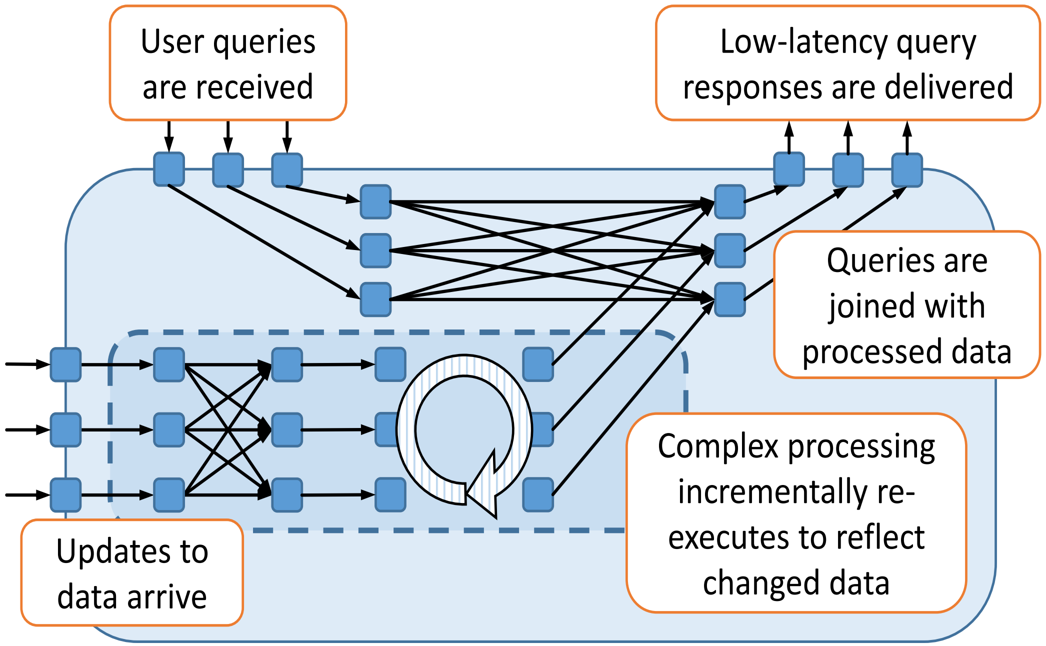
Paper Review
Naiad: A Timely Dataflow System[1]

Theo Long

19th October 2022

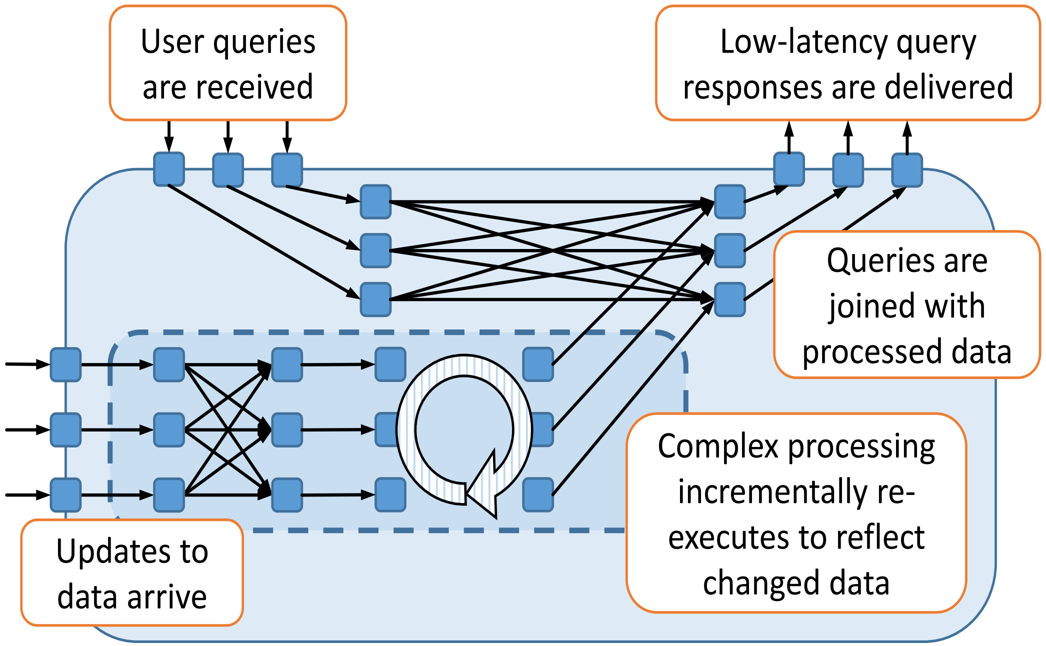
[1]All figures and information from Murray et. al (2013), unless otherwise specified

Naiad: High Performance Parallel Processing



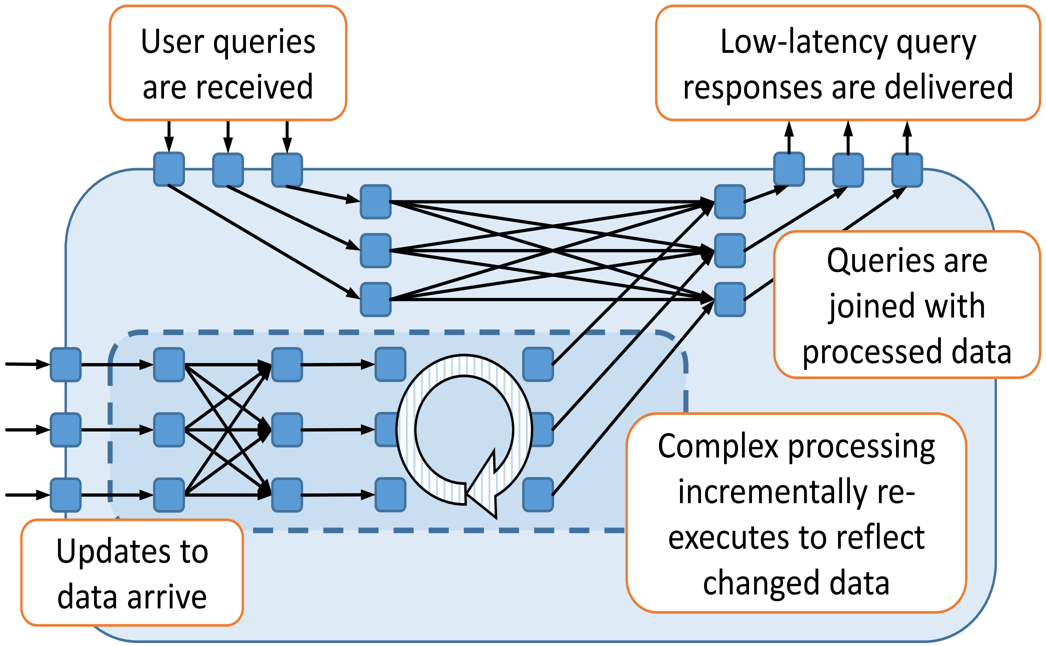
Naiad: High Performance Parallel Processing

1. High throughput (like batch processing)



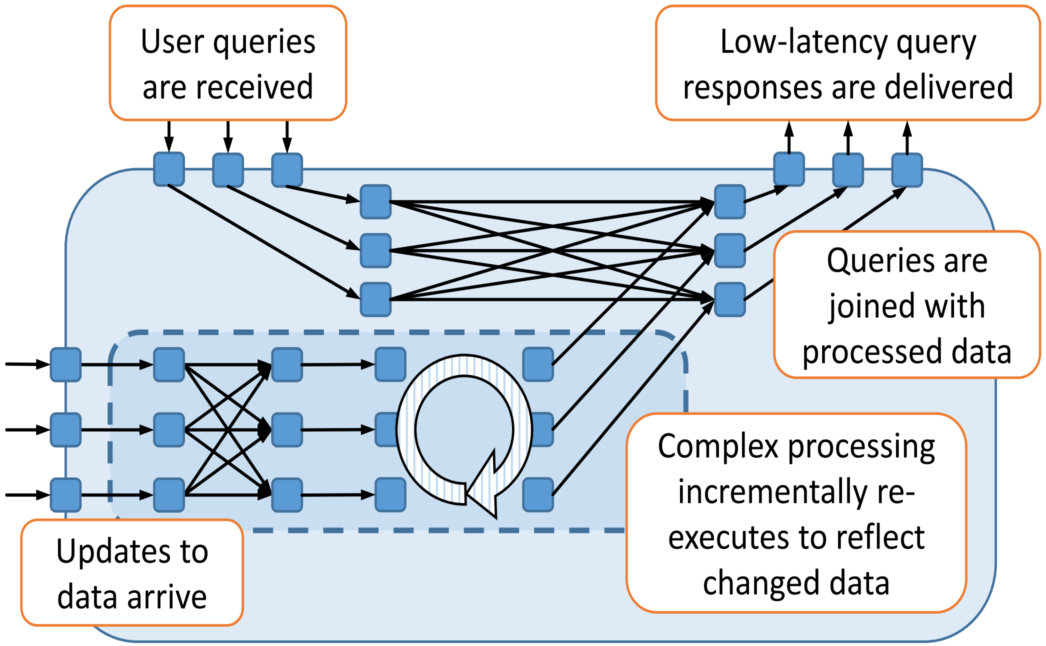
Naiad: High Performance Parallel Processing

1. High throughput (like batch processing)
2. Low latency (like stream processing)



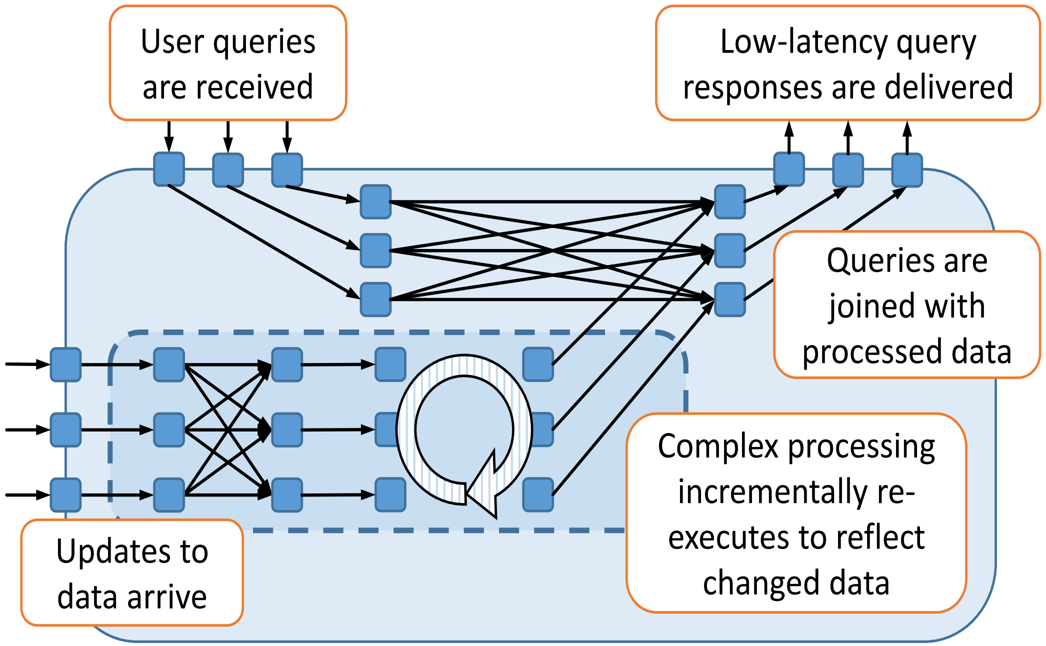
Naiad: High Performance Parallel Processing

1. High throughput (like batch processing)
2. Low latency (like stream processing)
3. Iterative Computation – i.e. loops



Naiad: High Performance Parallel Processing

1. High throughput (like batch processing)
2. Low latency (like stream processing)
3. Iterative Computation – i.e. loops
4. Strong Consistency Guarantees



What is Timely Dataflow?

What is Timely Dataflow?

- Directed Graph of stateful vertices

What is Timely Dataflow?

- Directed Graph of stateful vertices
- Allows for nested cycles

What is Timely Dataflow?

- Directed Graph of stateful vertices
- Allows for nested cycles
- Vertices send and receive timestamped messages allowing for a partial ordering of computations that **includes time**

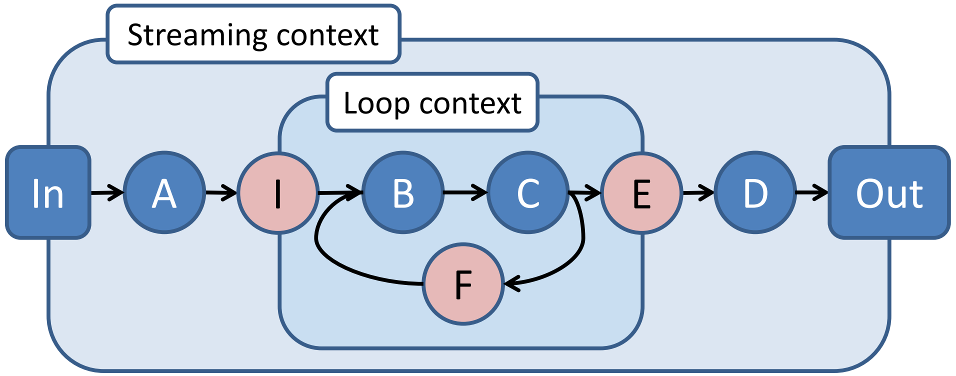
What is Timely Dataflow?

- Directed Graph of stateful vertices
- Allows for nested cycles
- Vertices send and receive timestamped messages allowing for a partial ordering of computations that **includes time**
- This allows for parallel computation **across epochs**

What is Timely Dataflow?

- Directed Graph of stateful vertices
- Allows for nested cycles
- Vertices send and receive timestamped messages allowing for a partial ordering of computations that **includes time**
- This allows for parallel computation **across epochs**
- Global notification when all messages of given timestamp received -> Allows us to reason about result ‘as of’ a certain time

Key Abstraction: Timestamps



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

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	$(e, \langle c_1, \ldots, c_k \rangle)$	$(e, \langle c_1, \ldots, c_k + 1 \rangle)$

Achieving Timely Dataflow with Timestamps

- Dataflow through the graph is done by **events** (messages or notification requests between nodes)

Achieving Timely Dataflow with Timestamps

- Dataflow through the graph is done by **events** (messages or notification requests between nodes)
- Time always increases, so for an event $\boldsymbol{e}$**,** only events with time $\boldsymbol{t} \geq \boldsymbol{te}$ can depend on $\boldsymbol{e}$

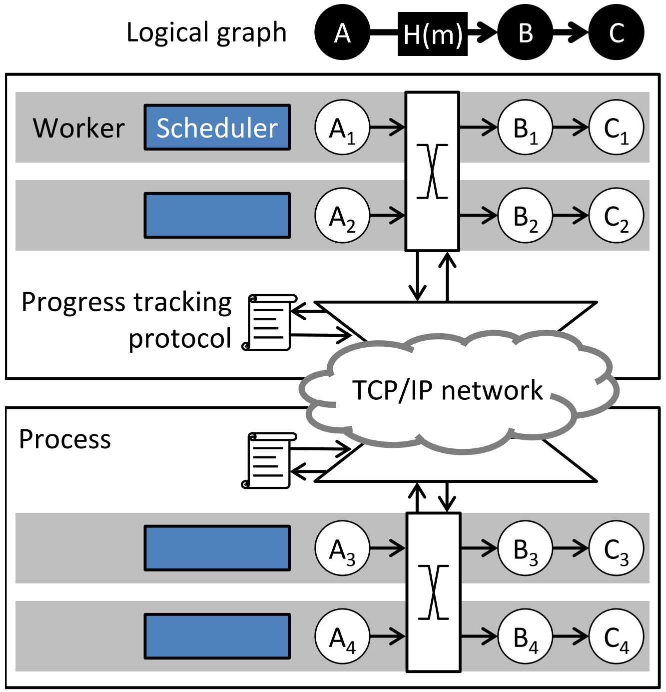
Achieving Timely Dataflow with Timestamps

- Dataflow through the graph is done by **events** (messages or notification requests between nodes)
- Time always increases, so for an event $\boldsymbol{e}$**,** only events with time $\boldsymbol{t} \geq \boldsymbol{te}$ can depend on $\boldsymbol{e}$
- Scheduler a list of **pointstamps** = time + location of unprocessed events
    - Each pointstamp has an **occurrence** count and **predecessor** count

Achieving Timely Dataflow with Timestamps

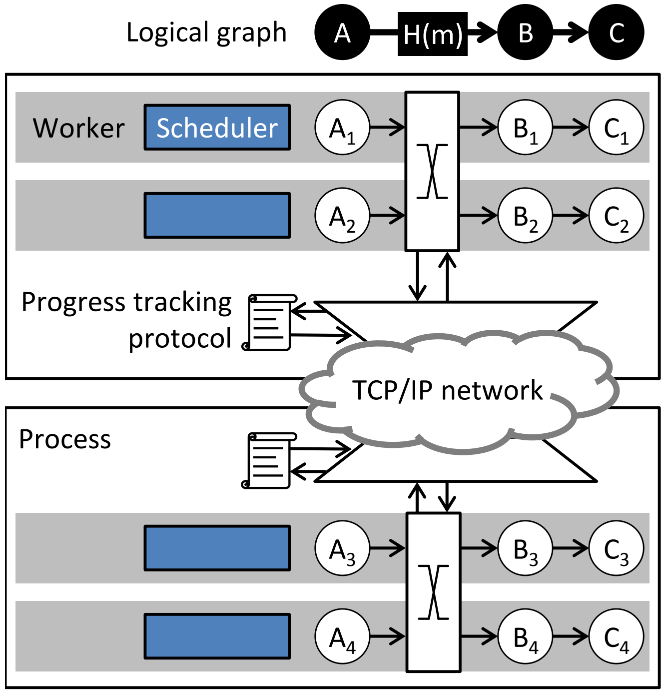
- Dataflow through the graph is done by **events** (messages or notification requests between nodes)
- Time always increases, so for an event $\boldsymbol{e}$**,** only events with time $\boldsymbol{t} \geq \boldsymbol{te}$ can depend on $\boldsymbol{e}$
- Scheduler a list of **pointstamps** = time + location of unprocessed events
    - Each pointstamp has an **occurrence** count and **predecessor** count
- When a pointstamp no longer has any predecessors (it is on the **frontier**), all corresponding events can be released

Making Timely Dataflow Distributed



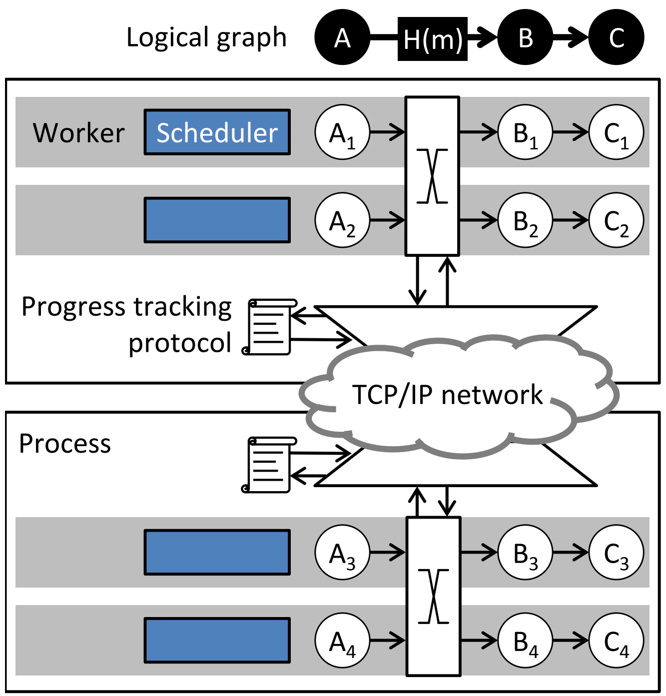
Making Timely Dataflow Distributed

- Naiad implements **data parallelism** – operating on multiple parts of the data at once



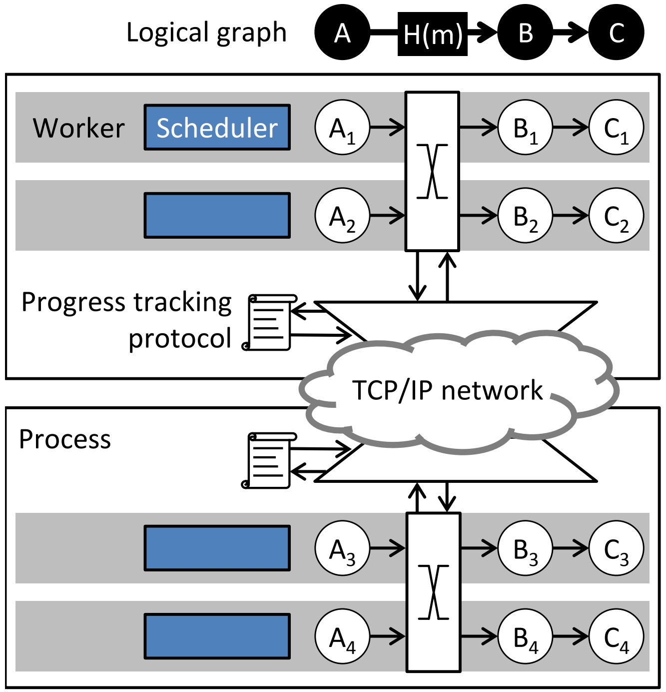
Making Timely Dataflow Distributed

- Naiad implements **data parallelism** – operating on multiple parts of the data at once
- Logical computation graph translated into **physical** graph



Making Timely Dataflow Distributed

- Naiad implements **data parallelism** – operating on multiple parts of the data at once
- Logical computation graph translated into **physical** graph
- Dependencies are always measured in the logical graph – this is not always optimal, but guarantees correctness



Making Timely Dataflow Distributed

- Each worker maintains a **local** view of global occurrence and precedence counts

Making Timely Dataflow Distributed

- Each worker maintains a **local** view of global occurrence and precedence counts
- When an event is dispatched by a worker, it broadcasts update to all other workers

Making Timely Dataflow Distributed

- Each worker maintains a **local** view of global occurrence and precedence counts
- When an event is dispatched by a worker, it broadcasts update to all other workers
- By processing incoming updates from a given worker in FIFO order, we guarantee that the local frontier is always a subset of the global frontier

Optimizing Broadcasts

1. Progress tracking is done on logical graph, which is much smaller than physical graph

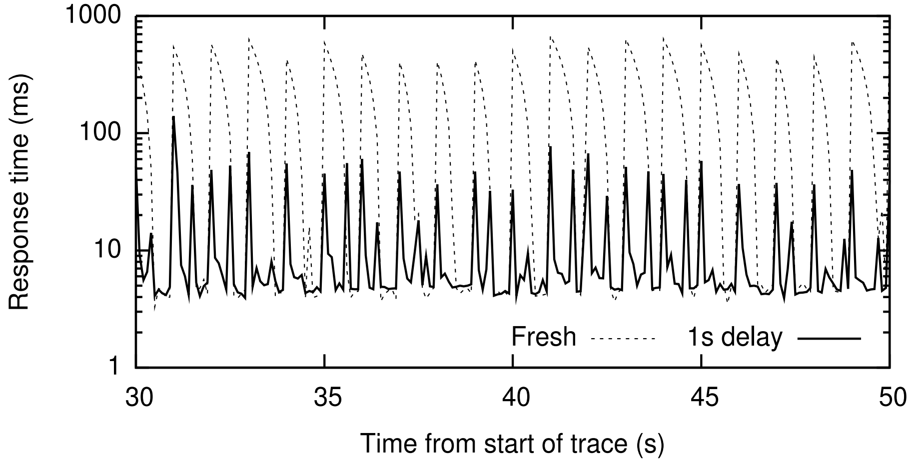
2. Updates are accumulated in a local buffer – updates with the same pointstamp are combined into one update

What do Naiad programs look like?

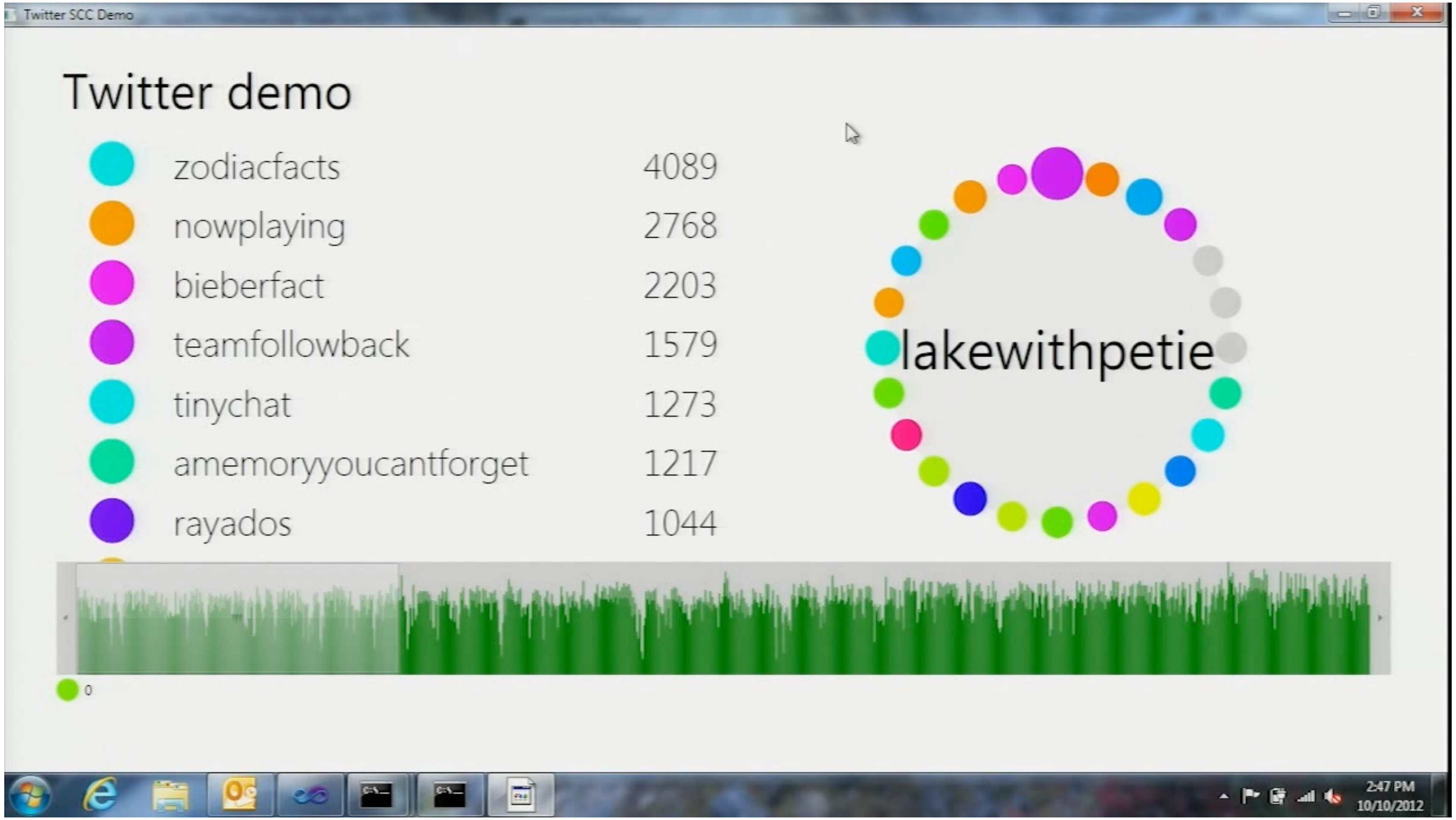
```
// 1a. Define input stages for the dataflow.
var input = controller.NewInput<string>();

// 1b. Define the timely dataflow graph.
// Here, we use LINQ to implement MapReduce.
var result = input.SelectMany(y => map(y))
                  .GroupBy(y => key(y),
                     (k, vs) => reduce(k, vs));

// 1c. Define output callbacks for each epoch
result.Subscribe(result => { ... });

// 2. Supply input data to the query.
input.OnNext(/* 1st epoch data */);
input.OnNext(/* 2nd epoch data */);
input.OnNext(/* 3rd epoch data */);
input.OnCompleted();
```

Naiad is Fast for Iterative Graph Computation

- Want to perform connected components on twitter mention graph
  - Also want to show top hashtag in each component
- One input stream of incoming tweets – 32k per second
- One query stream of usernames – 1 every 100ms



Naiad is fast for iterative graph computation



Source: 'Introducing Project Naiad and Differential Dataflow', https://www.youtube.com/watch?v=nRp3EQy6ccw

Thoughts, Comments, and Further Questions

- Is this kind of real-time querying useful?
  - Many analytics workloads can be performed in batch mode

Thoughts, Comments, and Further Questions

- Is this kind of real-time querying useful?
  - Many analytics workloads can be performed in batch mode
- Extremely sensitive to ‘micro-stragglers’ – single worker stalls
  - Requires significant system optimization and tuning, not an easy ‘run anywhere’ system

Thoughts, Comments, and Further Questions

- Is this kind of real-time querying useful?
  - Many analytics workloads can be performed in batch mode
- Extremely sensitive to ‘micro-stragglers’ – single worker stalls
  - Requires significant system optimization and tuning, not an easy ‘run anywhere’ system
- Requires good partitions for good parallelism – relies on developer

Thoughts, Comments, and Further Questions

- Is this kind of real-time querying useful?
  - Many analytics workloads can be performed in batch mode
- Extremely sensitive to ‘micro-stragglers’ – single worker stalls
  - Requires significant system optimization and tuning, not an easy ‘run anywhere’ system
- Requires good partitions for good parallelism – relies on developer
- Cannot support dynamic computation graphs

Thoughts, Comments, and Further Questions

- Timely Dataflow allows for **Differential Dataflow** – rather than recompute on all the data, just recompute on the part that changed

Thoughts, Comments, and Further Questions

- Timely Dataflow allows for **Differential Dataflow** – rather than recompute on all the data, just recompute on the part that changed
- Very well suited to real-time graph computation where number of iterations might be dynamic (e.g. BFS)

Thoughts, Comments, and Further Questions

- Timely Dataflow allows for **Differential Dataflow** – rather than recompute on all the data, just recompute on the part that changed
- Very well suited to real-time graph computation where number of iterations might be dynamic (e.g. BFS)
- Matches performance of many specialized systems of the time

Questions?