Naiad: a timely dataflow system

Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, Martín Abadi

Zak Singh

Naiad is a distributed system for high-throughput, low-latency, cyclic dataflow

What do we look for in a dataflow system?

System

Batch Processors | Stream Processors | Graph Processors

What do we look for in a dataflow system?

System

Batch Processors

Stream Processors

Graph Processors

Implies

Consistent

Low Latency

Supports Iteration

Batch Processors (MapReduce)

- Operate on “data at rest”
- “every night, calculate the previous day’s total sales”
- High throughput
- Easy to use and scale (very popular!)

Consistent

Low Latency

Supports Iteration

Batch Processors (MapReduce)

- Operate on “data at rest”
- “every night, calculate the previous day’s total sales”
- High throughput
- Easy to use and scale (very popular!)

- High latency
- No support for incremental computation
  - Have to recalculate from scratch every time

Consistent

Low Latency

Supports Iteration

Stateless Stream Processing

- Operate on "data in motion"
- "Running sum of total sales"
- **Fed timestamped events as they occur** by a message broker/queue (Kafka, Debezium, etc)

Consistent

Low Latency

Supports Iteration

Stateless Stream Processing

- Operate on “data in motion”
- “Running sum of total sales”
- **Fed timestamped events as they occur** by a message broker/queue (Kafka, Debezium, etc)

- Out of order arrivals mean aggregations not guaranteed to be correct

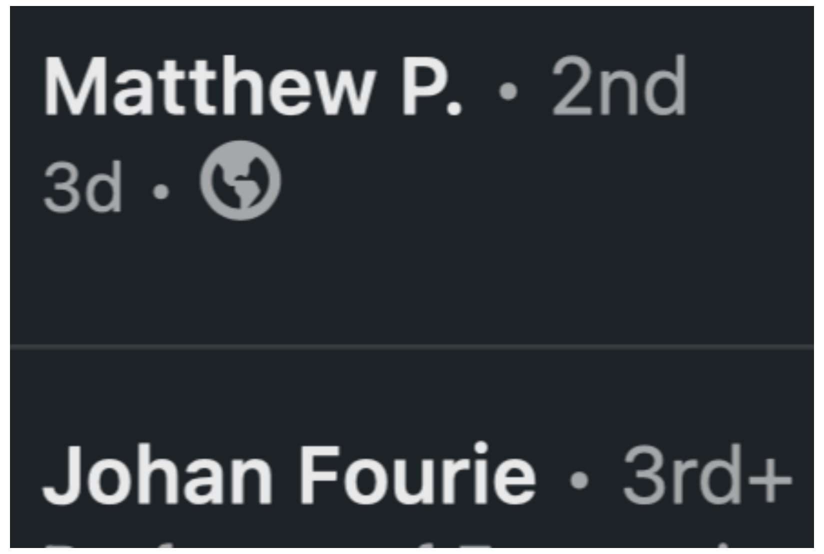
Consistent

Low Latency

Supports Iteration

Graph Processing

- “Find the degree of connection (shortest path) between me and another user on LinkedIn”
- GraphX (on top of Spark), Giraph
- No clear victor in the space, open problem
- Why? Graph traversals require **iterative algorithms**



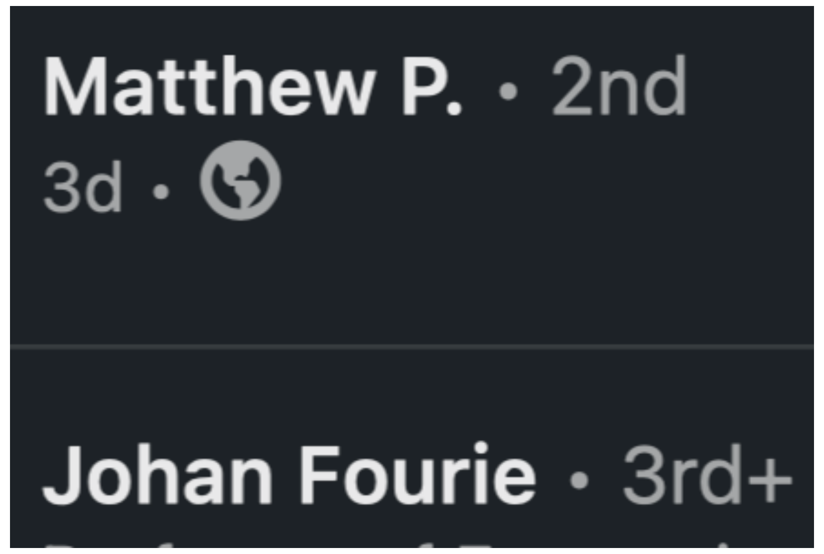
Consistent

Low Latency

Supports Iteration

Graph Processing

- "Find the degree of connection (shortest path) between me and another user on LinkedIn"
- GraphX (on top of Spark), Giraph
- No clear victor in the space, open problem
- Why? Graph traversals require **iterative algorithms**



- Most dataflow systems are acyclic
- Hard to parallelize iteration

Consistent

Low Latency

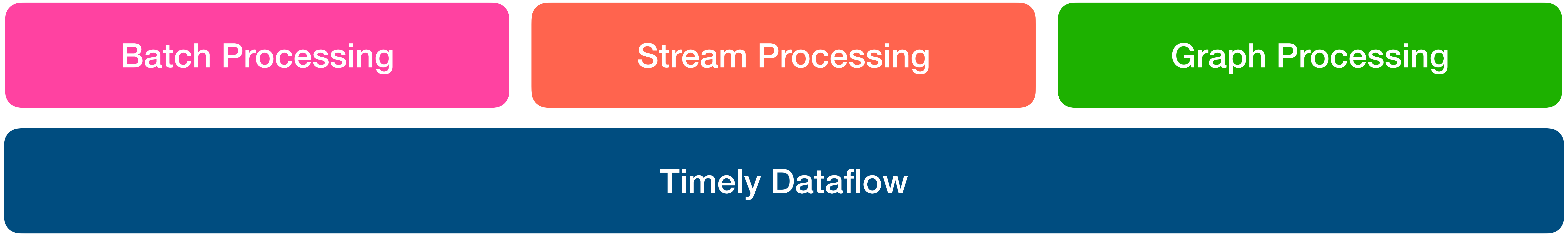
Supports Iteration

One framework to rule them all?

Timely Dataflow

Consistent | Low Latency | Supports Iteration

A shared runtime



How does Timely Dataflow achieve all this?

How does Timely Dataflow achieve all this?

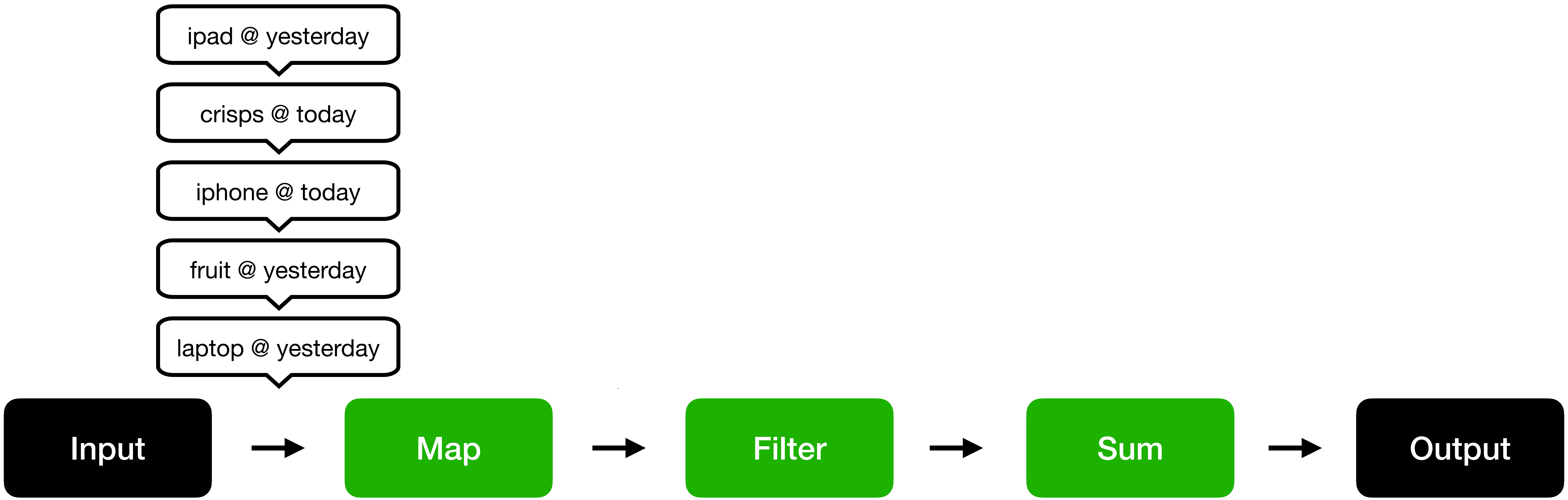
Timestamps!

Stateless Stream Consistency

How much revenue are we making from high value item sales, per day?

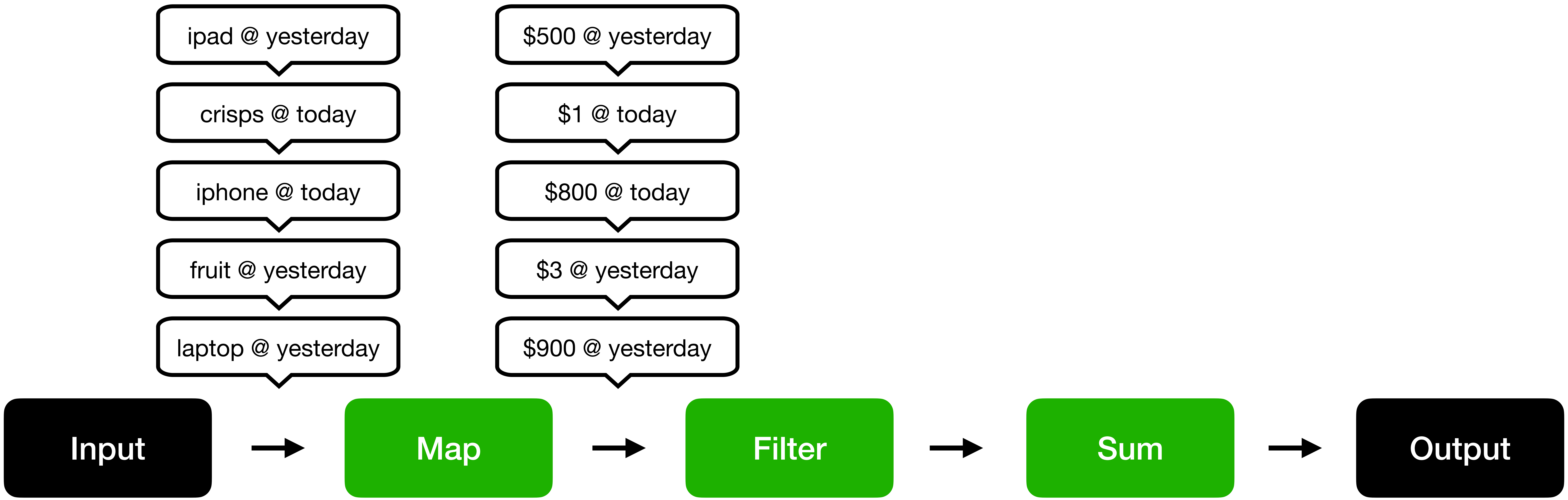
Stateless Stream Consistency

How much revenue are we making from high value item sales, per day?



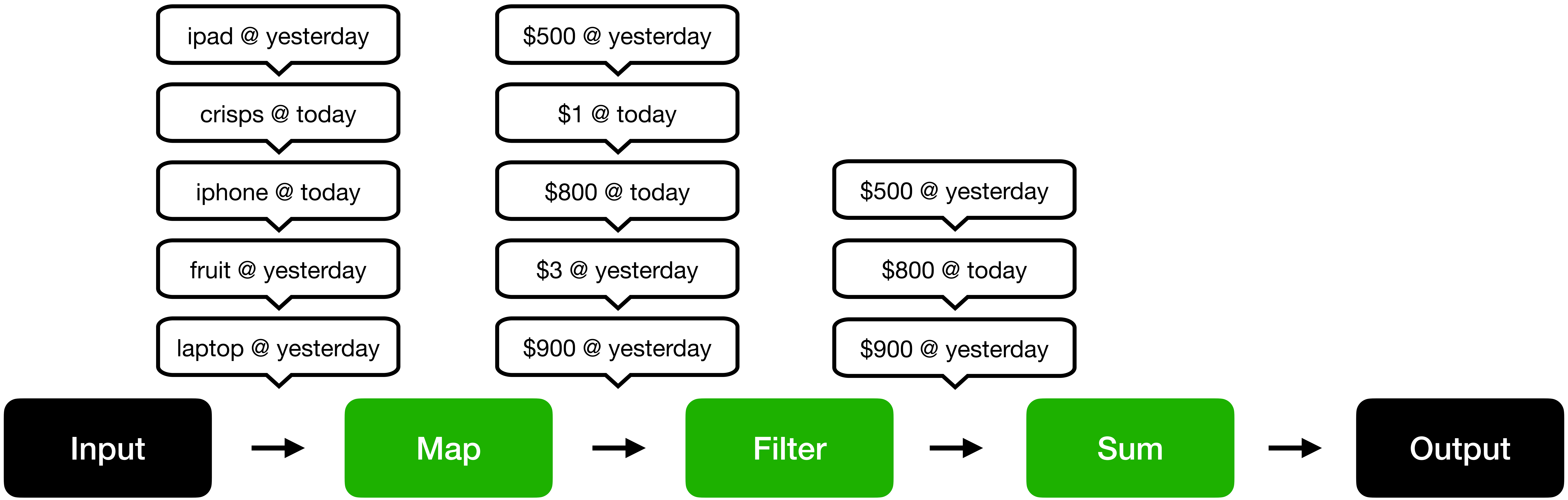
Stateless Stream Consistency

How much revenue are we making from high value item sales, per day?



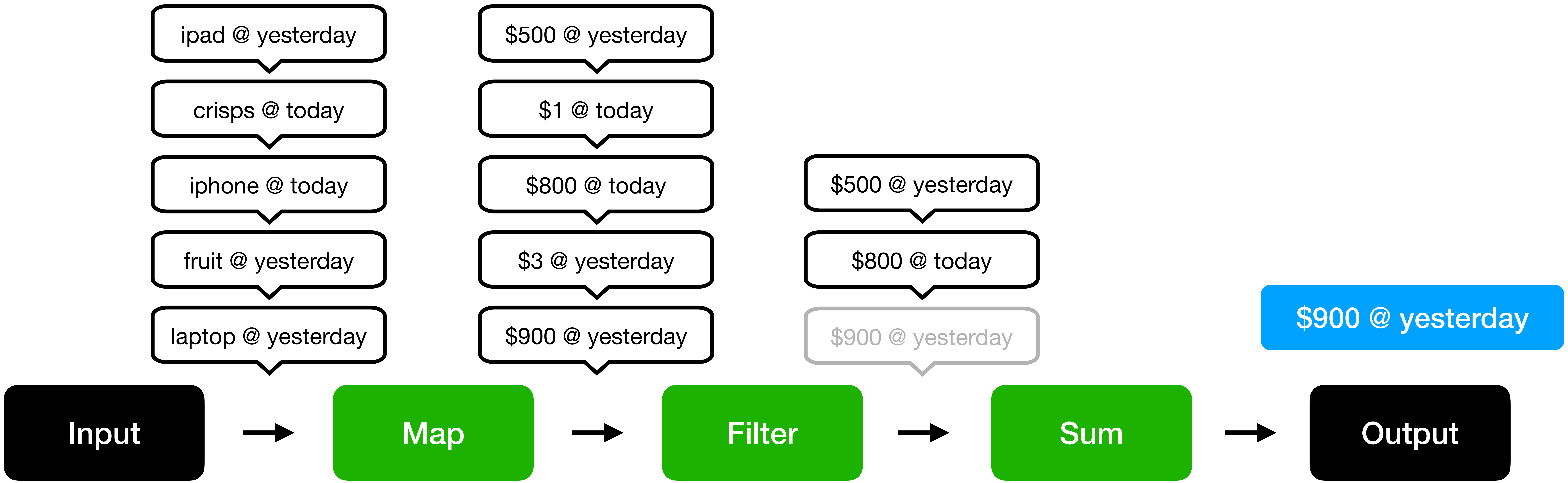
Stateless Stream Consistency

How much revenue are we making from high value item sales, per day?



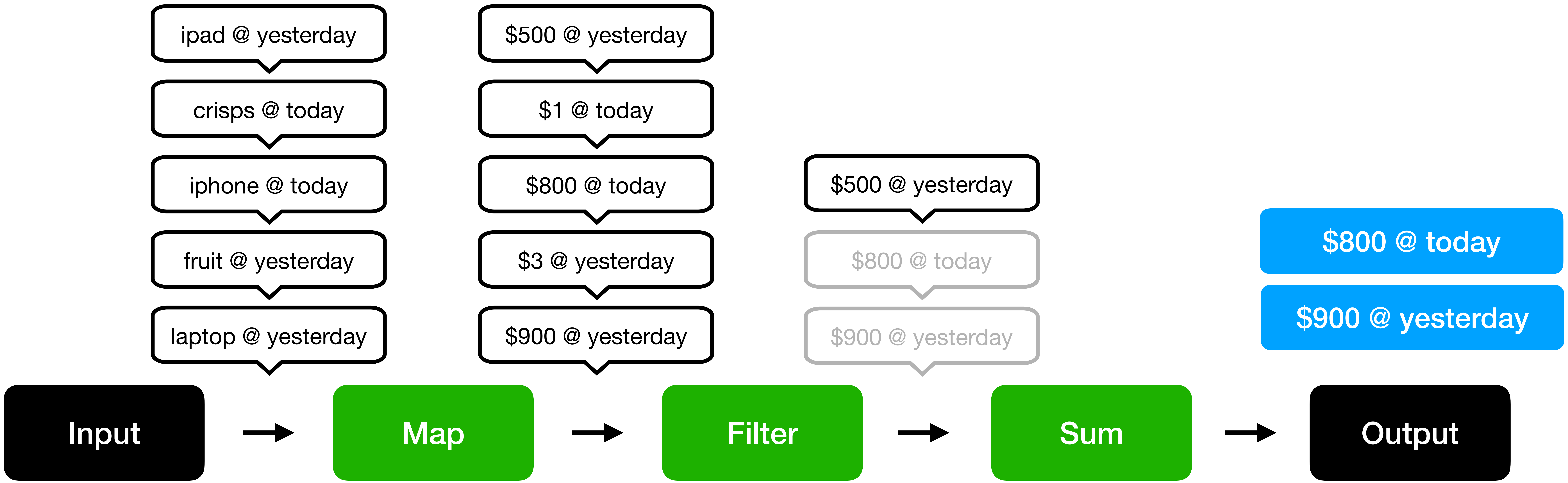
Stateless Stream Consistency

How much revenue are we making from high value item sales, per day?



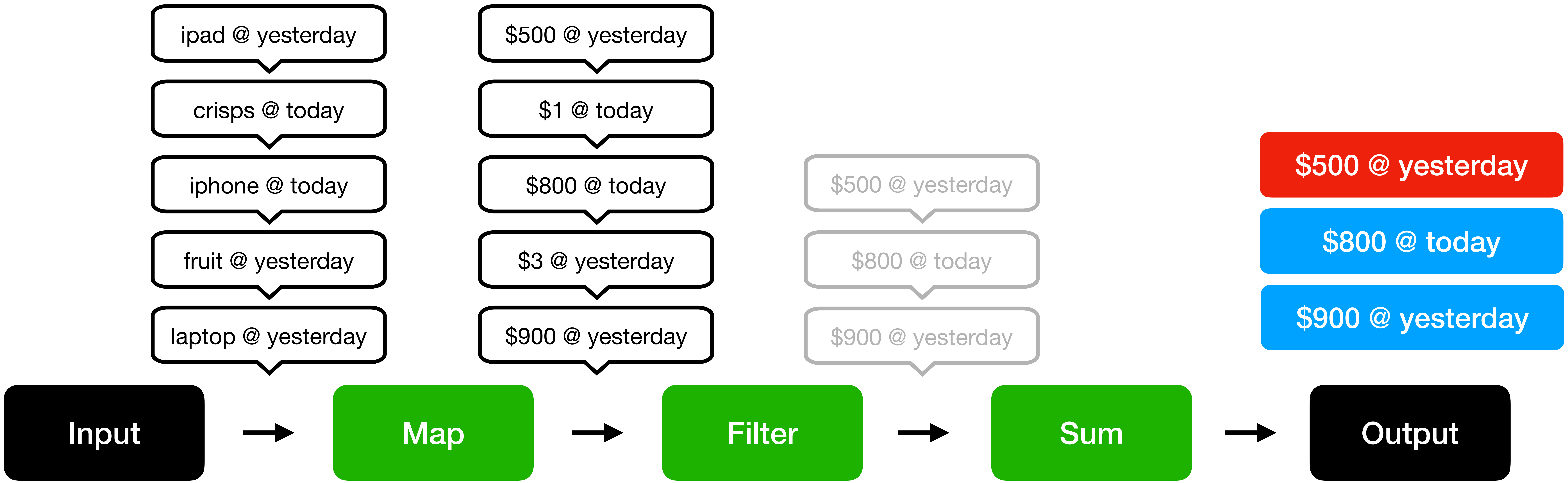
Stateless Stream Consistency

How much revenue are we making from high value item sales, per day?



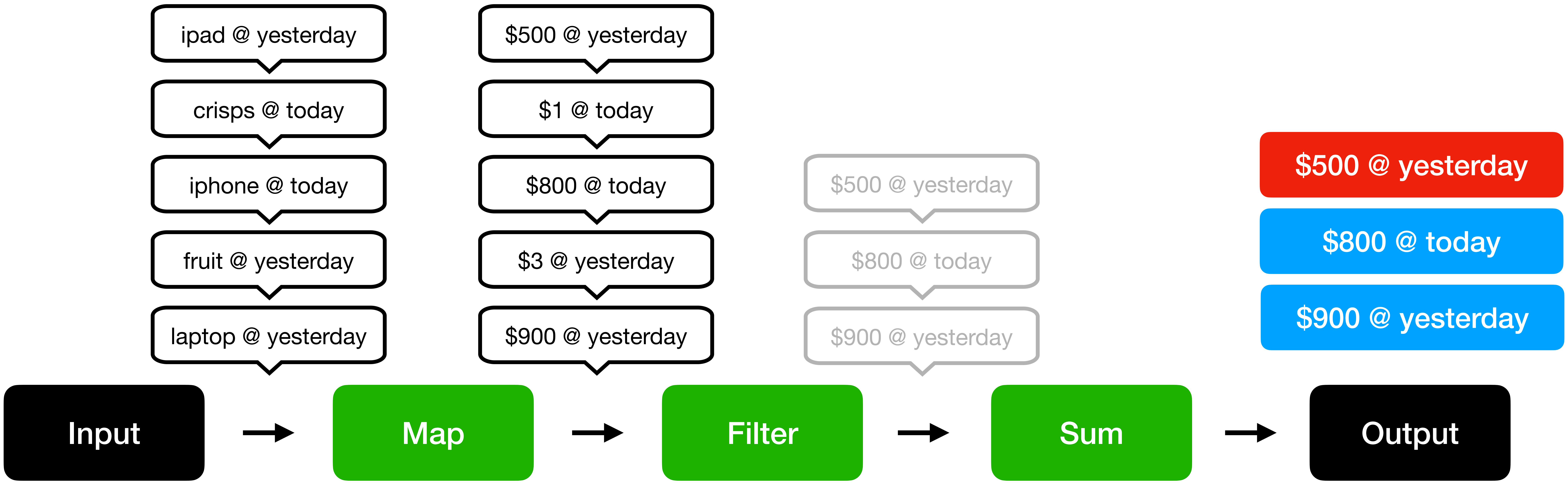
Stateless Stream Consistency

How much revenue are we making from high value item sales, per day?



Stateless Stream Consistency

How much revenue are we making from high value item sales, per day?

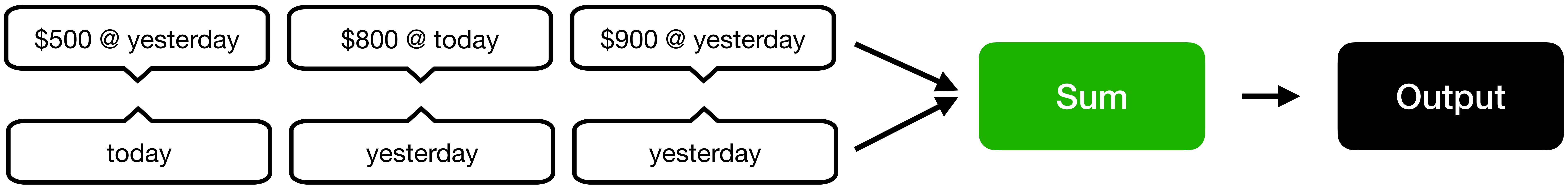


Sum needs to know the **minimum timestamp still upstream** so that it can **statefully** hold onto yesterday's records until it's seen all of them

Timely Data Flow Consistency

How much revenue are we making from high value item sales, per day?

Event plane



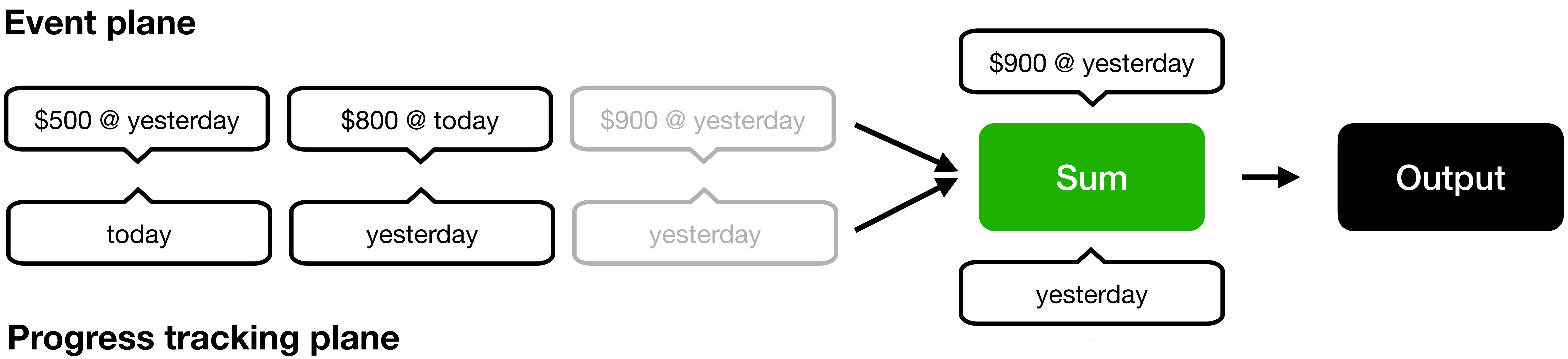
Progress tracking plane

"Minimum timestamp after me"

Sum maintains internal state

Timely Data Flow Consistency

How much revenue are we making from high value item sales, per day?



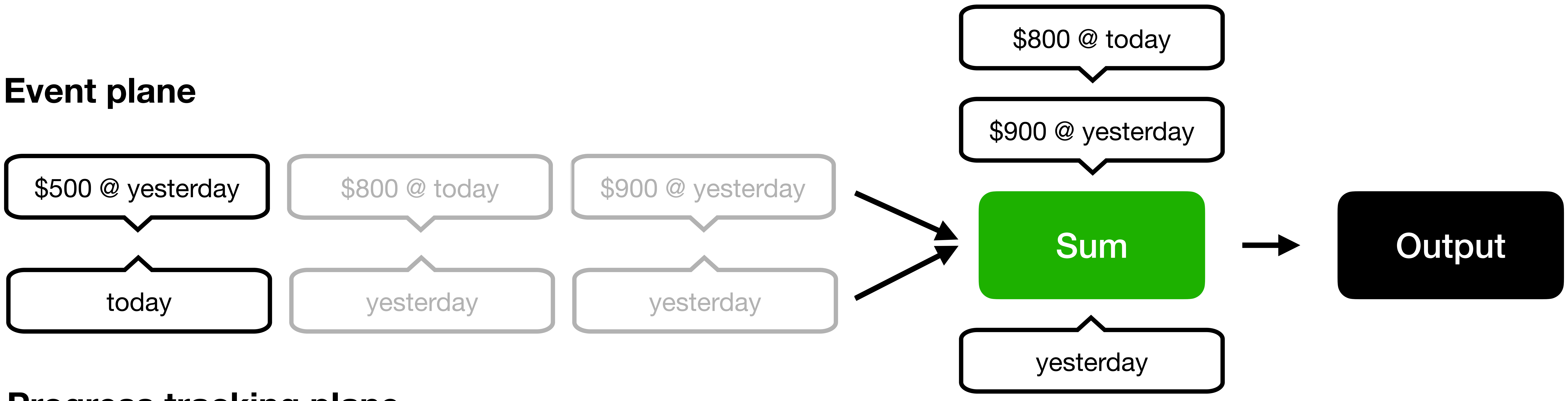
“Minimum timestamp after me”

Sum maintains internal state

Timely Data Flow Consistency

How much revenue are we making from high value item sales, per day?

Event plane



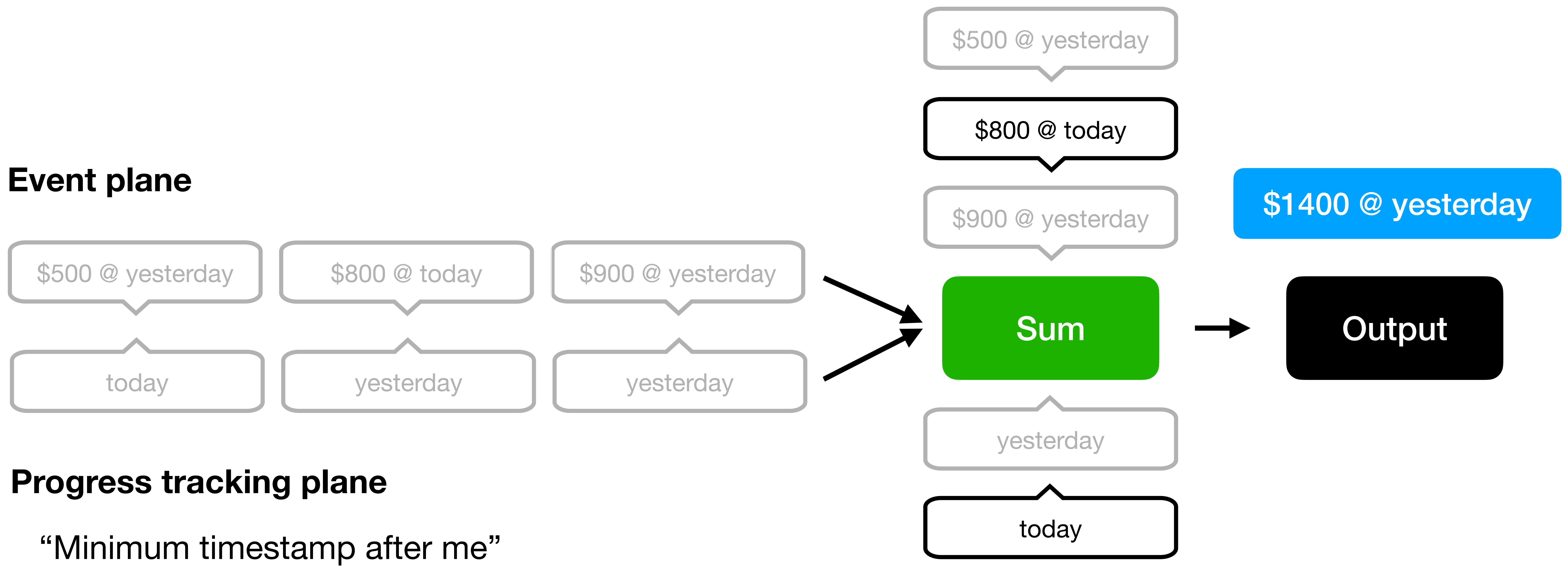
Progress tracking plane

"Minimum timestamp after me"

Sum maintains internal state

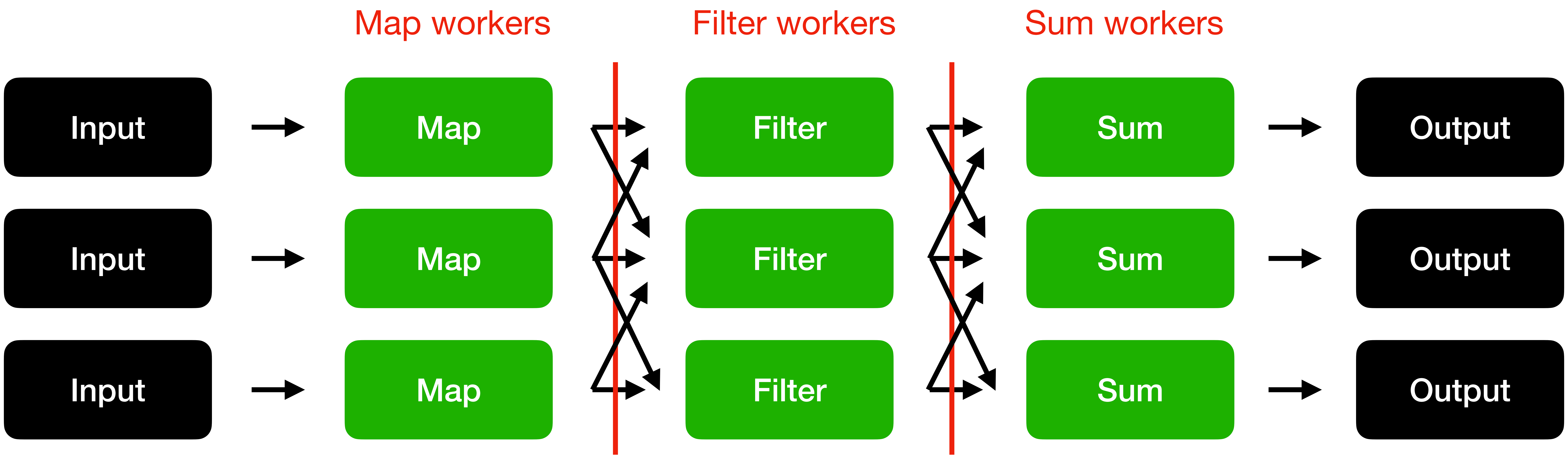
Timely Data Flow Consistency

How much revenue are we making from high value item sales, per day?



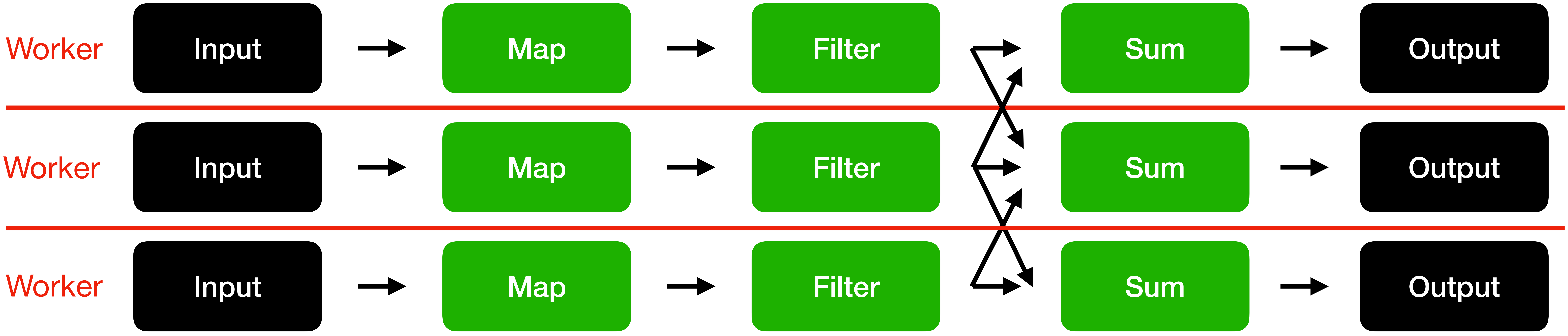
Sum maintains internal state

Coordination: The usual way



- Each worker has no awareness of larger graph
- Each operator is stateless (in most systems)

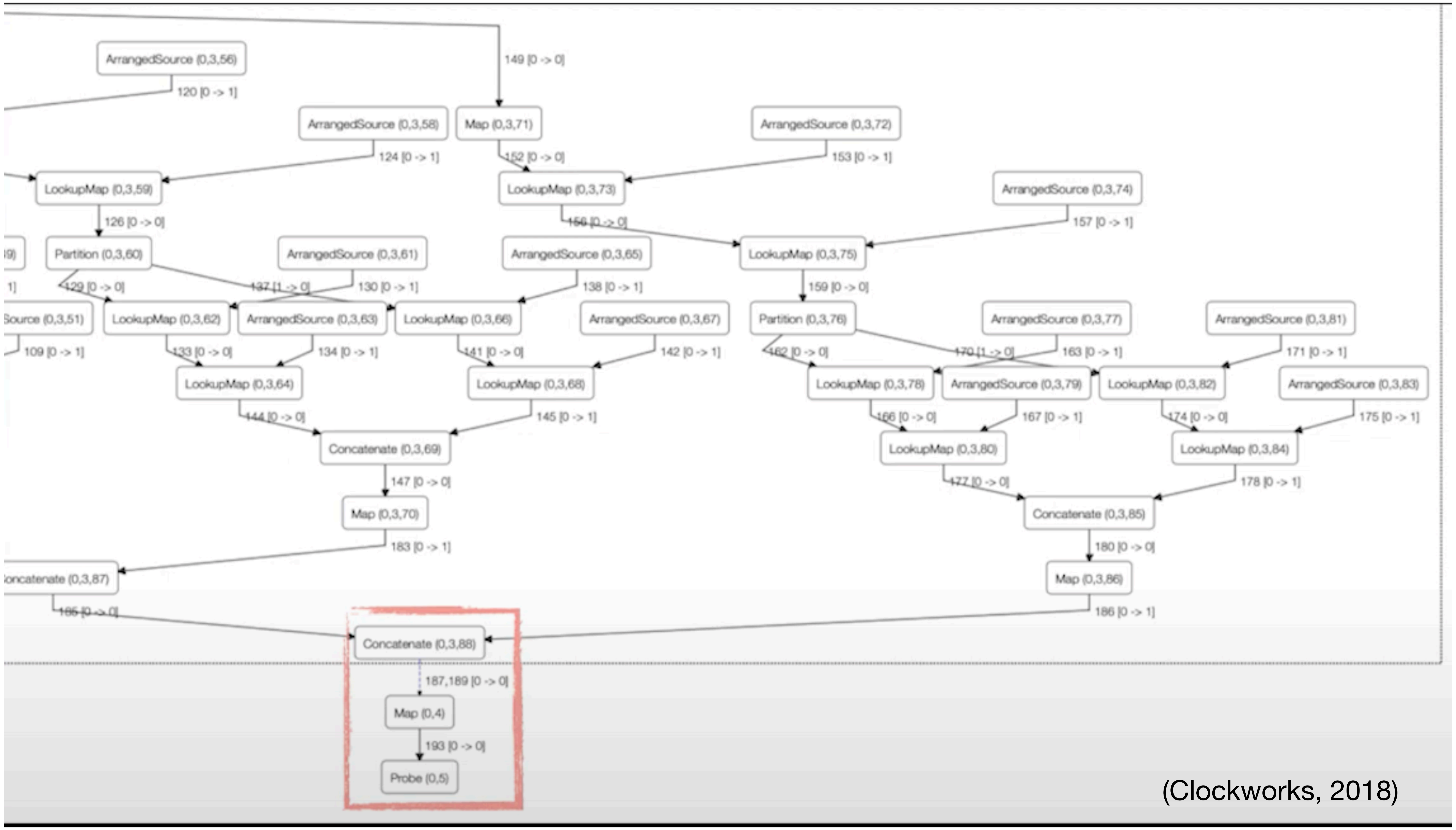
Parallelization: The Timely Dataflow way



- Coordination only occurs where needed (the Sum operator)
- Consistency guaranteed, while maintaining low latency!

Efficiency gains at scale

- Paths don't coordinate unless they need to!



(Clockworks, 2018)

Recap

Timely Dataflow

Recap

Timely Dataflow

Consistent

- Workers coordinate to determine minimum timestamp upstream at each operator

Recap

Timely Dataflow

Consistent

Low Latency

- Workers coordinate to determine minimum timestamp upstream at each operator

- But only when needed

Recap

Timely Dataflow

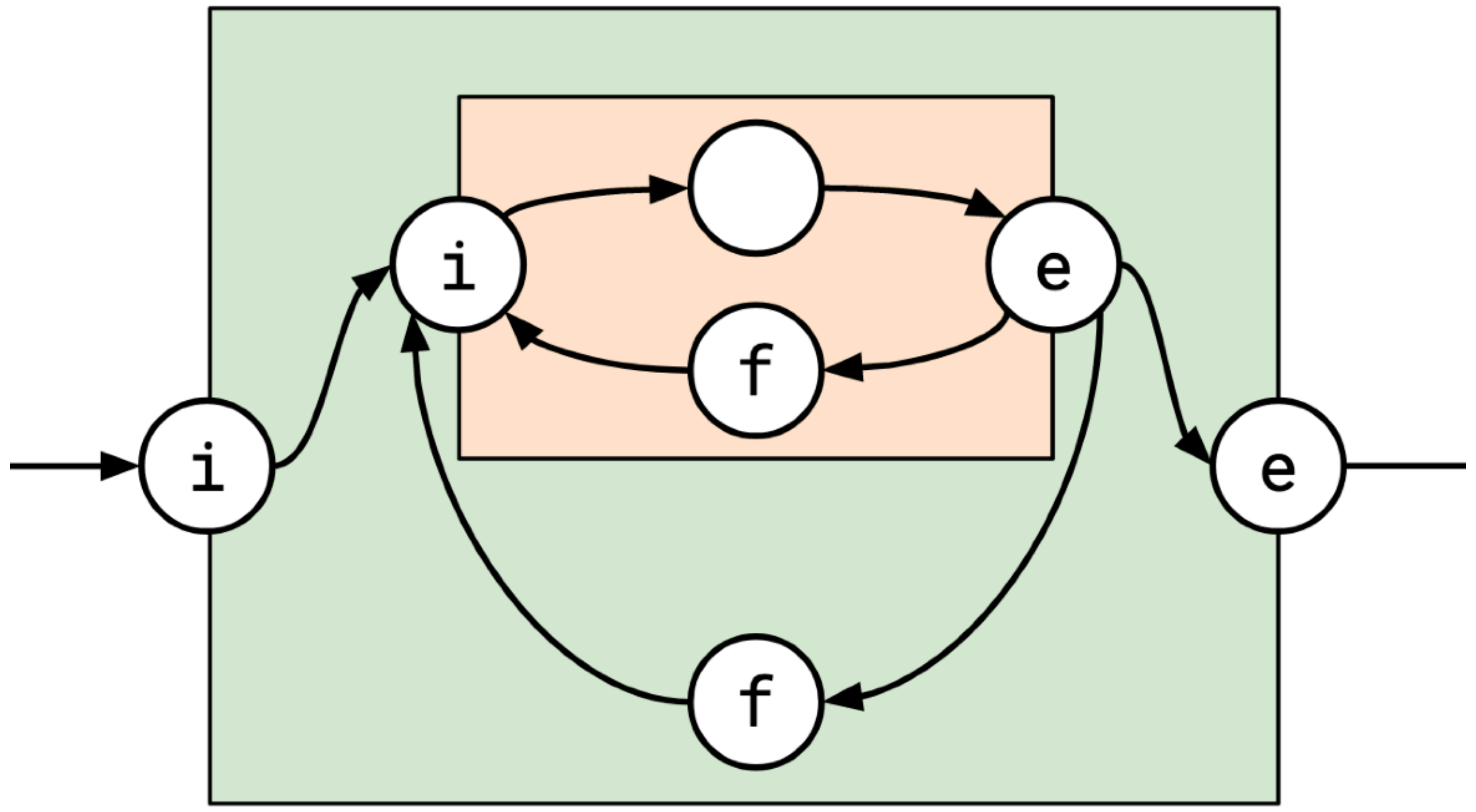
Consistent | Low Latency | Supports Iteration?

- Workers coordinate to determine minimum timestamp upstream at each operator

- But only when needed

Expressive iteration

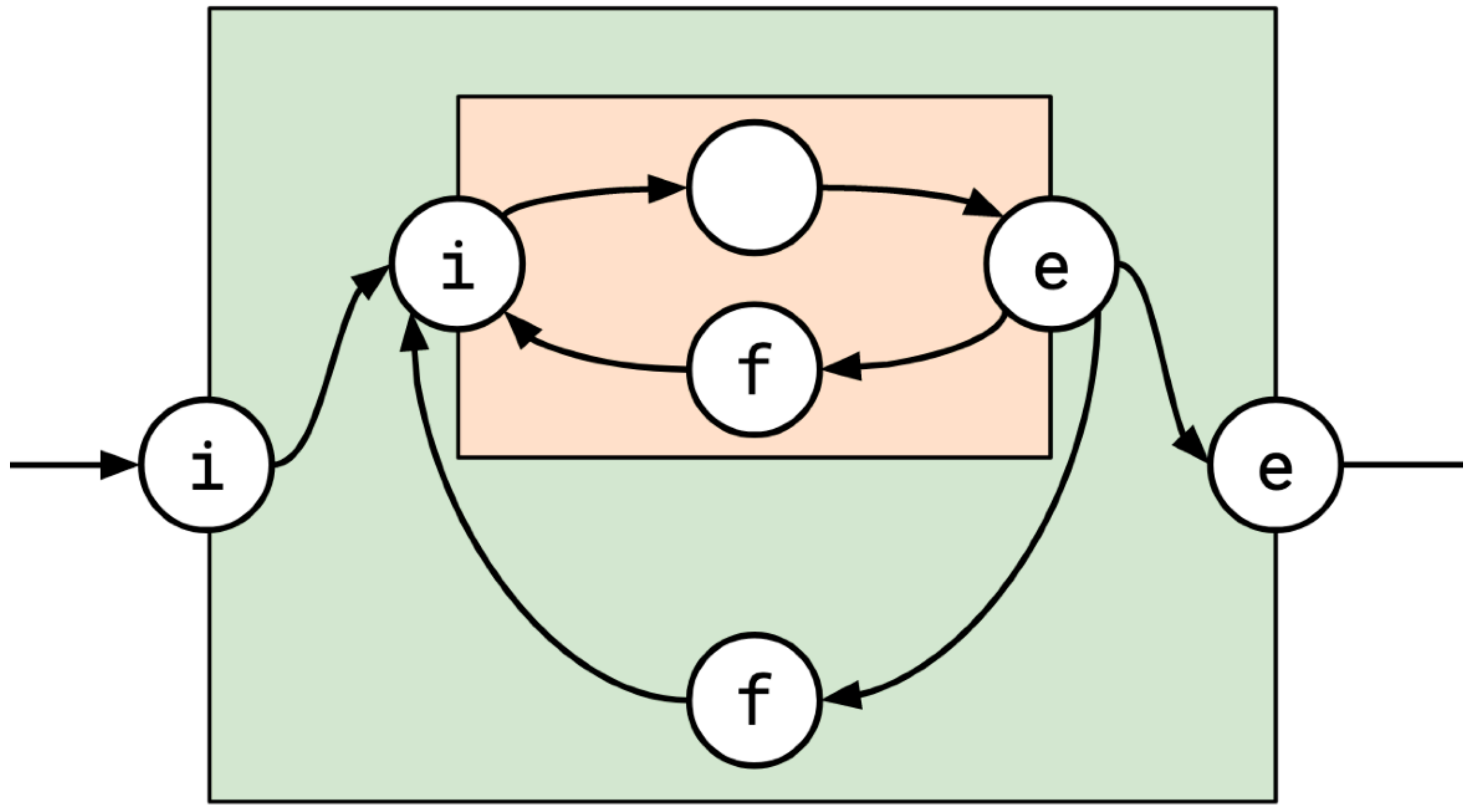
- Timestamps + stateful vertices make iteration achievable
- Append a loop counter to each timestamp on entry to loop
- Increment counter by passing through *feedback node*
- Arbitrarily nested loops supported (just append more loop counters to the timestamp)



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

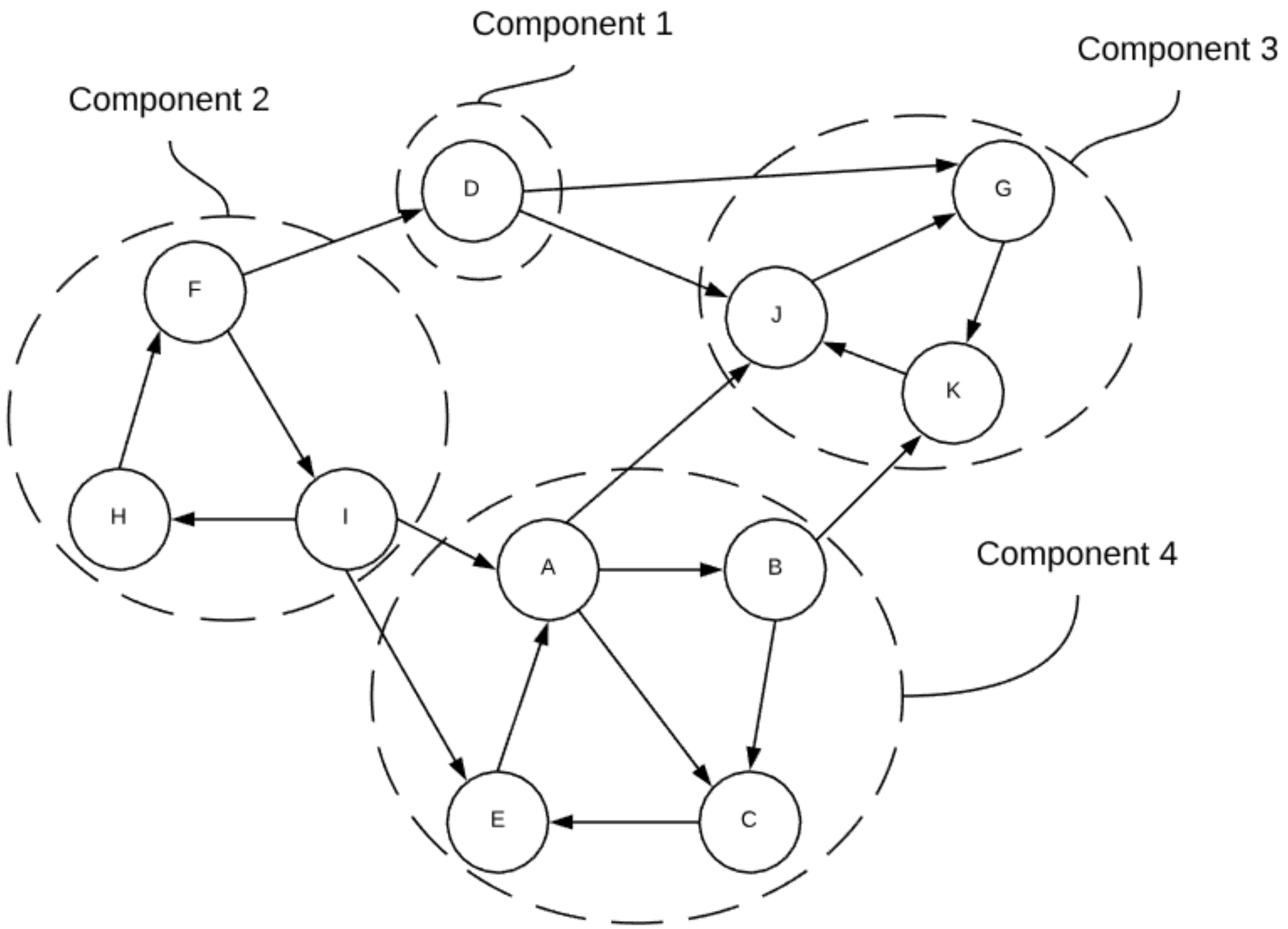
Expressive iteration

- Timestamps + stateful vertices make iteration achievable
- Append a loop counter to each timestamp on entry to loop
- Increment counter by passing through *feedback node*
- Arbitrarily nested loops supported (just append more loop counters to the timestamp)
- Still maintains consistency and low latency!



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

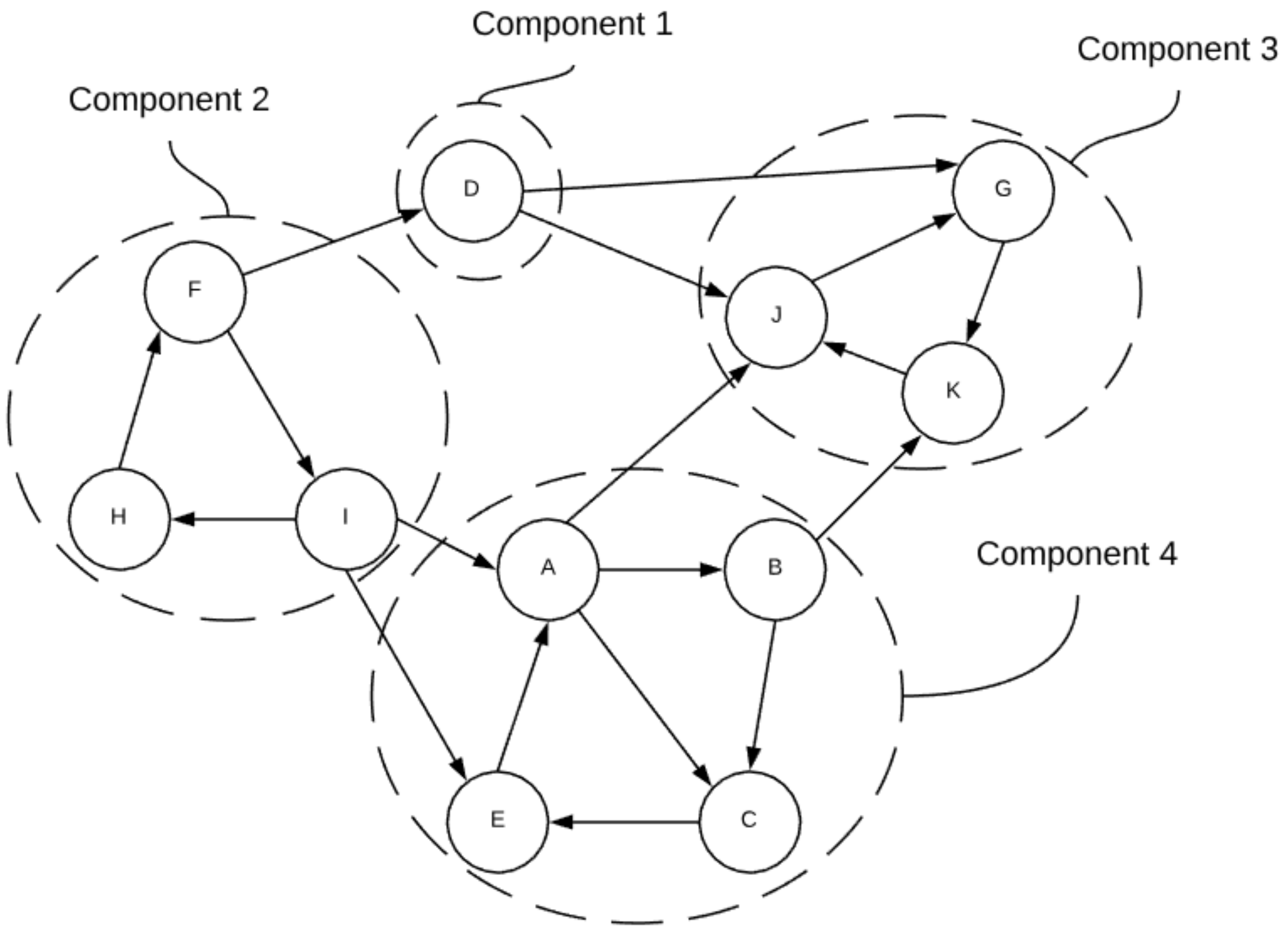
Performance (SCC)



Connected	Cores	Livejournal	orkut
GraphX	128	59s	53s
SociaLite	128	54s	78s
Myria	128	37s	57s
BigDatalog	128	27s	33s
Timely Dataflow	**1, 2**	**20s, 11s**	**43s, 26s**

(Clockworks, 2019)

Performance (SCC)



Connected	Cores	Livejournal	orkut
GraphX	128	59s	53s
SociaLite	128	54s	78s
Myria	128	37s	57s
BigDatalog	128	27s	33s
Timely Dataflow	**1, 2**	**20s, 11s**	**43s, 26s**
Differential update	**1, 2**	**98us, 109us**	**200us, 216us**

(Clockworks, 2019)

So why isn't everyone using it?

Timely Dataflow

Consistent	Low Latency	Supports Iteration

So why isn't everyone using it?

(Opinions are my own)

Timely Dataflow

Consistent	Low Latency	Supports Iteration

Generalized to a fault?

- Timely Dataflow is only the “simplest solution” when you need all of these properties (consistency, low latency, iteration)
  - Hard to come up with use case: real-time graph analytics?
- For *most* large-scale data processing, batch solutions are sufficient (and much simpler to use/reason about)
  - i.e. LinkedIn only calculates up to 3 degrees of separation, which can be done via batch processing, albeit inefficiently (but who cares??)
  - Timely Dataflow’s fault tolerance unclear compared to other frameworks
- Basic API is elegant, but unintuitive

10 years on: who *is* using it?

- Has been entirely rewritten in Rust over past 5 years
- Timely dataflow by itself is too low level / too complex for most users
- Ability to build **abstractions** on top of it has become the killer feature
- Frank McSherry is now a founder of materialize.com, "The Streaming Database for Real-time Analytics"
- **Users write normal SQL queries**, which are automatically translated to Timely Dataflow magic

10 years on: who *is* using it?

- Has been entirely rewritten in Rust over past 5 years
- Timely dataflow by itself is too low level / too complex for most users
- Ability to build **abstractions** on top of it has become the killer feature
- Frank McSherry is now a founder of materialize.com, “The Streaming Database for Real-time Analytics”
- **Users write normal SQL queries**, which are automatically translated to Timely Dataflow magic

Materialize raises a $60M Series C, bringing total funding to over $100M

We’re excited to share the news that we have raised $60 million in Series C funding, led by our ne...

Materialize

The Only Platform for Streaming Joins

While other stream processing tools are limited to basic joins, if any, Materialize brings the same powerful join capabilities found in a traditional database to streams of data.

Materialize Join Capabilities:

- **Inner**, **Left (outer)**, **Right**, **Full** and **Cross** Joins
- Multi-way Joins
- **Lateral joins**

View Joins Documentation

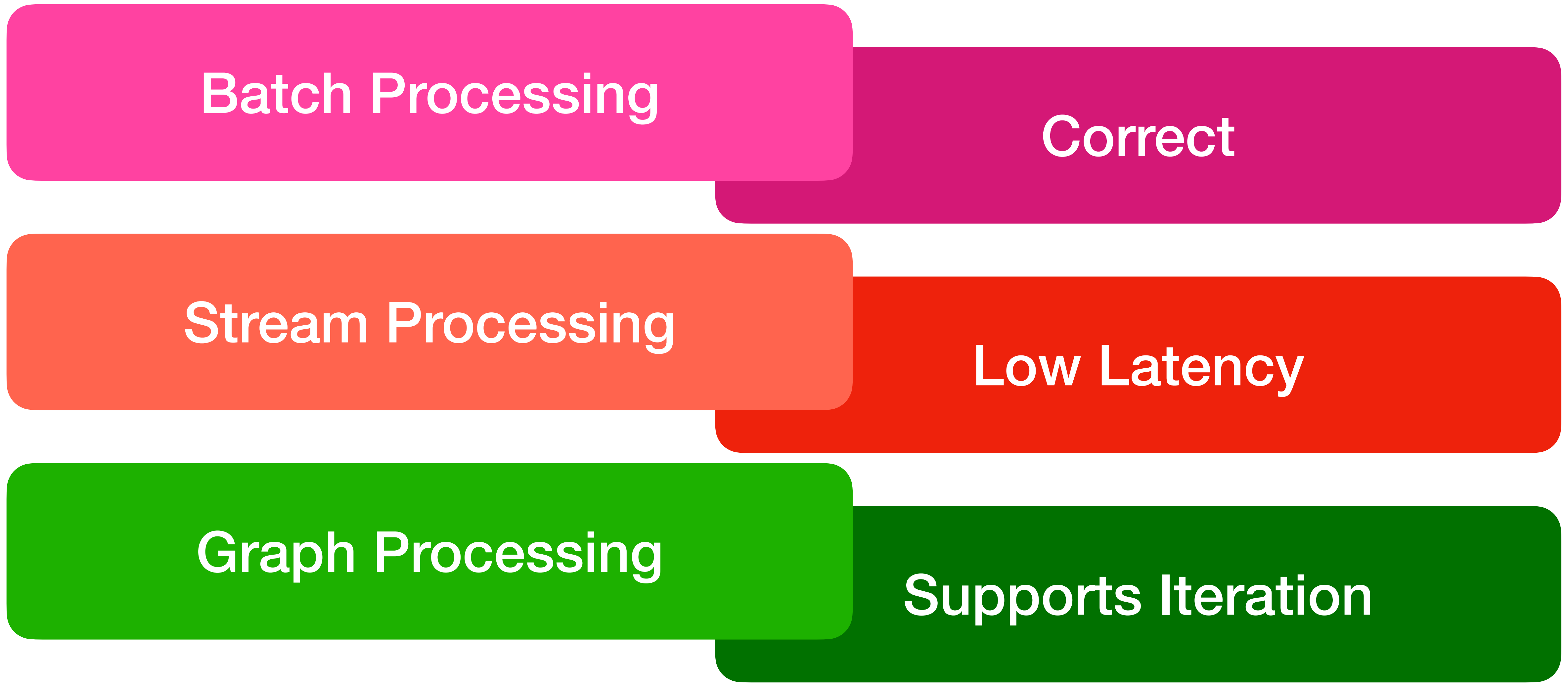
```sql
CREATE MATERIALIZED VIEW user_join AS
  SELECT
    u.id, SUM(p.amount), last_login
  FROM users
  -- Inner join
  JOIN purchases p ON p.user_id=u.id
  -- Left (outer) join + subquery
  LEFT JOIN (
    SELECT user_id, MAX(ts) as last_login
    FROM logins GROUP BY 1
  ) lg ON lg.user_id=u.id
  GROUP BY u.id;
```

In conclusion

- Timely Dataflow is a "shared foundation" for dataflow applications
- Guarantees consistency, low latency, and supports iteration
- A design and engineering feat

However…

- "Killer usecase" is rare
- API is complex, too low-level
  - Materialize, other abstractions address this for specific usecases



Questions?

Recommended Watching

- “Timely Dataflow in three easy steps | Frank McSherry” (https://youtu.be/yOnPmVf4YWo)
- “Naiad: A Timely Dataflow System” (https://youtu.be/yyhMI9r0A9E)
- “It's About Time: An Introduction to Timely Dataflow | Clockworks” (https://youtu.be/ZN7nOwJTSZ0)