Naiad

A Timely Dataflow System

Florian Klein (fbk24), Large-Scale Data Processing and Optimisation (R244)

Naiad: A Timely Dataflow System

Derek G. Murray Frank McSherry Rebecca Isaacs
Michael Isard Paul Barham Martín Abadi
Microsoft Research Silicon Valley
{derekmur,mcsherry,risaacs,misard,pbar,abadi}@microsoft.com

Abstract

Naiad is a distributed system for executing data parallel, cyclic dataflow programs. It offers the high throughput of batch processors, the low latency of stream processors, and the ability to perform iterative and incremental computations. Although existing systems offer some of these features, applications that require all three have relied on multiple platforms, at the expense of efficiency, maintainability, and simplicity. Naiad resolves the complexities of combining these features in one framework.

A new computational model, *timely dataflow*, underlies Naiad and captures opportunities for parallelism across a wide class of algorithms. This model enriches dataflow computation with timestamps that represent logical points in the computation and provide the basis for an efficient, lightweight coordination mechanism.

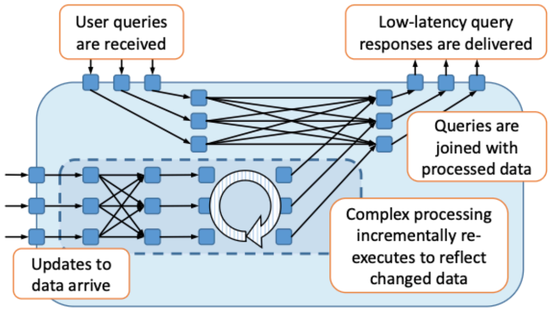
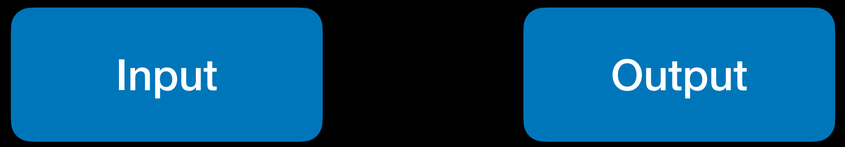


Figure 1: A Naiad application that supports real-time queries on continually updated data. The dashed rectangle represents iterative processing that incrementally updates as new data arrive.

requirements: the application performs iterative process-

Background

Dataflow Programming

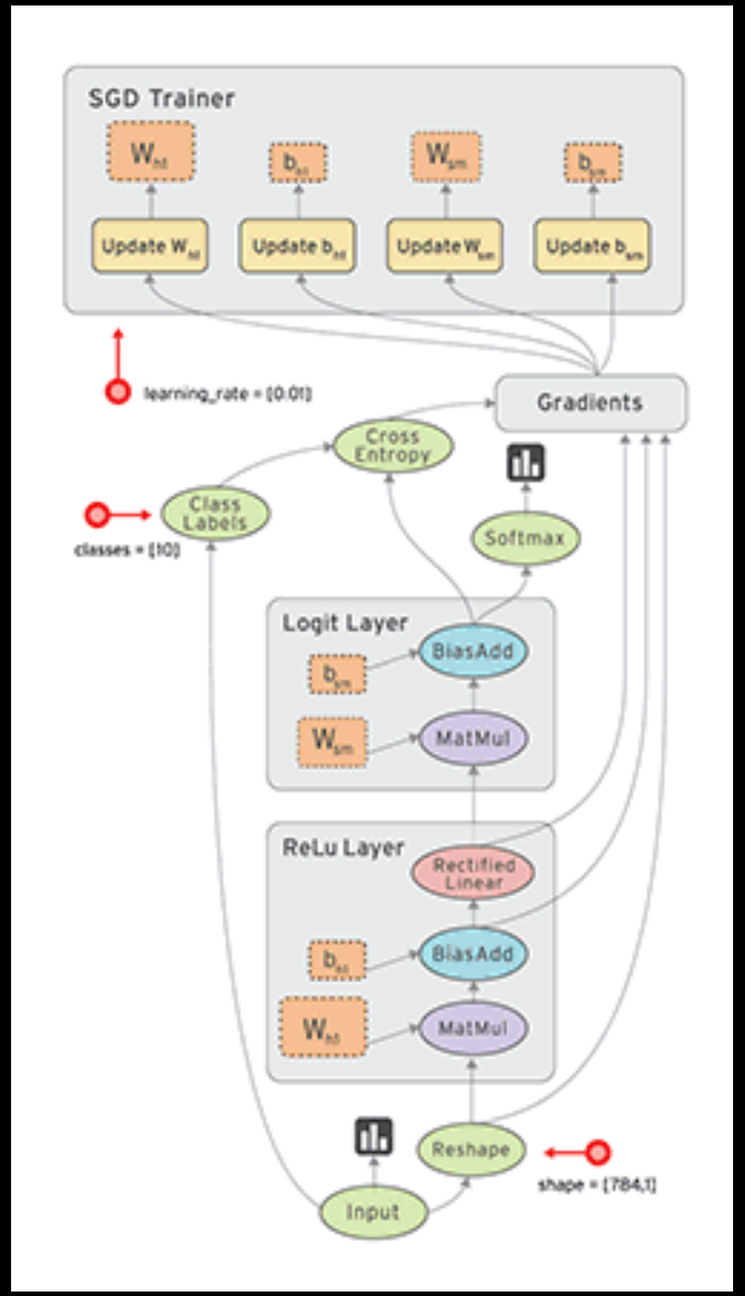


Background

Dataflow Programming

Motivation
Dataflow Programming

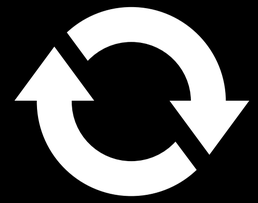
- ML frameworks (e.g. TensorFlow)
- Realtime Analytics
- Network & Packet Processing



[1]

Motivation

Most important metrics



Modern data processing tasks require:

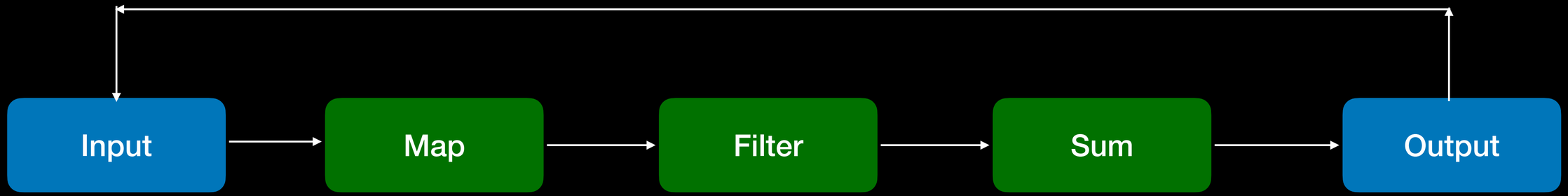
A. **Timeliness:** High Throughput and Low Latency

B. **Consistency:** Correct results even when we have late arrivals

C. **Expressiveness:** Iterative and Incremental Computations

Motivation

Expressiveness and Iterative Computations



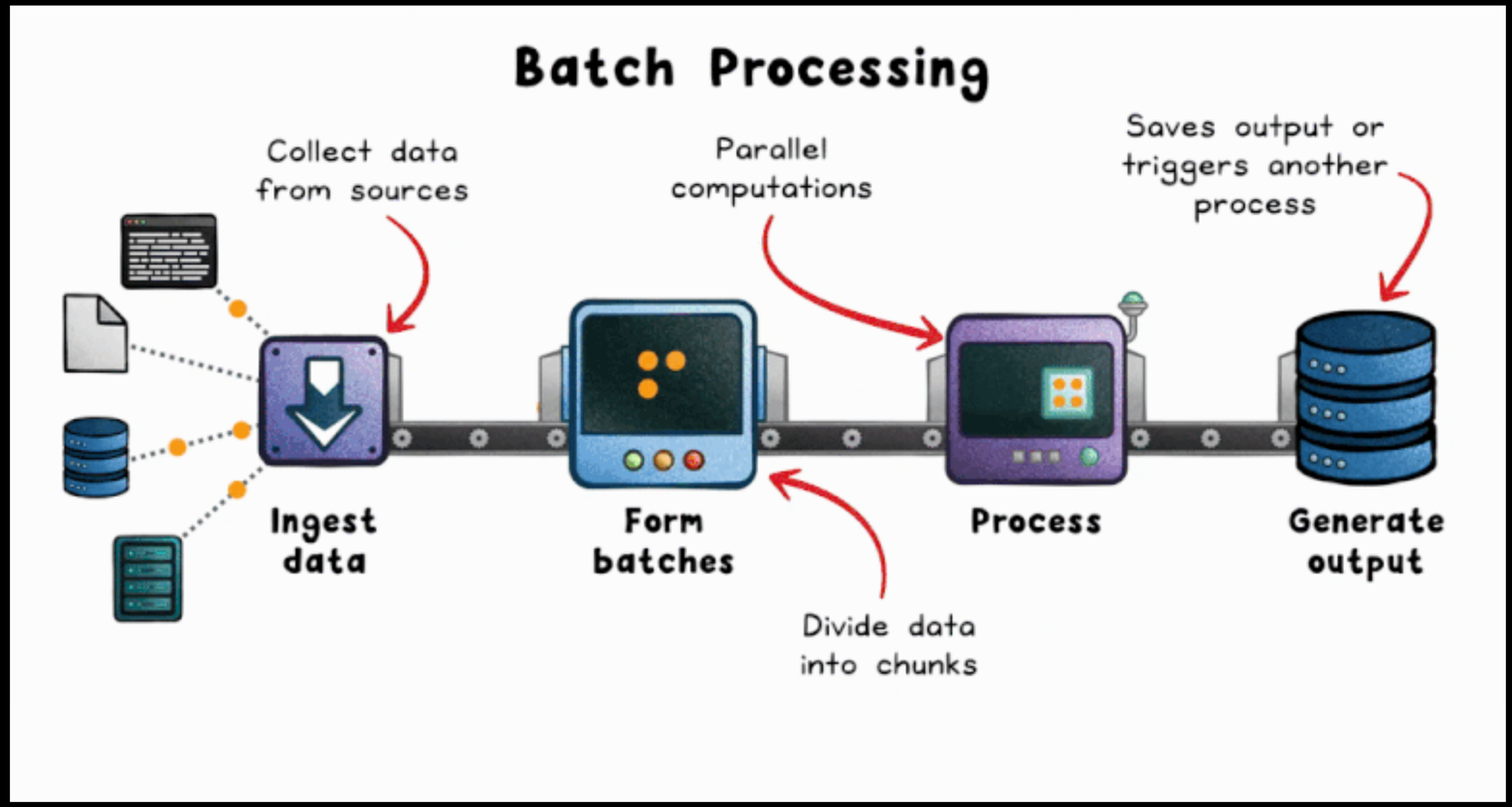
Example Applications:

- PageRank (uses previous scores + new input for next iteration)
- Streaming Statistics (e.g. Rolling Average, …)

Existing Approaches
Batch Processing

- Support synchronous iterations (on one batch)
- Higher Latency

```
let mut iteration = 0;
while !converged {
    // run parallel on all data
    process_batch();
    // wait for all workers to finish
    block_on(global_barrier);
    iteration += 1;
}
```

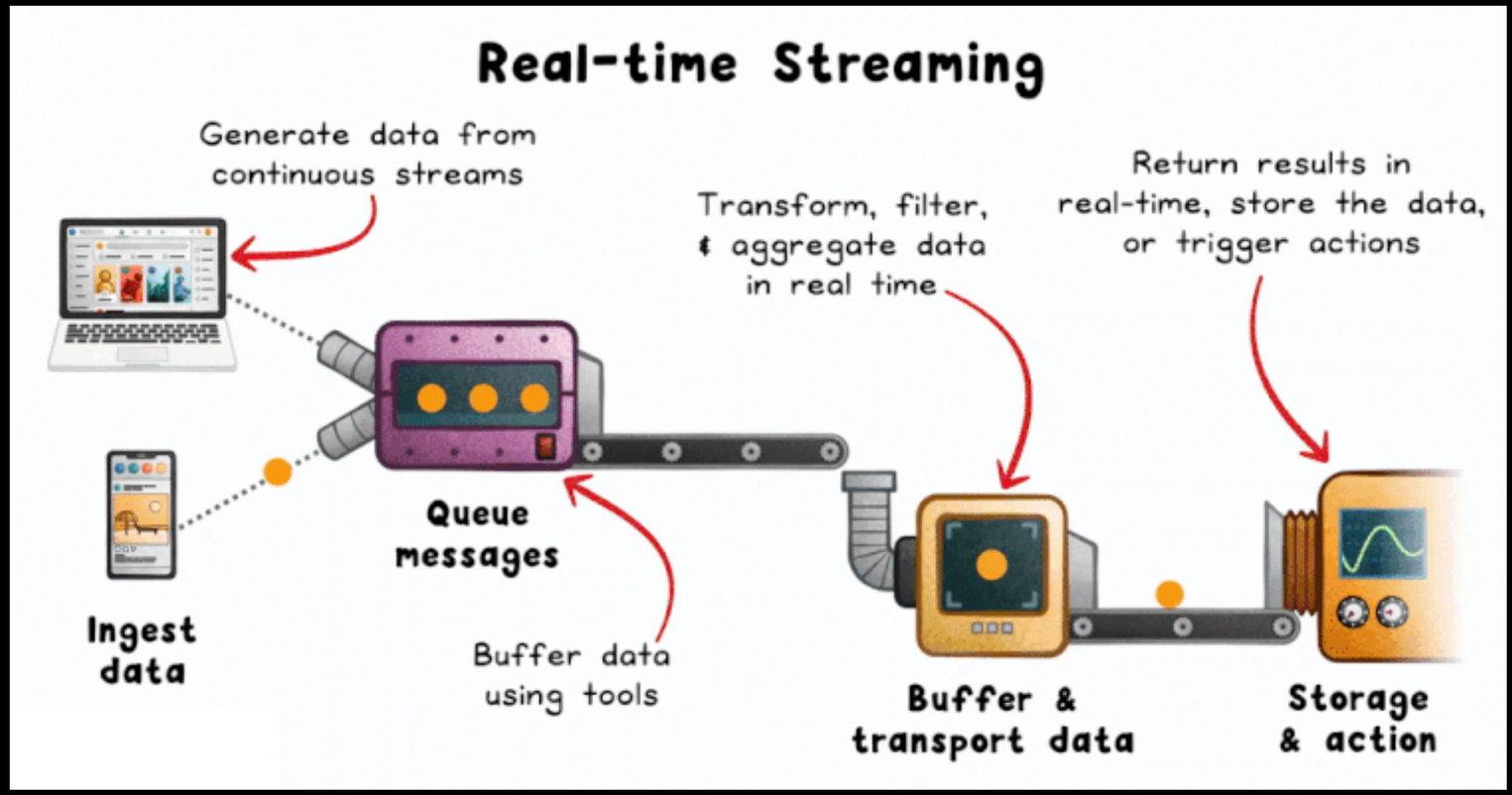


[2]

Existing Approaches
Stream Processing

- Low Latency
- Only non-iterative algorithms

```
while let Some(event) = stream.next() {
    // update incrementally
    process(event);
    emit_partial_results();
    // ...more data may still arrive
}
```



[2]

Existing Approaches

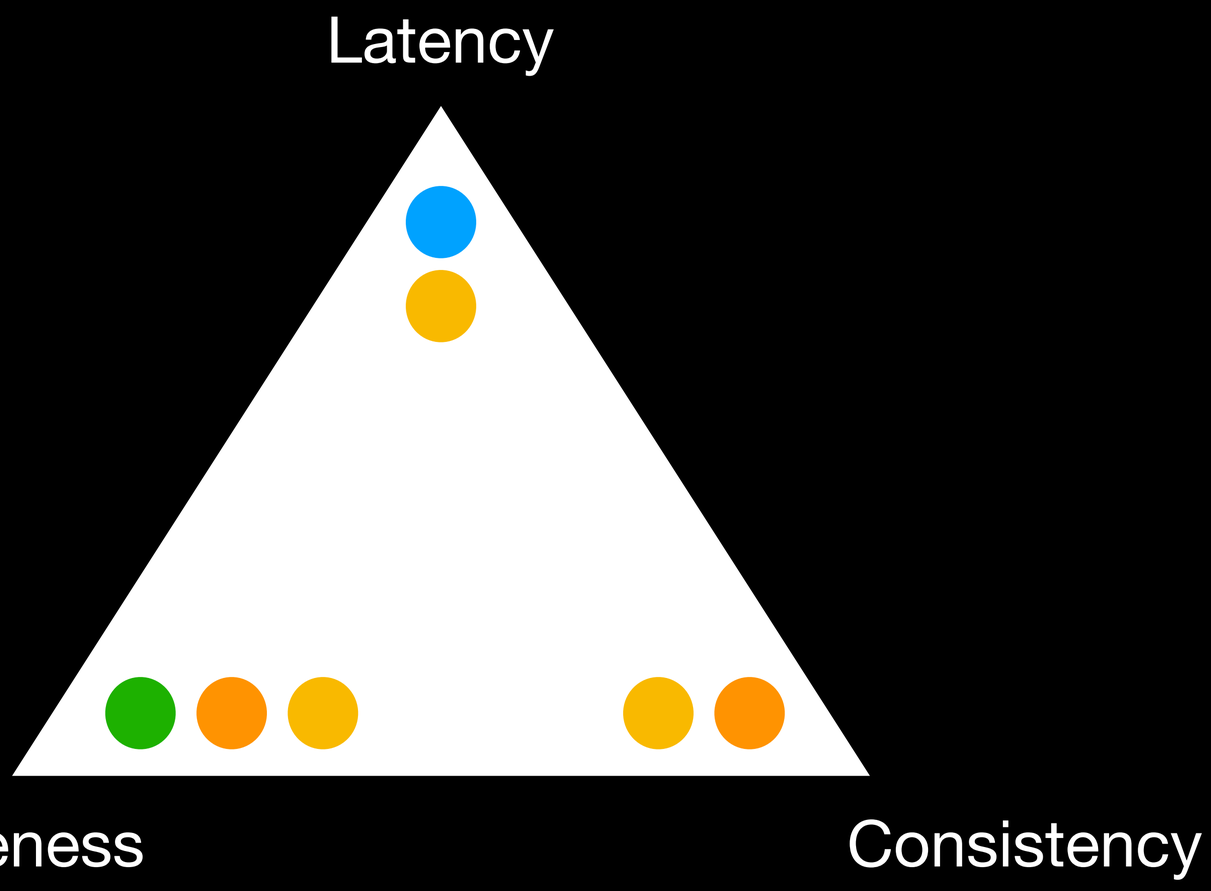
Comparison

Stream Processing

Batch Processing

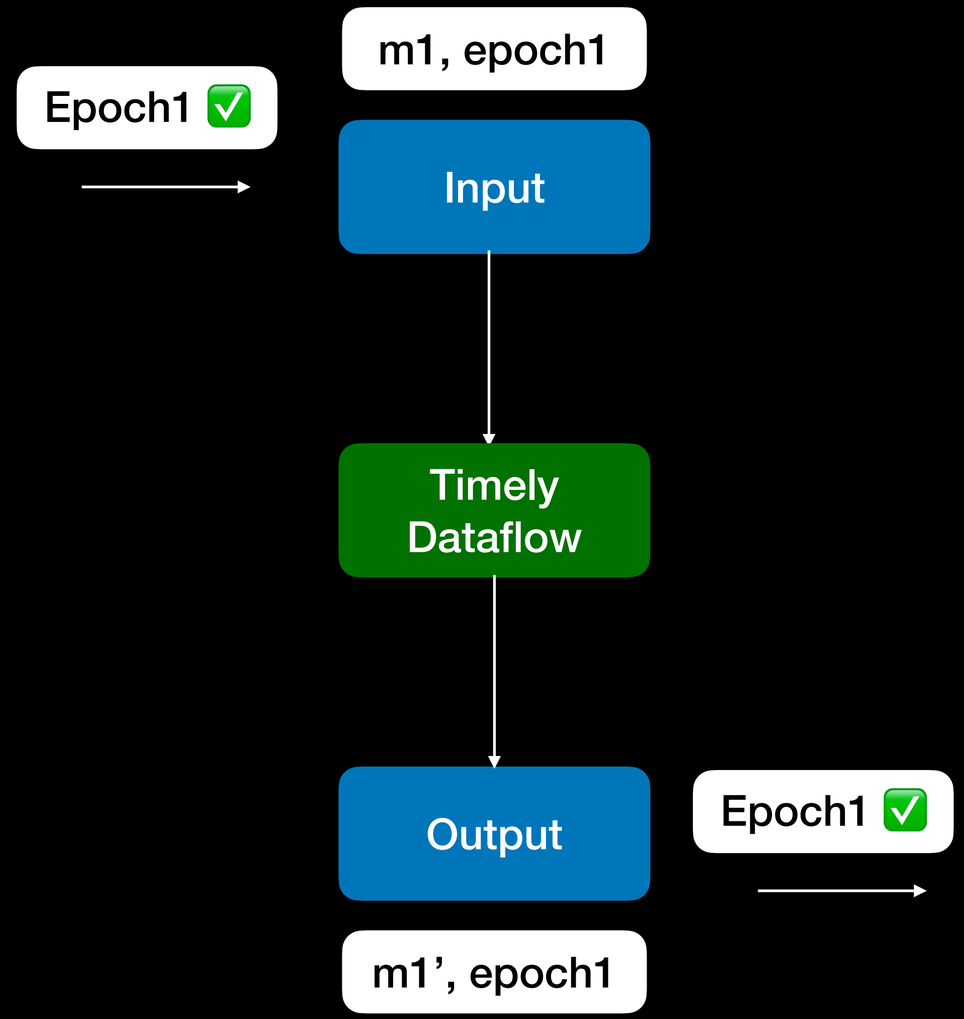
Database

Timely Dataflow



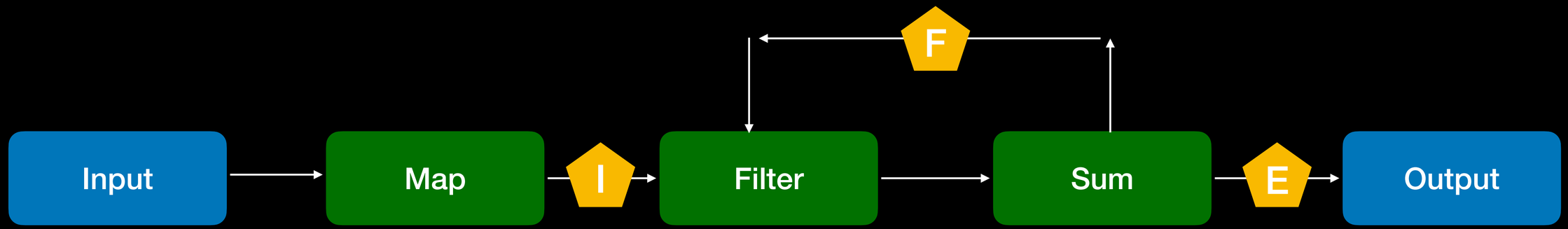
Approach
Outside Perspective

- External Producer labels each message with an integer *epoch*
- Notifies when no more messages from a given *epoch*
- Outputs transformed messages with *epoch* label and also notifies when epoch done

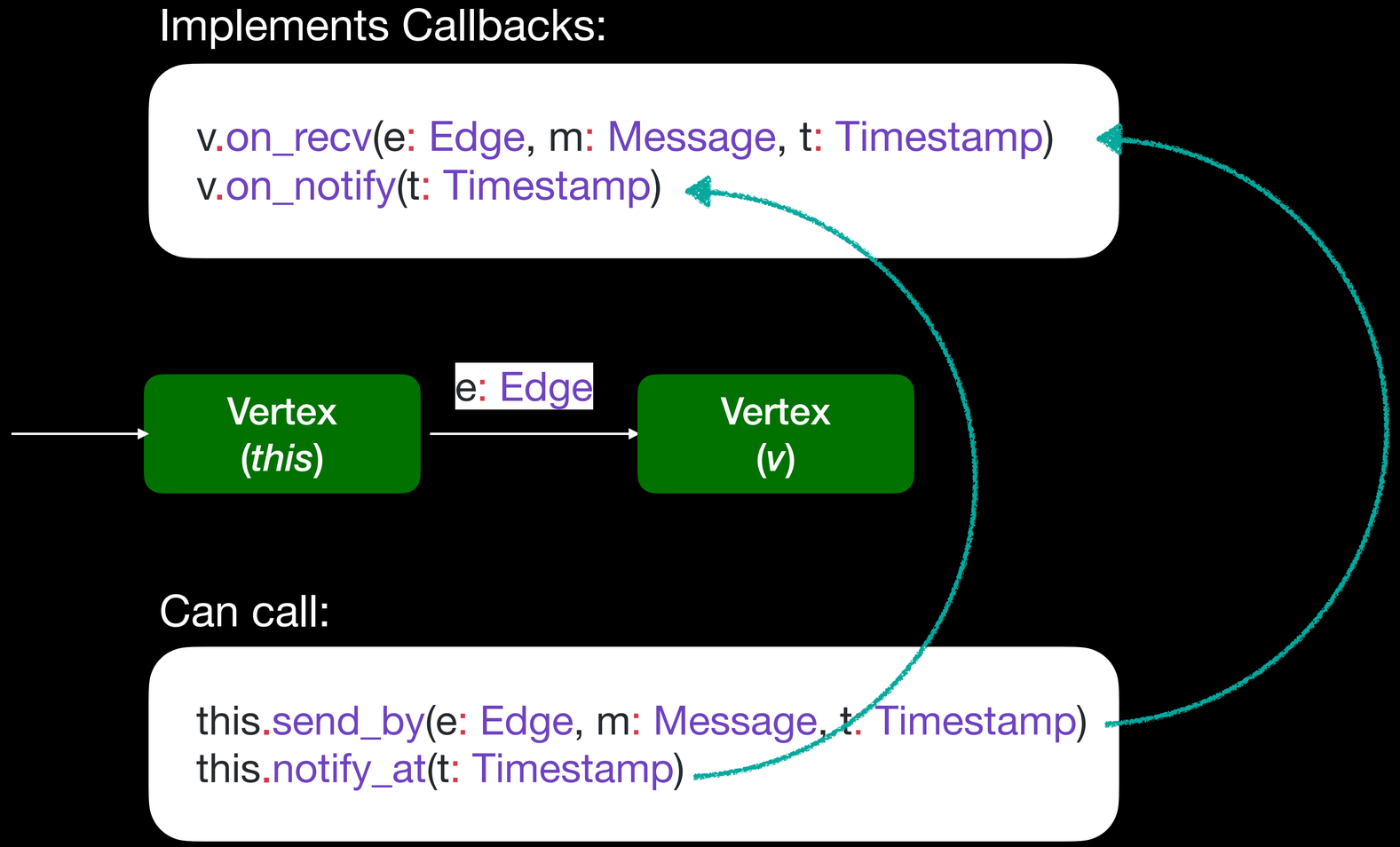


Approach

Timely Dataflow: Overview



Approach
Vertices



Approach
Vertices

- Ordering of send / recv is flexible, BUT:

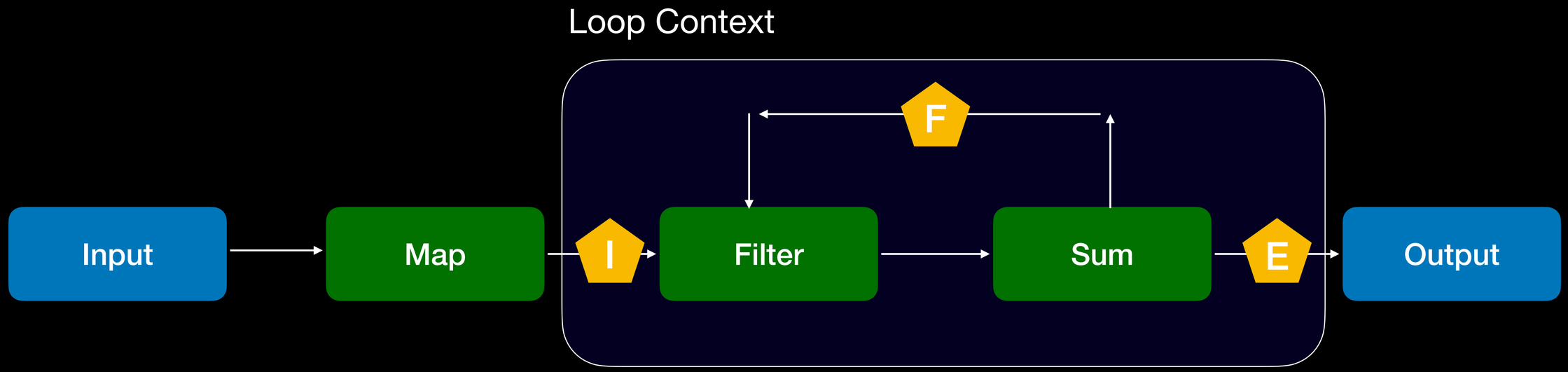
  For t’ <= t:

  v.on_notify(t) only after all v.on_recv(e, m, t’)

- When invoked with t, only invoke send/notify with t’ >= t („not sent backwards)

Approach

Logical Timestamps: Loops



Loops require: Ingress, Egress, and a Feedback Vertex.

We can define:

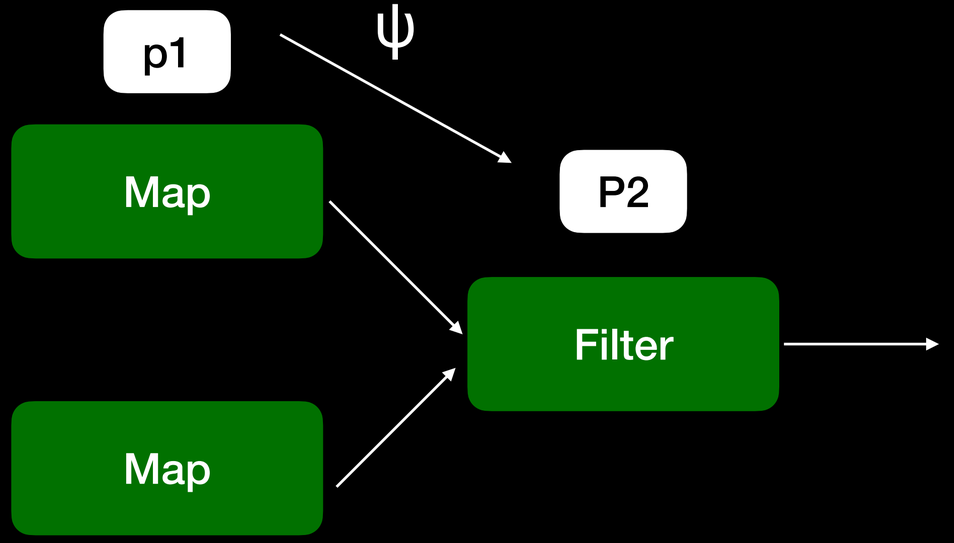
$$\text{Timestamp} : (\overbrace{e \in \mathbb{N}}^{\text{epoch}}, \overbrace{\langle c_1, \ldots, c_k \rangle \in \mathbb{N}^k}^{\text{loop counters}})$$

Vertex	Input timestamp	Output timestamp
Ingress	$(e, \langle c_1, \ldots, c_k \rangle)$	$(e, \langle c_1, \ldots, c_k, 0 \rangle)$
Egress	$(e, \langle c_1, \ldots, c_k, c_{k+1} \rangle)$	$(e, \langle c_1, \ldots, c_k \rangle)$
Feedback	$(e, \langle c_1, \ldots, c_k \rangle)$	$(e, \langle c_1, \ldots, c_k + 1 \rangle)$

Approach
Pointstamps & Scheduling

- Events uniquely identified by:

$$\text{Pointstamp} : (t \in \text{Timestamp}, \overbrace{l \in \text{Edge} \cup \text{Vertex}}^{\text{location}}).$$



Methods change the pointstamp of m:

- v.send_by(e, m, t) -> (t, e)
- v.notify_at(t) -> (t, v)
- „p1 *could result* in p2“ <-> ∃ ψ(t1) <= t2; then find minimal (earlierst) paths
  - ψ: path, where we transform p1 according to the update rules (l1 -> l2)

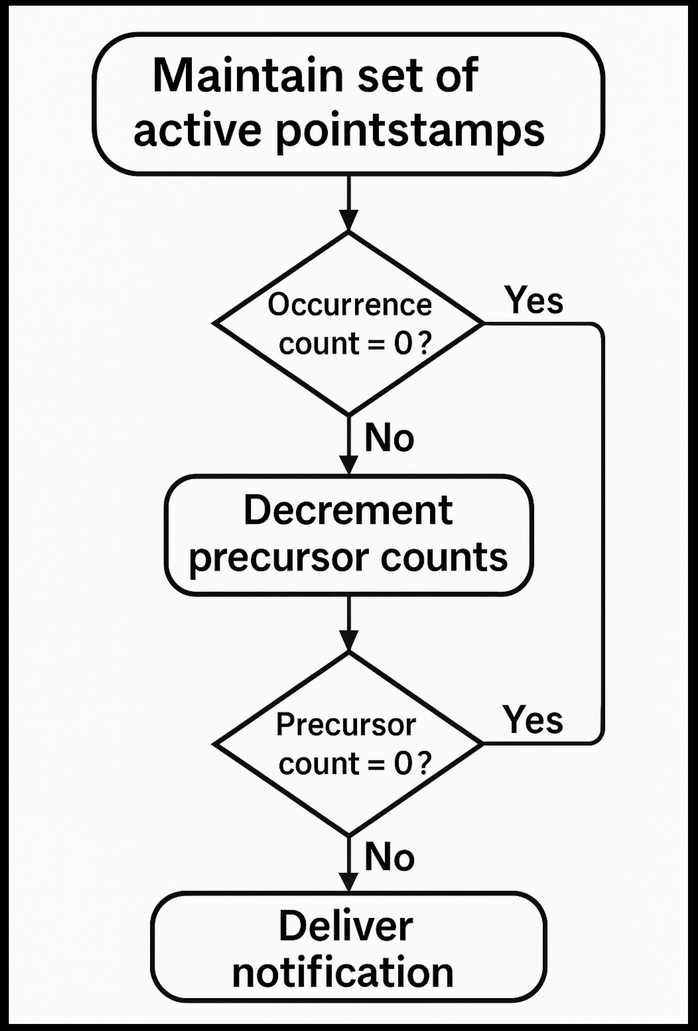
Approach

Pointstamps & Scheduling

Scheduler:

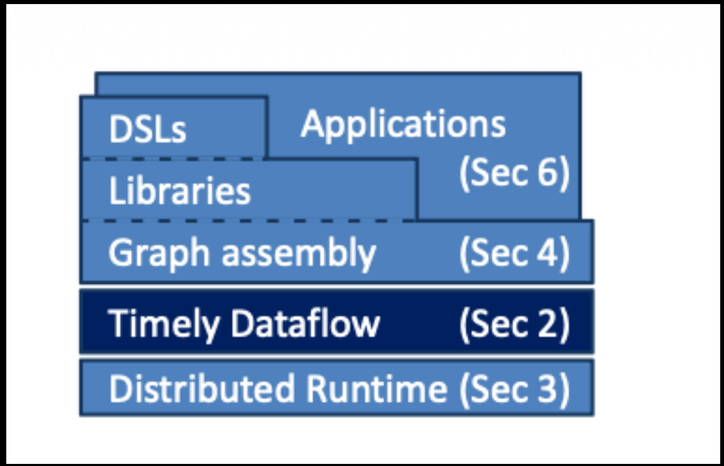
- Maintains a set of active point stamps
- Tracks two counters per pointstamp:
  - **Occurrence**: Outstanding events with that timestamp
  - **Precursor**: Number of active timestamps that *could result* in this pointstamp

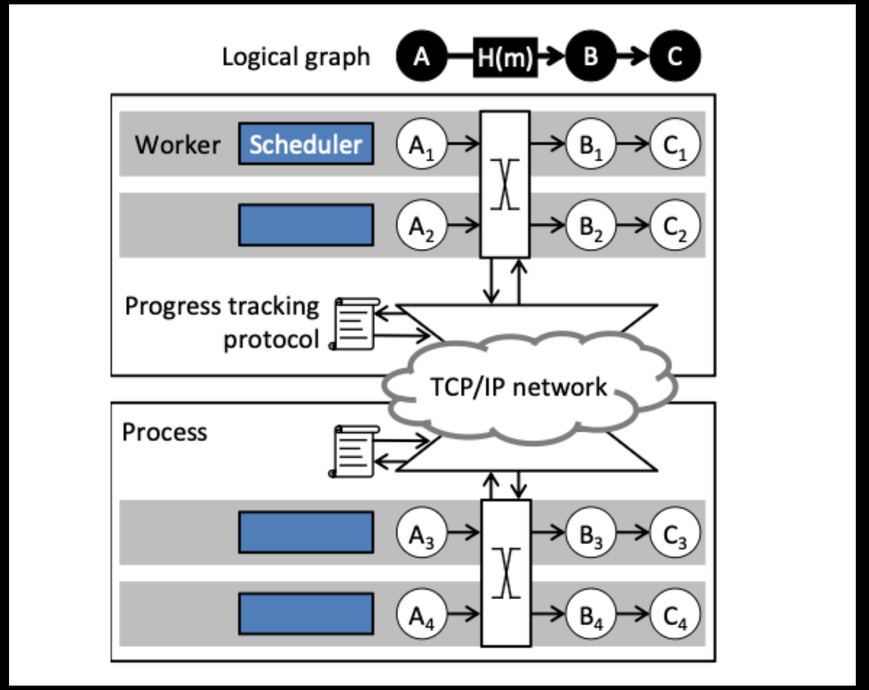
P1 scheduled when precursor = 0 & active



Approach

Naiad - Distributed Timely Dataflow





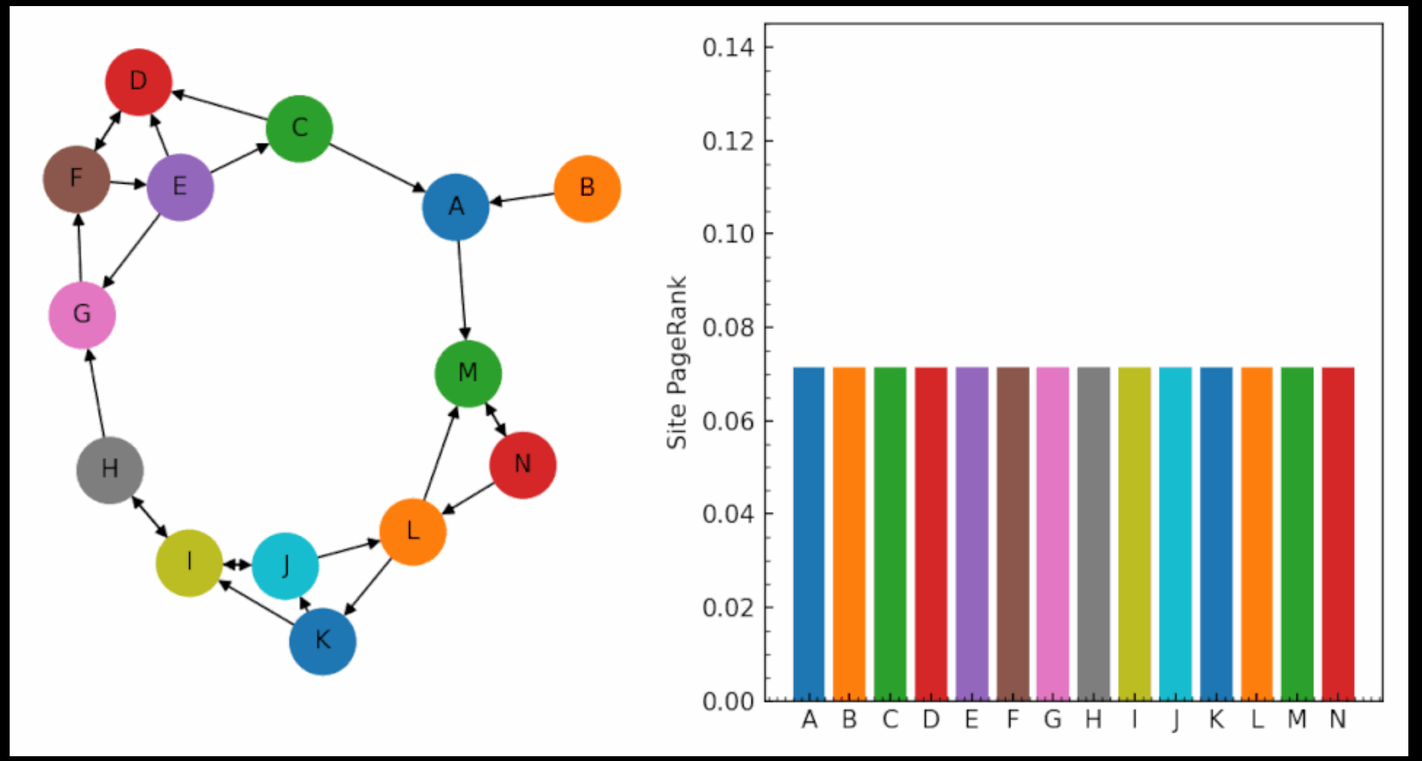
Approach

Naiad - Key Differences

	Single-threaded Timely	Distributed Naiad
Frontier tracking	Single global frontier	Each worker has *local frontier*
Progress metadata	Per-vertex pointstamps	Projected (logical-level) pointstamps (smaller state)
Coordination	None (local only)	Broadcast + aggregation of progress updates
Delivery condition	when no earlier local events	when no earlier events on *any* worker
Fault tolerance	Not required	Checkpoint + restore of vertex and progress state

Results

Performance Evaluation



[3]

Evaluation on Pagerank, Strong / Weakly Connected Components (SCC, WCC), All Pair Shortest Path (ASP)

Results

Performance Evaluation

Algorithm	PDW	DryadLINQ	SHS	Naiad
PageRank	156,982	68,791	836,455	**4,656**
SCC	7,306	6,294	15,903	**729**
WCC	214,479	160,168	26,210	**268**
ASP	671,142	749,016	2,381,278	**1,131**

Table 1: Running times in seconds of several graph algorithms on the Category A web graph. Non-Naiad measurements are due to Najork *et al.* [34].

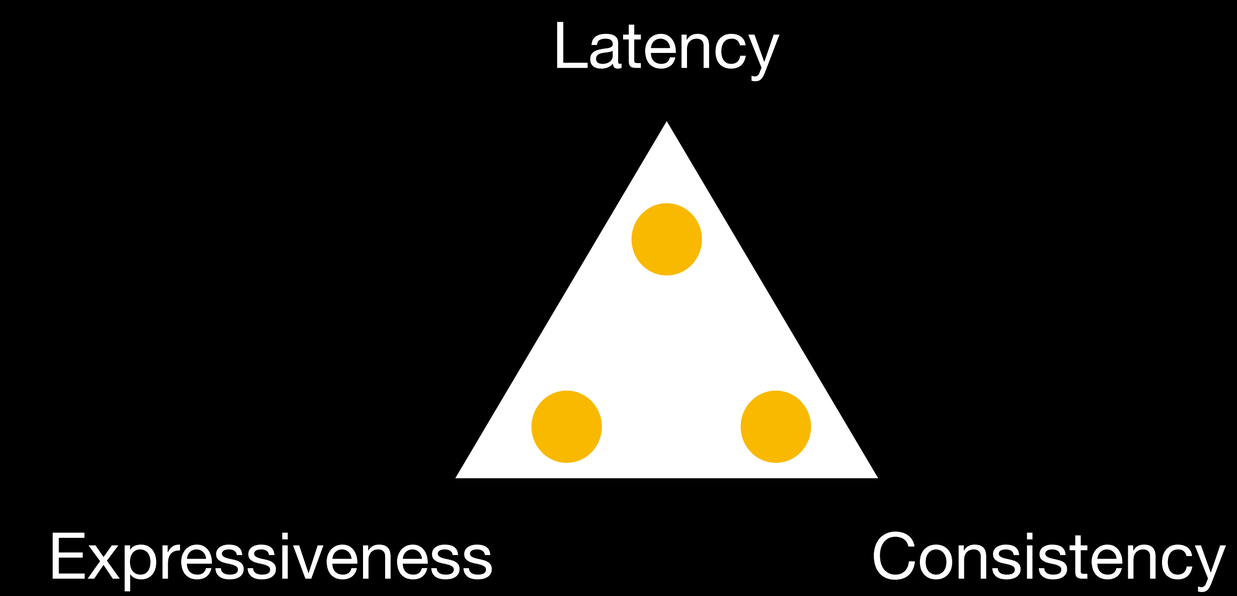
10-600x speedup!

Comparison: Distributed database (PDW), A general-purpose batch processor (DryadLINQ), and a purpose-built distributed graph store (SHS).

Discussion

Pro

- **Unified model**: Before, iteration was synchronous and slow; streaming had low latency but no iterative algorithms: Naiad / Timely Dataflow solved that problem

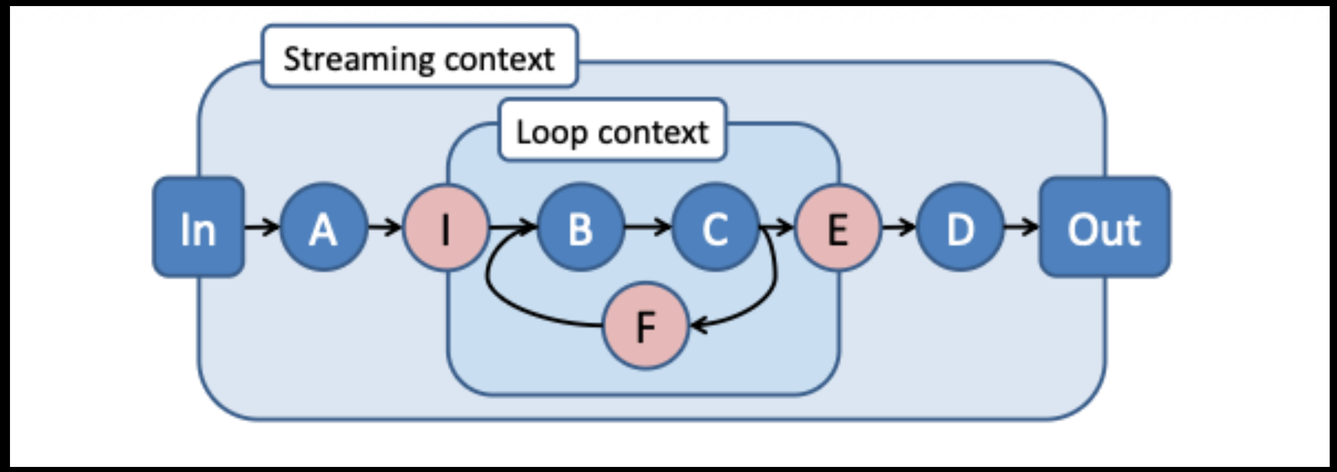


Discussion

Pro

- **Timestamps for Iterative Processes**: No global barrier like in batch systems, coordination only when needed, deterministic iteration in streaming systems

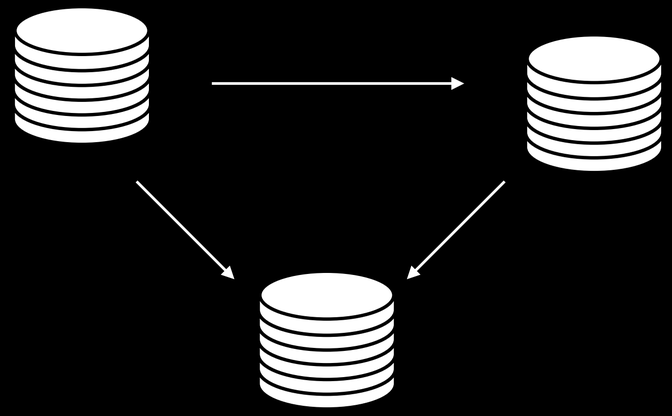
$$\text{Timestamp} : (\overbrace{e \in \mathbb{N}}^{\text{epoch}}, \overbrace{\langle c_1, \ldots, c_k \rangle \in \mathbb{N}^k}^{\text{loop counters}})$$



Discussion

Contra

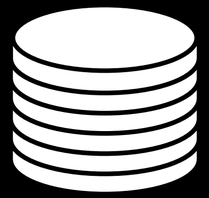
- **Coordination**: Global protocol still requires all workers to participate; there is a limit to scalability („it’s not a database“)

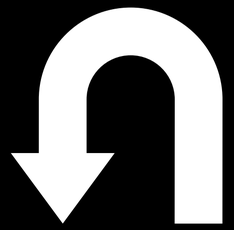


Discussion

Contra

- F**ault-tolerance is too simplistic: Recovery requires a global rollback to the** last checkpoint and temporarily pauses the whole system. (e.g., compare to Google Millwheel)





Discussion

Contra

- **Complexity for programmer:** non-trivial in comparison to e.g. Spark as e.g. reasoning about timestamps is required
- However, implementations nowadays abstract that away as well.

```
worker.dataflow(|scope| {
    scope.input_from(&mut input)
        .exchange(|x| *x)
        .inspect(move |x| println!("worker {}:\thello {}", index, x))
        .probe_with(&mut probe);
});
```

Discussion

Impact: Timely Dataflow and Naiad

TimelyDataflow / timely-dataflow Public

Code Issues 66 Pull requests 36 Actions Projects Wiki


```rust
fn main() {
    timely::execute_from_args(std::env::args(), |worker| {
        let index = worker.index();
        let mut input = InputHandle::new();
        let mut probe = ProbeHandle::new();

        worker.dataflow(|scope| {
            scope.input_from(&mut input)
                .exchange(|x| *x)
                .inspect(move |x| println!("worker {}:\thello {}", index, x))
                .probe_with(&mut probe);
        });

        for round in 0..10 {
            if index == 0 { input.send(round); }
            input.advance_to(round + 1);
            while probe.less_than(input.time()) {
                worker.step();
            }
        }
    }).unwrap();
}
```

[4]

Discussion

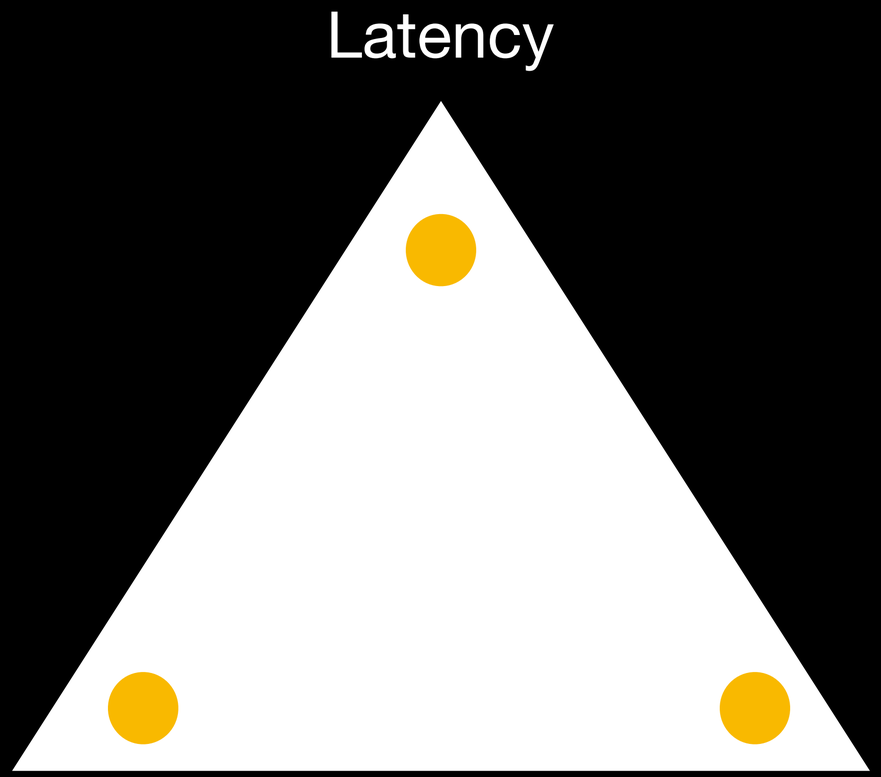
Impact: Timely Dataflow and Naiad

Concept from Naiad	Influenced
Timely Dataflow	Timely (Rust), Differential Dataflow
Frontier Tracking	Apache Flink (Watermarks), …
Low-latency iterative computation	Flink Unified Engine, Spark Structured Streaming
Iterative Computations	e.g. Streaming ML

Conclusion
Timely Dataflow

Timely Dataflow

- Great for data flow tasks!
- Inspired modern systems like timely dataflow and differential dataflow!



Naiad

- Unified low-latency streaming + iteration in one system
- Practical distributed progress

Thank you!

Questions?

[1] https://www.linkedin.com/posts/nikkisiapno_batch-processing-vs-real-time-streaming-activity-7316406319614251009-9yd1

[2] https://johncarlosbaez.wordpress.com/2016/06/05/programming-with-data-flow-graphs/

[3] https://en.wikipedia.org/wiki/File:Page_rank_animation.gif

[4] https://spark.apache.org/

https://nightlies.apache.org/flink/flink-docs-master/docs/dev/datastream/event-time/generating_watermarks/

https://github.com/TimelyDataflow/timely-dataflow