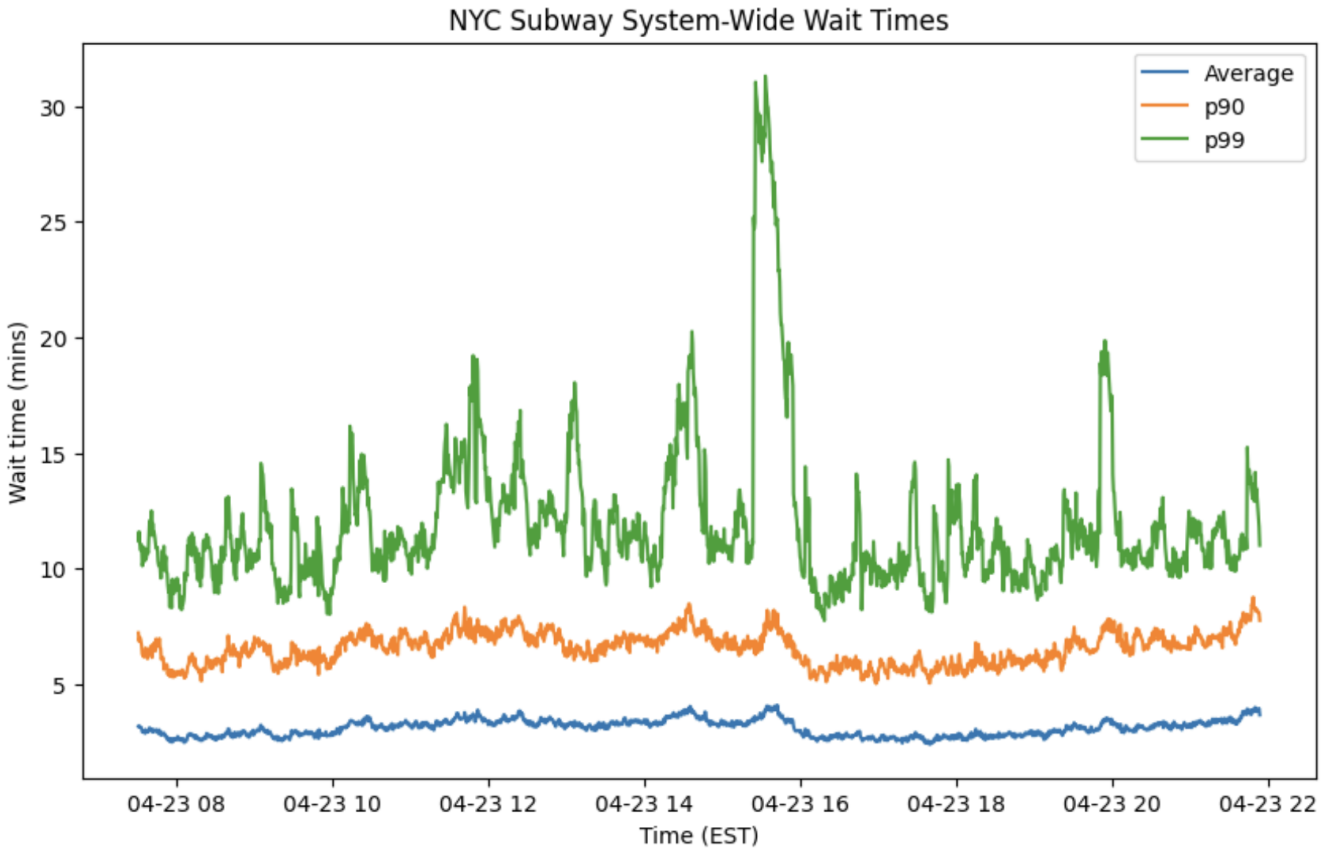
- Fewer pre-built adapters than some other frameworks
- Developer API is only in Python
  - Flink, Beam available in multiple languages
- Can execute graphs in **parallel** but not nodes
  - Node-level parallelization is often too fine grained

Areas where CSP can enhance developer experience

1. Research before deployment
2. Debugging
3. Unit testing
4. Performance tuning
5. Interactive execution (jupyter)

Research before deployment

- Playback **time-series** data for research
- CSP handles event ordering and synchronization



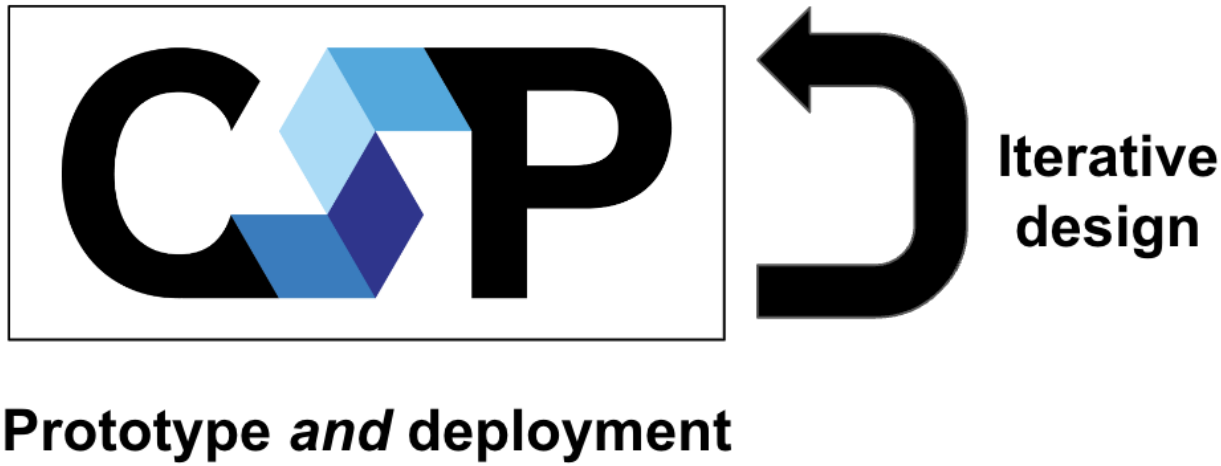
Research before deployment

- **Look-ahead bias:** using future data in historical analysis
- Example: knowing delays in transit route planning
- CSP removes look-ahead bias in research

```
>> python e_01_nyct_subway.py 635:456 L03:L R20:NQRW --num_trains 5

2024-04-20 00:27:24.166629 Departure Board:
 At station 14 St-Union Sq

Uptown L train to 8 Av in 1 minutes
Downtown L train to Canarsie-Rockaway Pkwy in 1 minutes
Uptown L train to 8 Av in 2 minutes
Uptown L train to 8 Av in 5 minutes
Downtown L train to Canarsie-Rockaway Pkwy in 5 minutes
```

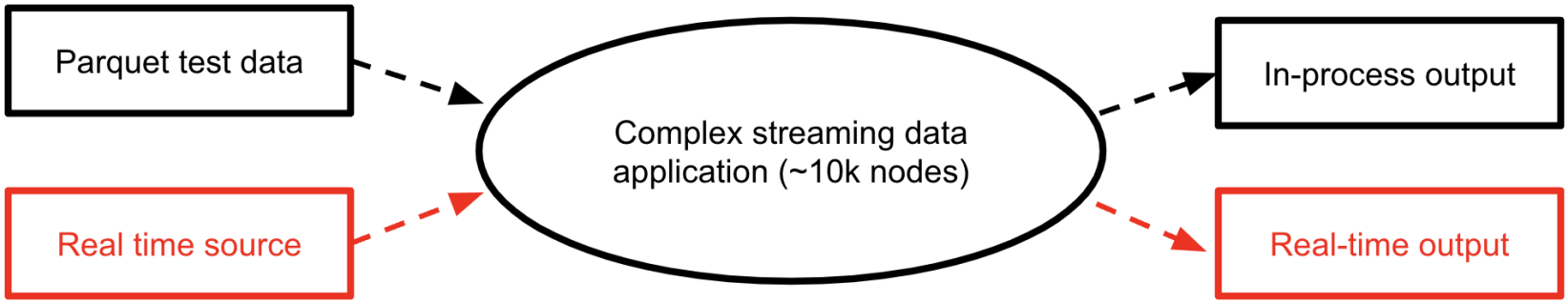
Debugging

- Single application for research and deployment
- Replay event stream to replicate a crash or error



Unit testing

- Real-time data is unstable in CI tests
- Static data allows for more comprehensive tests
- In CSP, simply swap out your adapters



Performance tuning

- *"Premature optimization is the root of all evil"* - Knuth
- CSP provides a **live profiler** to identify bottlenecks in your application
- Requires 1 line of code to configure!

```
with profiler.Profiler(http_port=8888) as p:
    # run the graph normally
    results = csp.run(graph, starttime=st, endtime=et)
```

Performance tuning

- Live timing metrics
- Periodic memory snapshots

Current time: 2022-08-23 17:25:14.048765

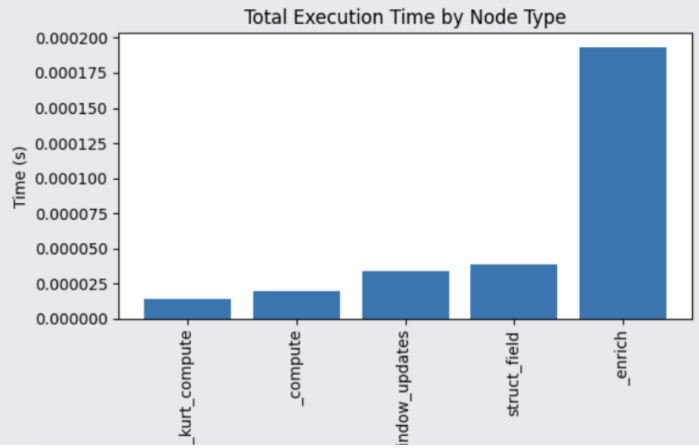
Graph build time: 0.00329 s

Cycle Executions/Times

Number of cycles executed	29.000000
Average cycle execution time (s)	0.000020
Maximum cycle execution time (s)	0.000161
Average graph utilization	0.095000

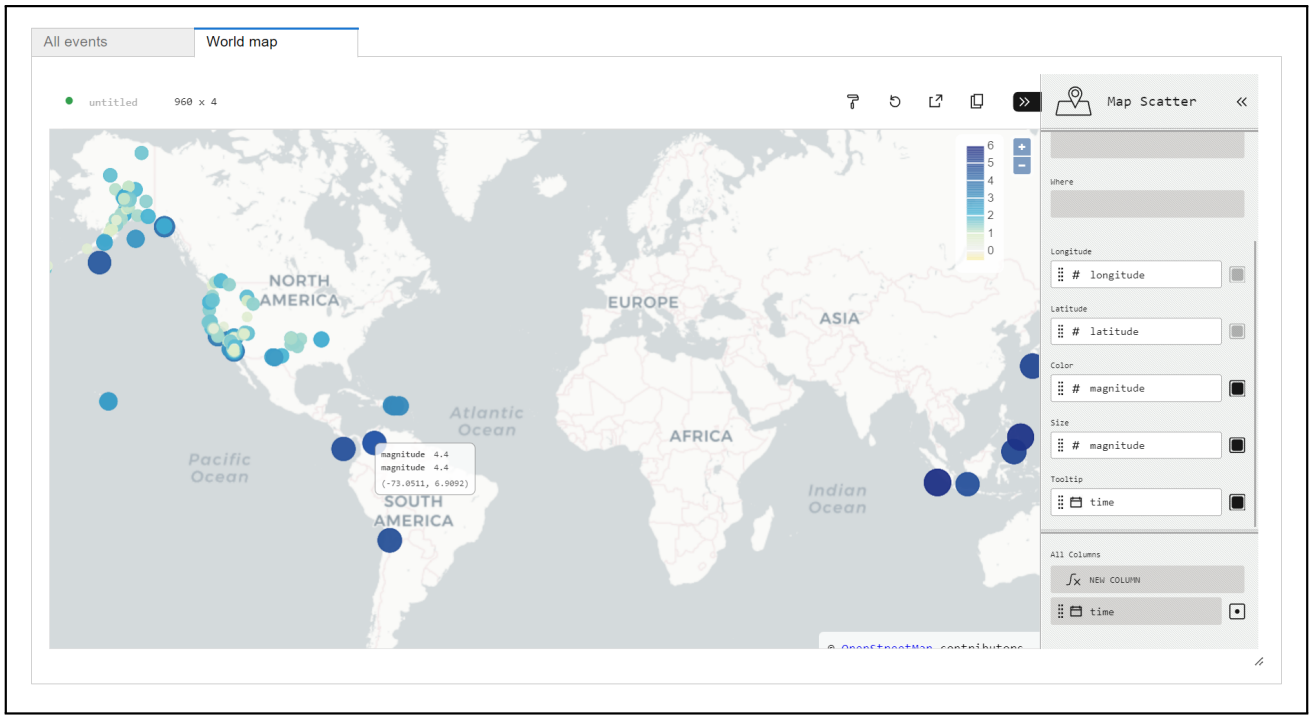
Node Executions/Time

Node Type	Executions	Total Time (s)	Average Time (s)	Max. Time (s)
_enrich	3	0.000194	0.000065	0.000069
struct_field	3	0.000039	0.000013	0.000017
_time_window_updates	3	0.000034	0.000011	0.000012
_compute	6	0.000019	0.000003	0.000005
_skew_kurt_compute	6	0.000014	0.000002	0.000003



Interactive execution

- CSP is fully functional in IPython/Jupyter
- Useful for prototyping and interactive demos



Live earthquake tracker running in a Jupyter notebook

Stream processing is hard, but it doesn't have to be!

- CSP abstracts away the complexity of stream processing
- Over 30 examples at github.com/Point72/csp
- User projects at github.com/csp-community

```
pip install csp
```