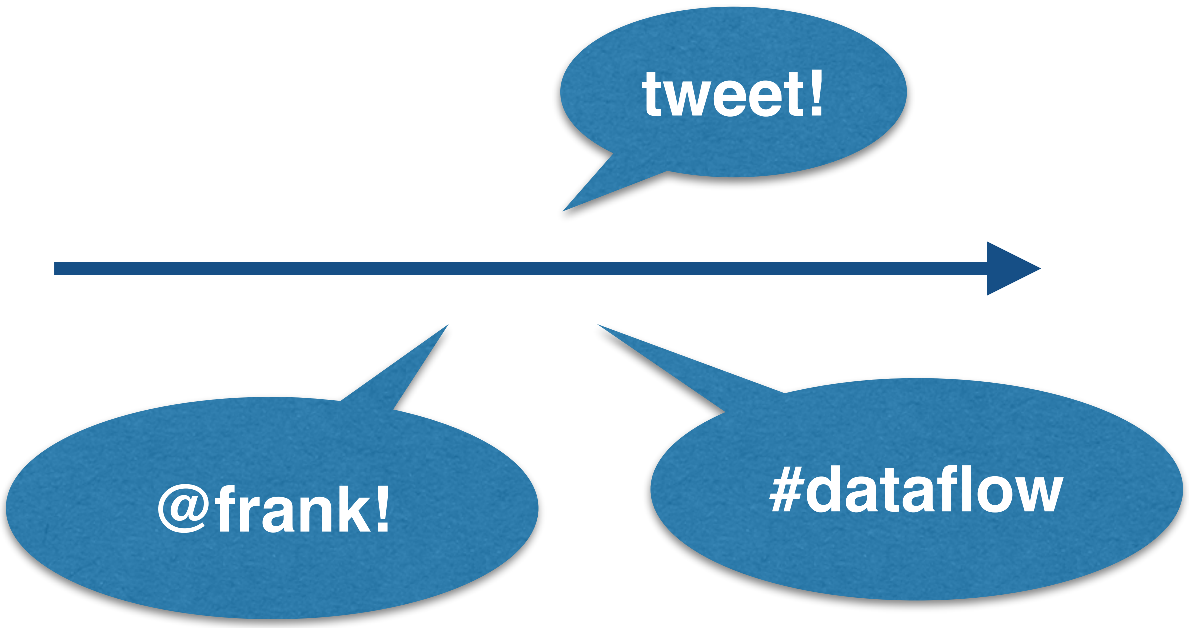
MATERIALIZE

Frank McSherry
mcsherry@materialize.io
@frankmcsherry

A motivating problem



What is the most popular #hashtag?

by component of @mention graph?

in real-time (millisecond latencies)?

Differential dataflow

An expressive programming framework that updates computation when inputs change.

http://github.com/TimelyDataflow/differential-dataflow

[based off of Differential Dataflow, CIDR 2013]

Goal: collection-oriented programming language,
but then allow the collections to change.

People are good at programming with collections.
(at least, better than with streams)

```
fn your_prog: [D] -> [R] = /* .. */;

// Intended experience:
for t in times {
    let output[t] = your_prog(input[t]);
}
```

d_output: Stream<(Data, Time, Diff)>

input streams of changes

```
let nodes = /* pairs (node, bool) */;
let edges = /* pairs (node, node) */;
```

“program” : dataflow assembly

```
nodes.join(&edges)    // one hop neighbors
     .concat(&nodes) // plus original nodes
     .distinct()      // extended neighborhood
```

dataflow execution

```
for t in times {
    nodes.insert(..);
    edges.insert(..);
}
```

```
let nodes = /* pairs (node, bool) */;
let edges = /* pairs (node, node) */;

nodes.join(&edges)    // one hop neighbors
     .concat(&nodes) // plus original nodes
     .distinct()     // extended neighborhood

for t in times {
    nodes.insert(..);
    edges.insert(..);
}
```

```
let nodes = /* pairs (node, bool) */;
let edges = /* pairs (node, node) */;
```

```
nodes.join(&edges)    // one hop neighbors
     .concat(&nodes) // plus original nodes
     .distinct()     // extended neighborhood
```

```
for t in times {
    nodes.insert(..); nodes.remove(..);
    edges.insert(..); edges.remove(..);
}
```

```
let nodes = /* pairs (node, bool) */;
let edges = /* pairs (node, node) */;
```

```
     nodes.join(&edges)    // one hop neighbors
          .concat(&nodes) // plus original nodes
          .distinct()     // extended neighborhood
```

```
for t in times {
    nodes.insert(..); nodes.remove(..);
    edges.insert(..); edges.remove(..);
}
```

```
let nodes = /* pairs (node, bool) */;
let edges = /* pairs (node, node) */;

nodes.iterate(|reach| {

     nodes.join(&edges)   // one hop neighbors
          .concat(&nodes) // plus original nodes
          .distinct()     // extended neighborhood

});

for t in times {
    nodes.insert(..); nodes.remove(..);
    edges.insert(..); edges.remove(..);
}
```

```
let nodes = /* pairs (node, bool) */;
let edges = /* pairs (node, node) */;

nodes.iterate(|reach| {

    reach.join(&edges)   // one hop neighbors
         .concat(&nodes) // plus original nodes
         .distinct()     // extended neighborhood

});

for t in ti
    nodes.i
    edges.i
}
```

Stream<((node, bool), (Time, u64), int)>

Secret sauce: Incremental computation done with respect to a partial order, rather than a total order.

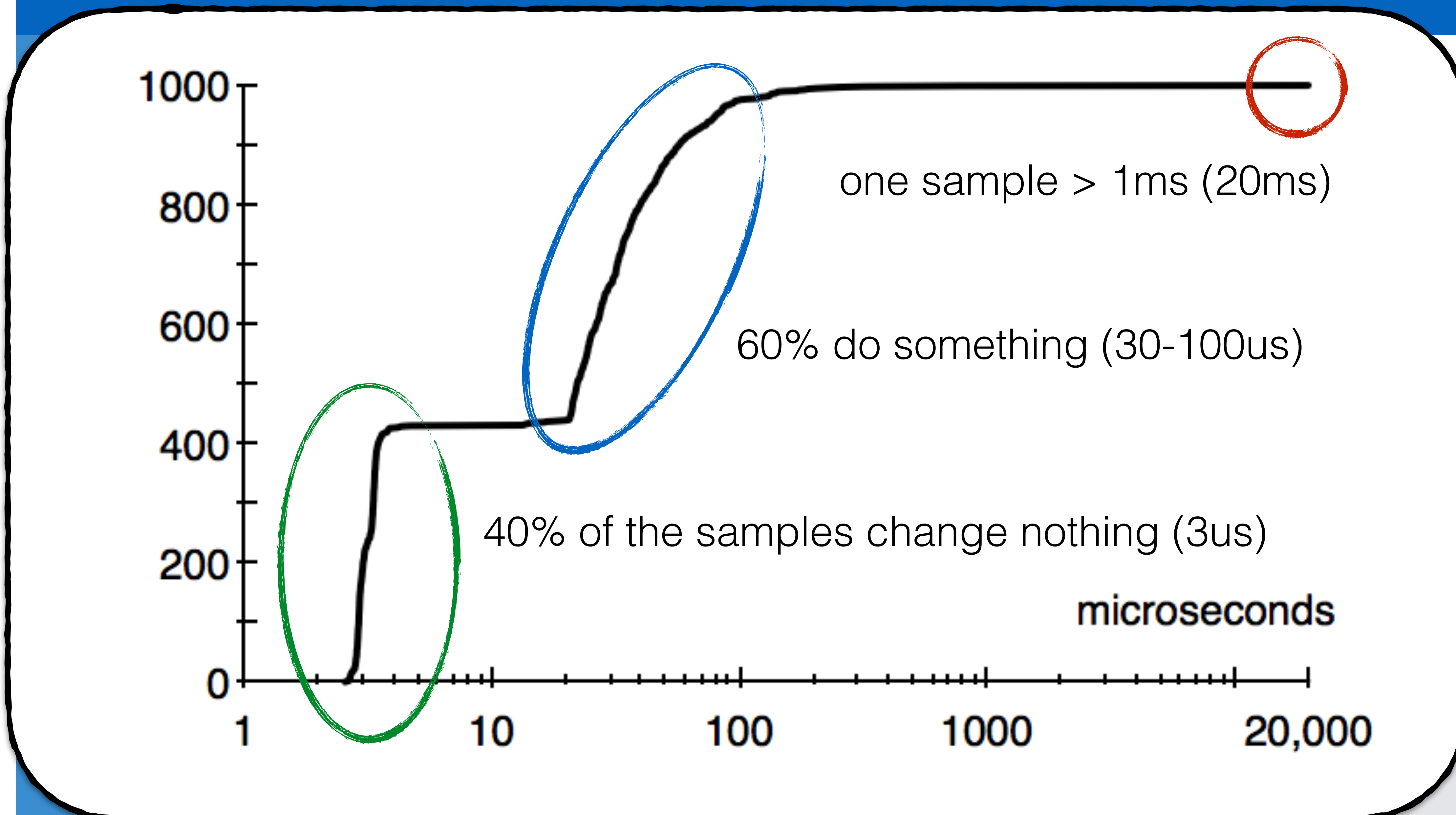
collection(t) = sum_{s ≤ t} differences(s)

Reach	cores	livejournal	orkut
GraphX	128	36s	48s
SociaLite	128	52s	67s
Myria	128	5s	6s
BigDatalog	128	17s	20s

[BigDatalog, SIGMOD 2016]

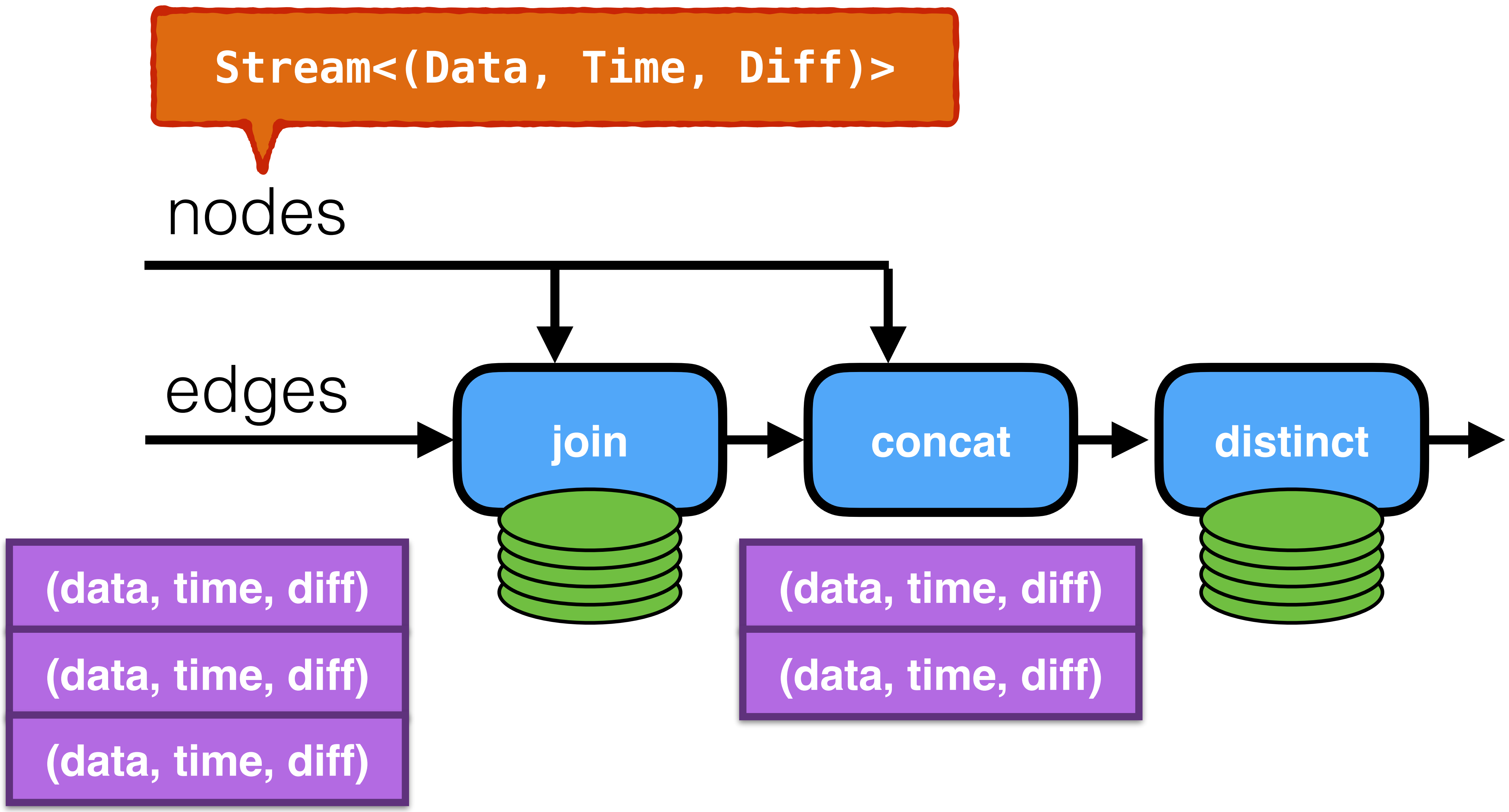
Reach	cores	livejournal	orkut
GraphX	128	36s	48s
SociaLite	128	52s	67s
Myria	128	5s	6s
BigDatalog	128	17s	20s
Differential	1	**7s**	**15s**

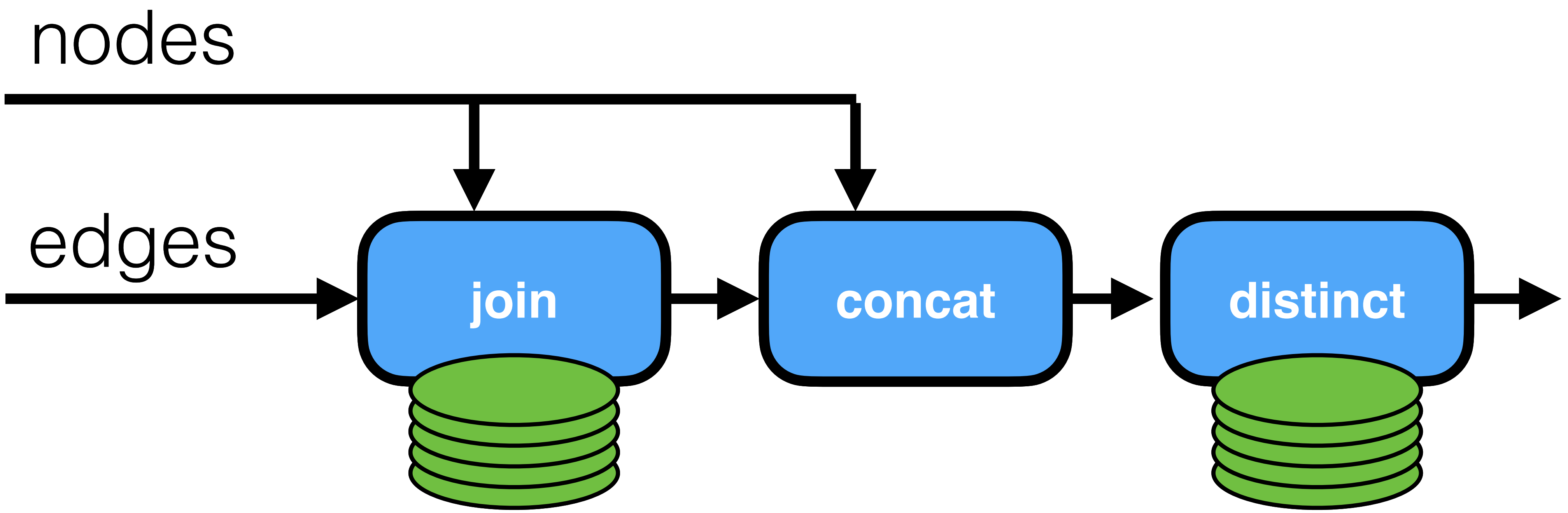
Reach	cores	livejournal	orkut

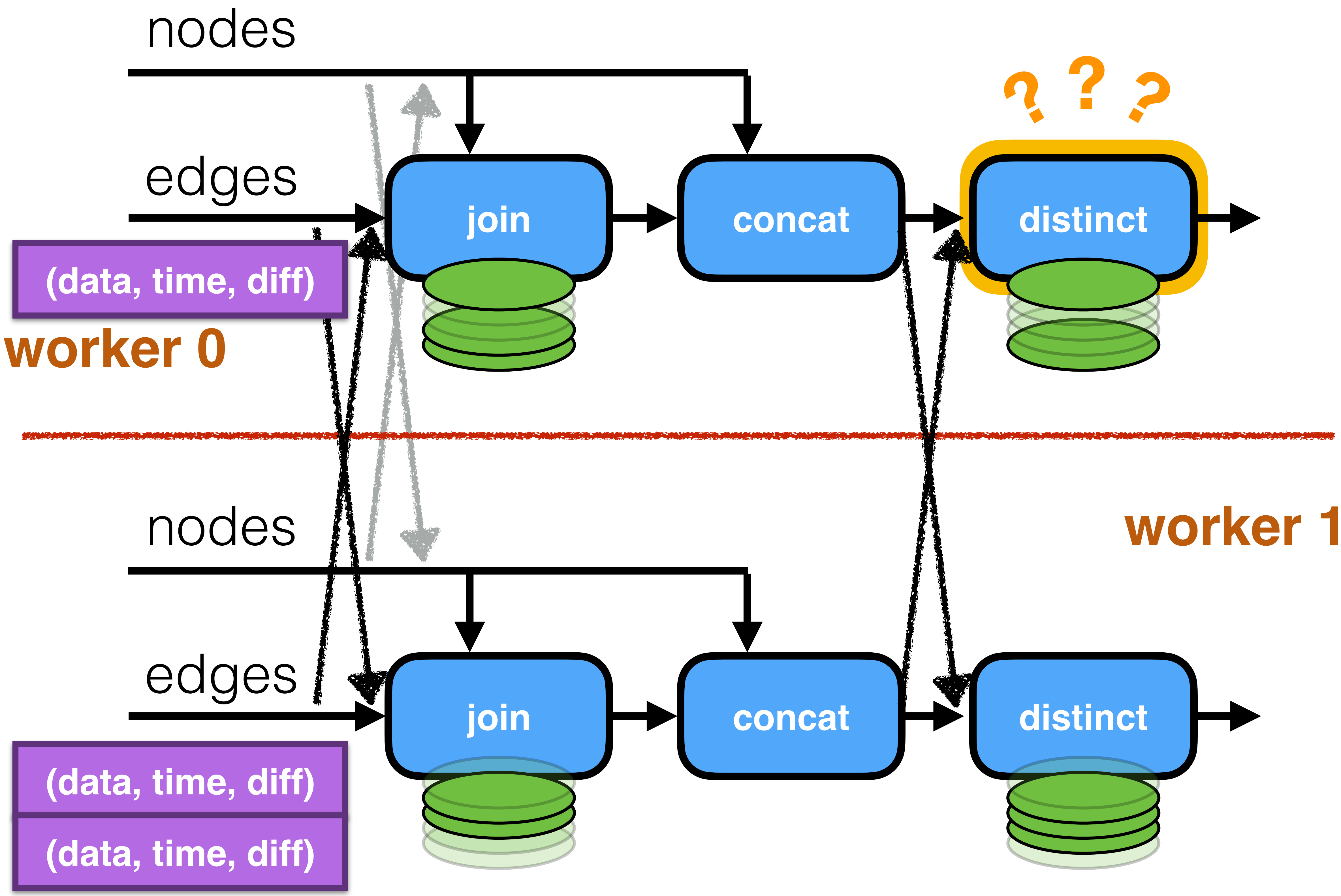


```
let nodes = /* pairs (node, bool) */;
let edges = /* pairs (node, node) */;

nodes.iterate(|reach| {

     reach.join(&edges)    // one hop neighbors
          .concat(&nodes) // plus original nodes
          .distinct()     // extended neighborhood

});

for t in times {
    nodes.insert(..);
    edges.remove(..);
}
```

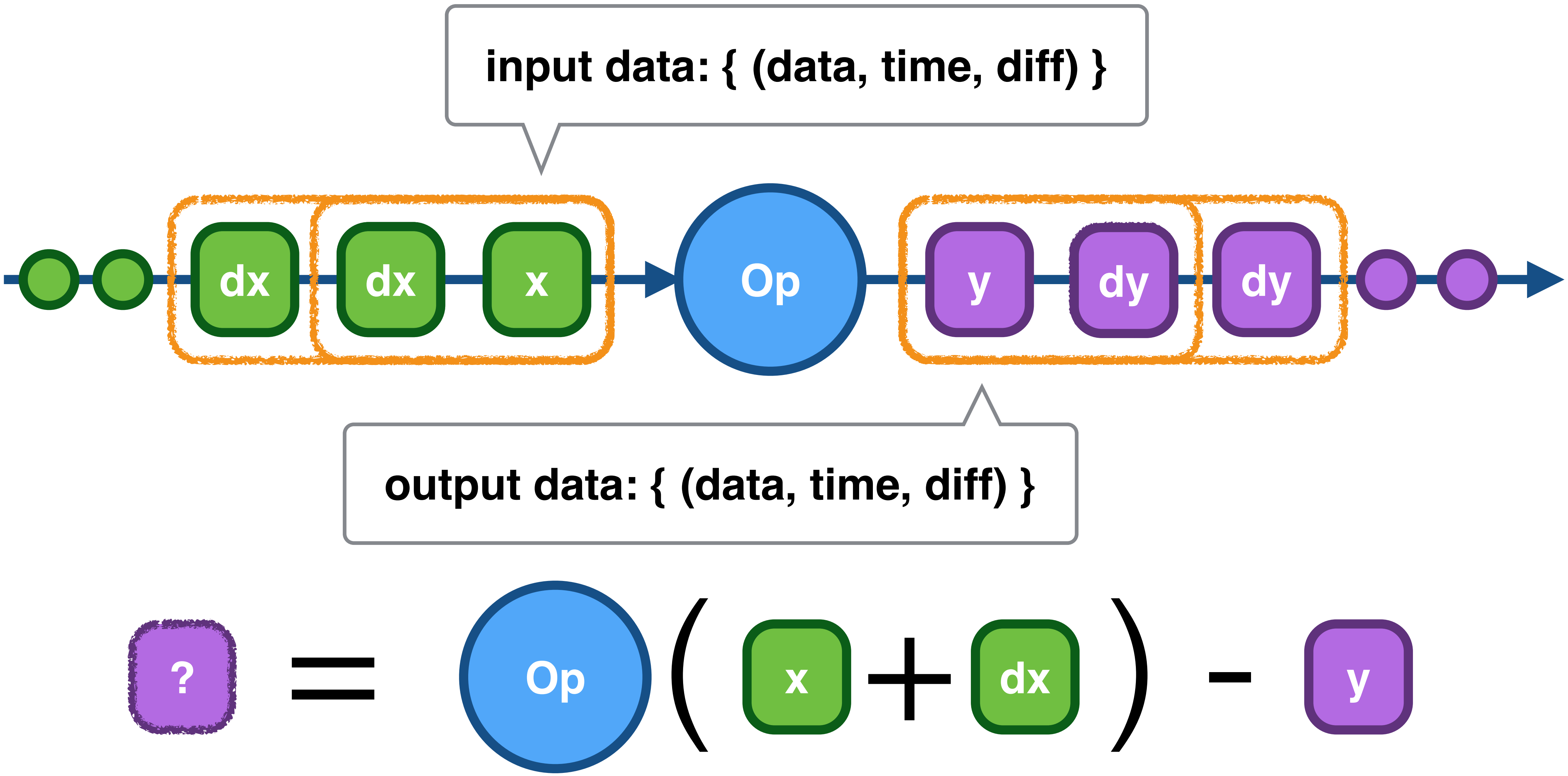
```
let nodes = /* pairs (node, node) */;
let edges = /* pairs (node, node) */;

nodes.iterate(|reach| {

    reach.join(&edges)   // one hop neighbors
         .concat(&nodes) // plus original nodes
         .min()          // smallest labels

});

for t in times {
    nodes.insert(..);
    edges.remove(..);
}
```


Stream<(Data, Time, Diff)>
nodes
edges
join
concat
distinct
(data, time, diff)
(data, time, diff)
(data, time, diff)
(data, time, diff)
(data, time, diff)


nodes
edges
join
concat
distinct

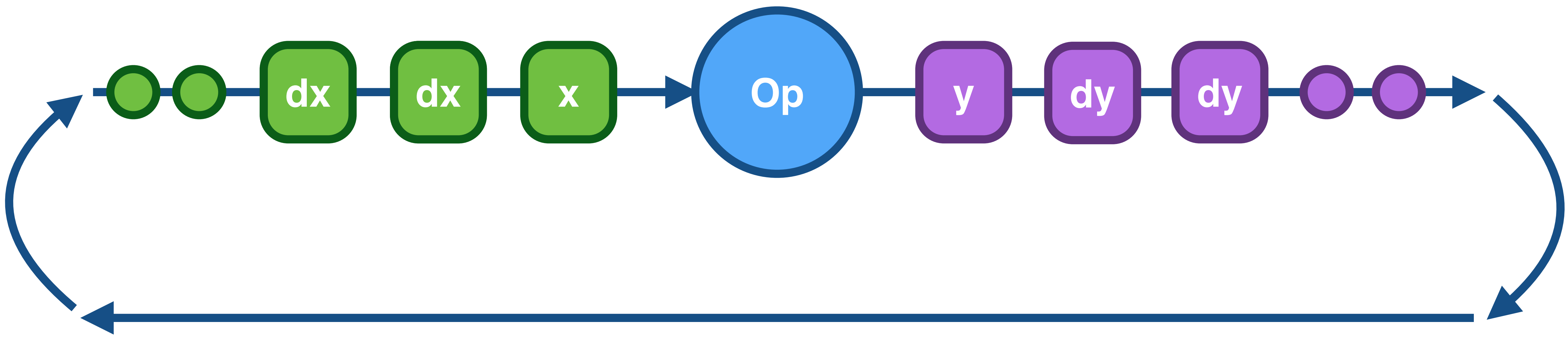

nodes
edges
(data, time, diff)
worker 0
join
concat
distinct
? ? ?
nodes
edges
worker 1
join
concat
distinct
(data, time, diff)
(data, time, diff)

Incremental Dataflow

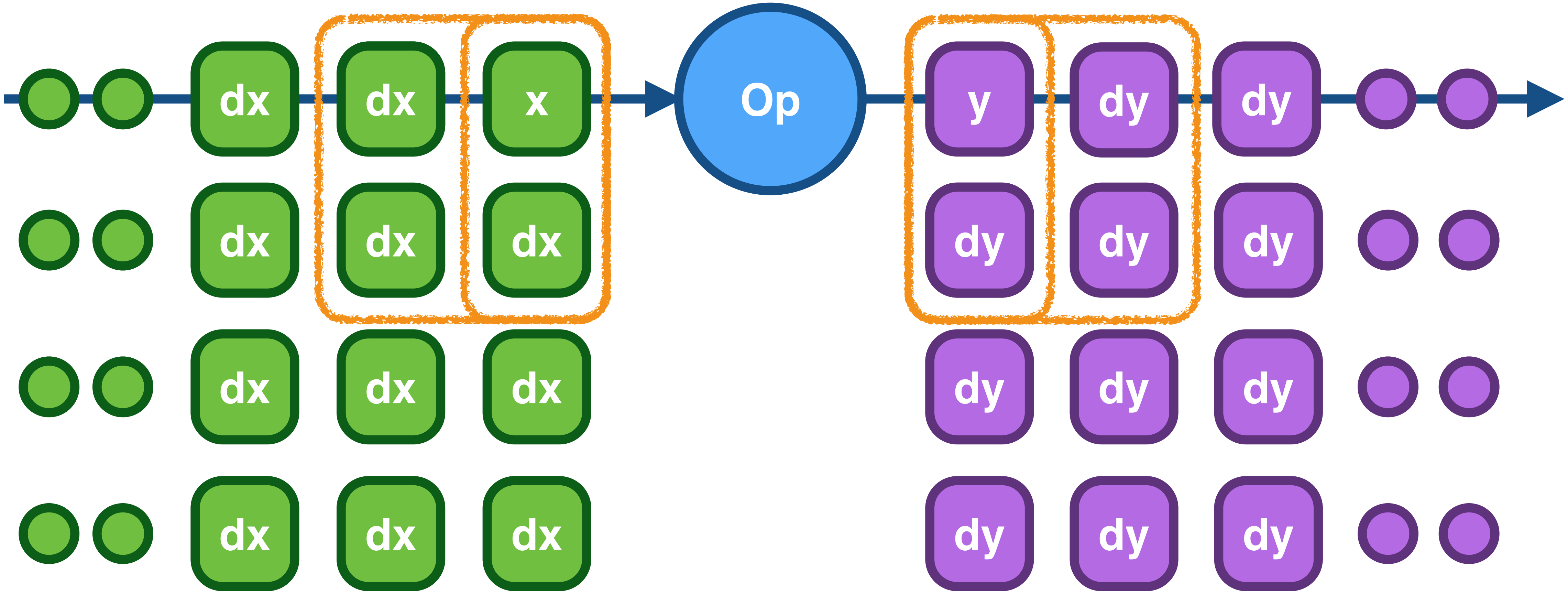


Iterative Dataflow

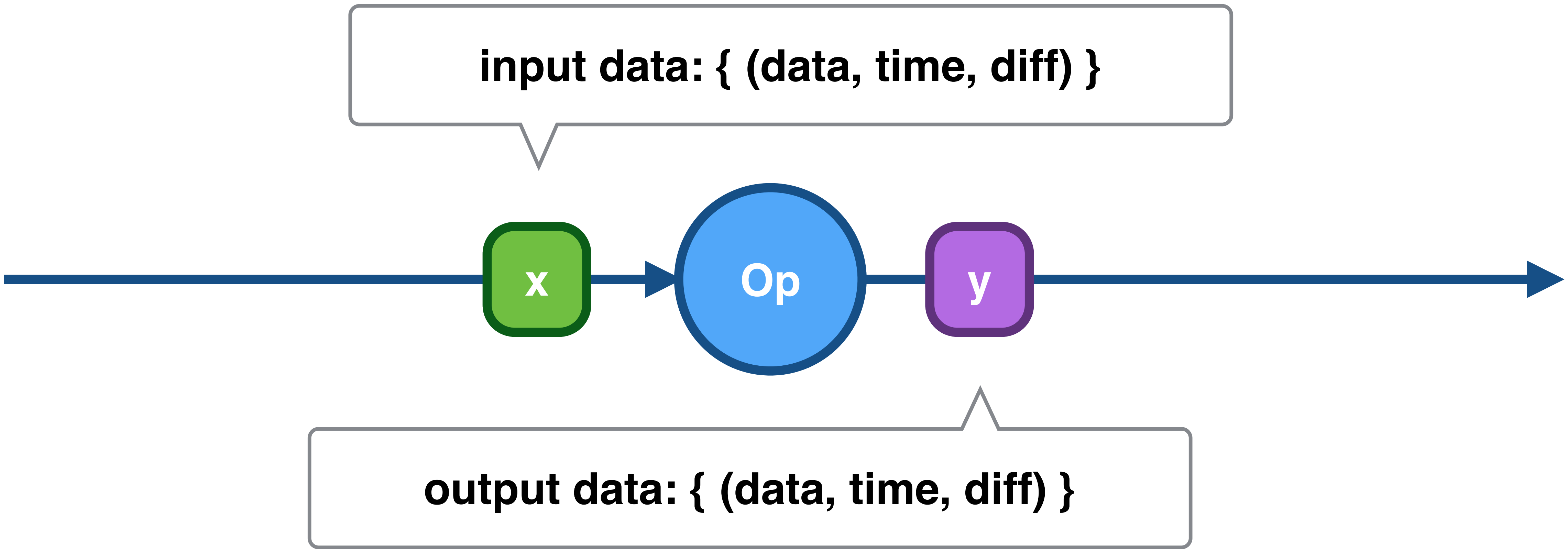
e.g. semi-naive bottom-up datalog



Differential Dataflow



Differential Dataflow

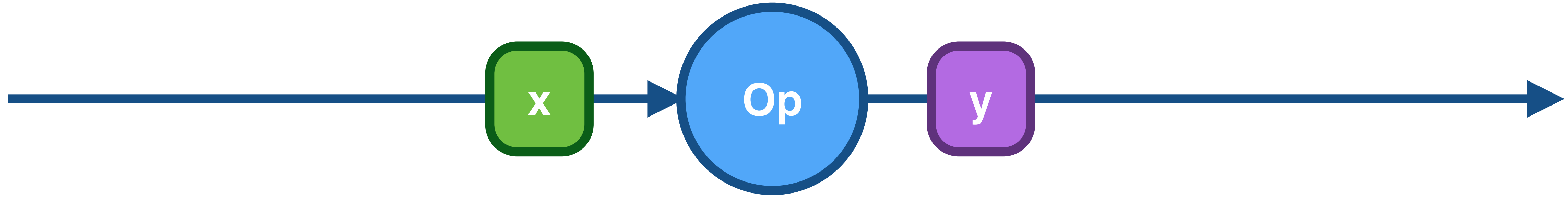


Important: times are only **partially** ordered

Differentiation on a discrete partial order

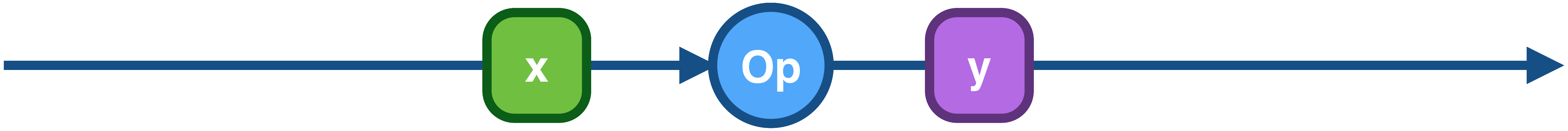
Differential Dataflow

with data-parallel operators



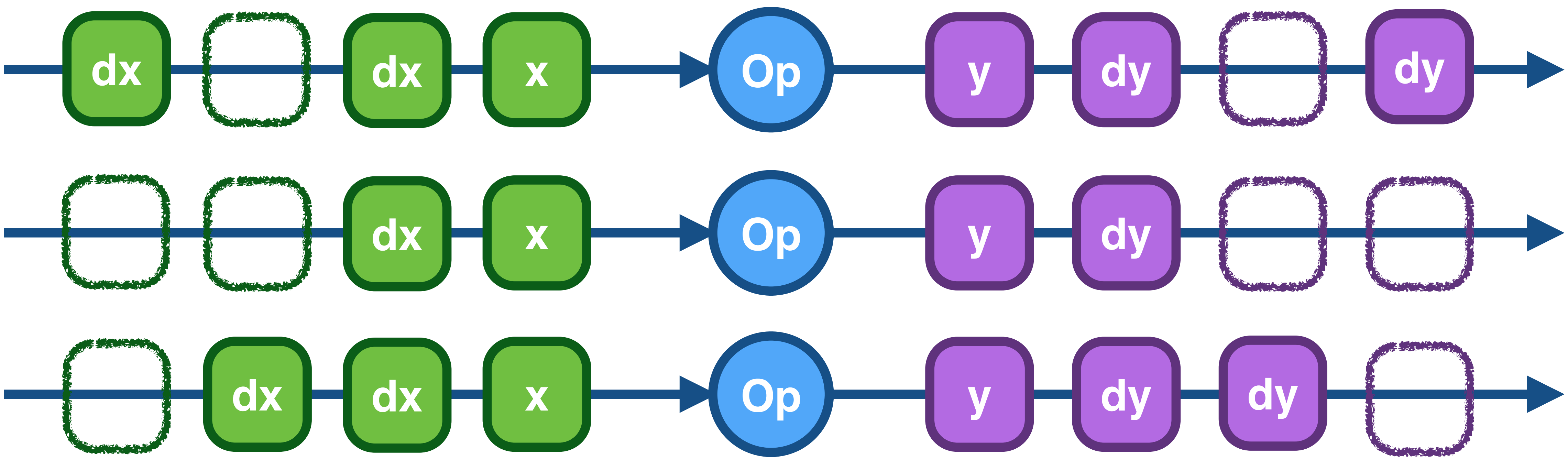
Differential Dataflow

with data-parallel operators



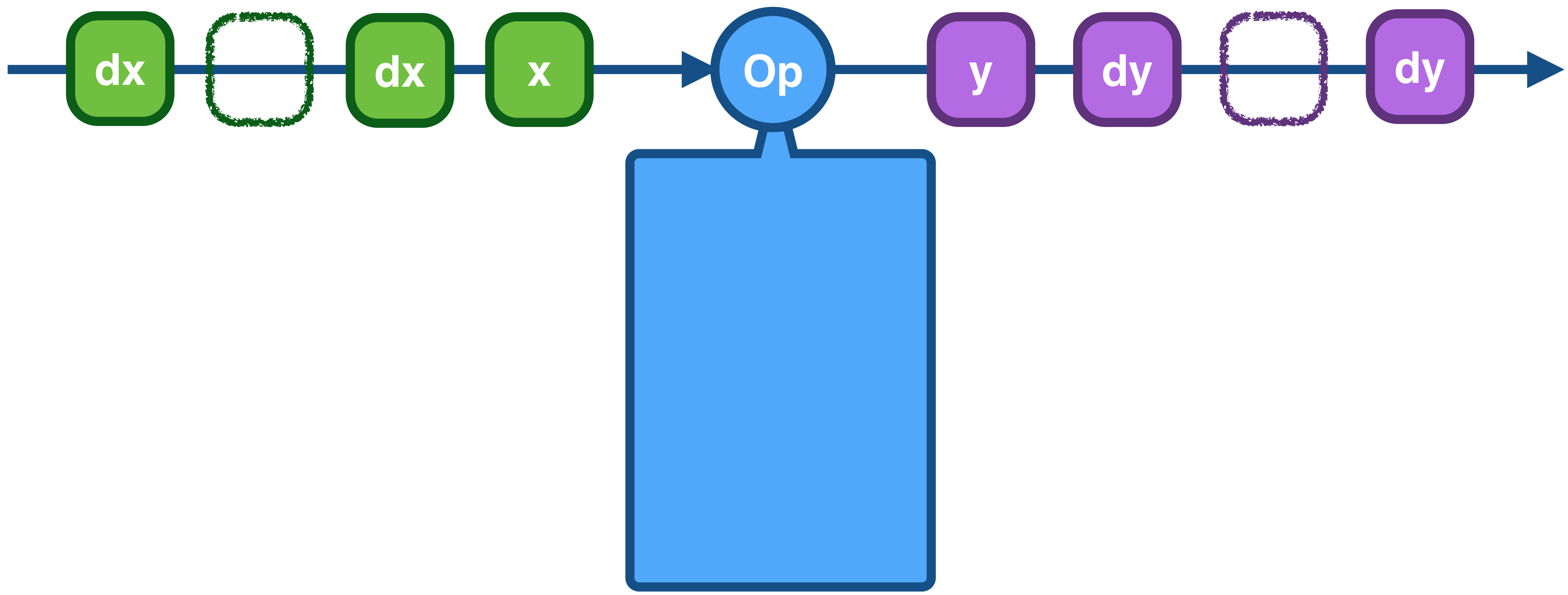
Differential Dataflow

with data-parallel operators



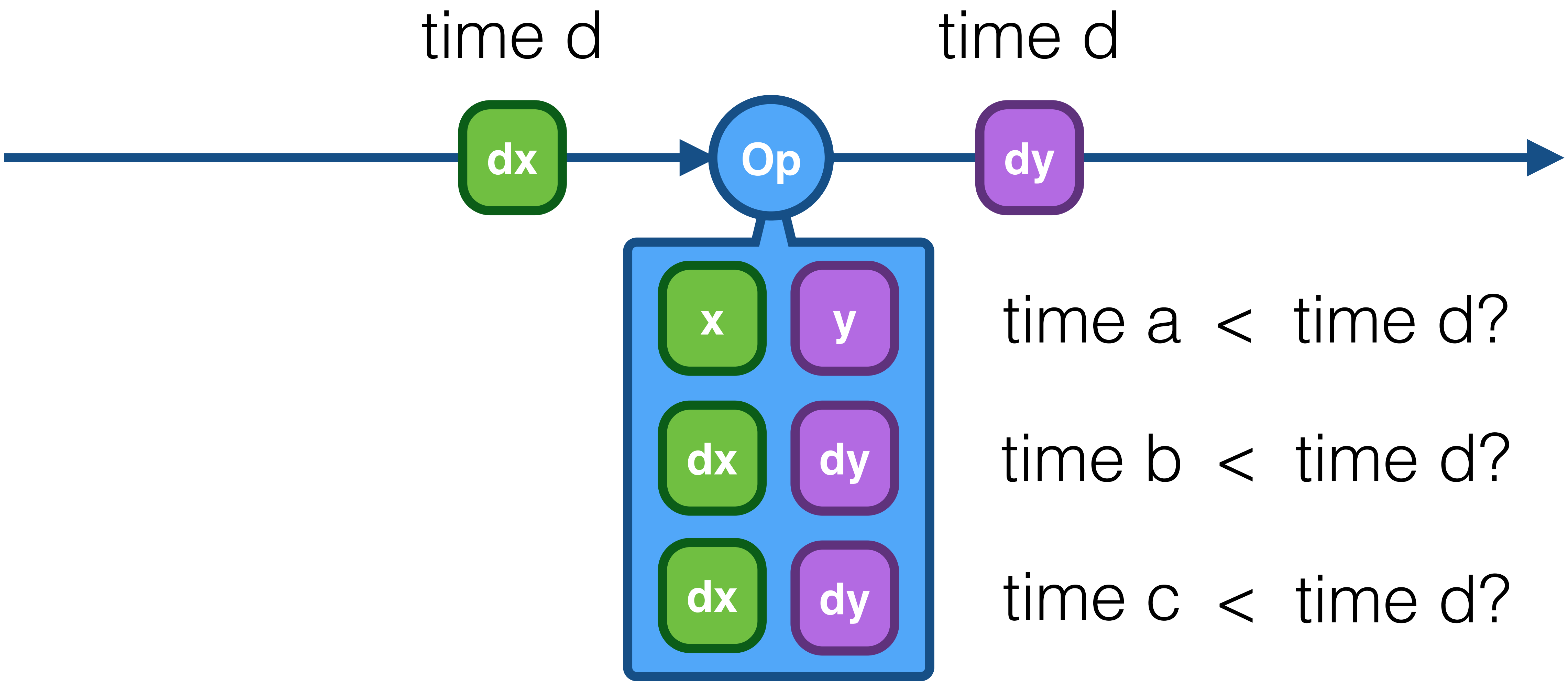
Differential Dataflow

with data-parallel operators



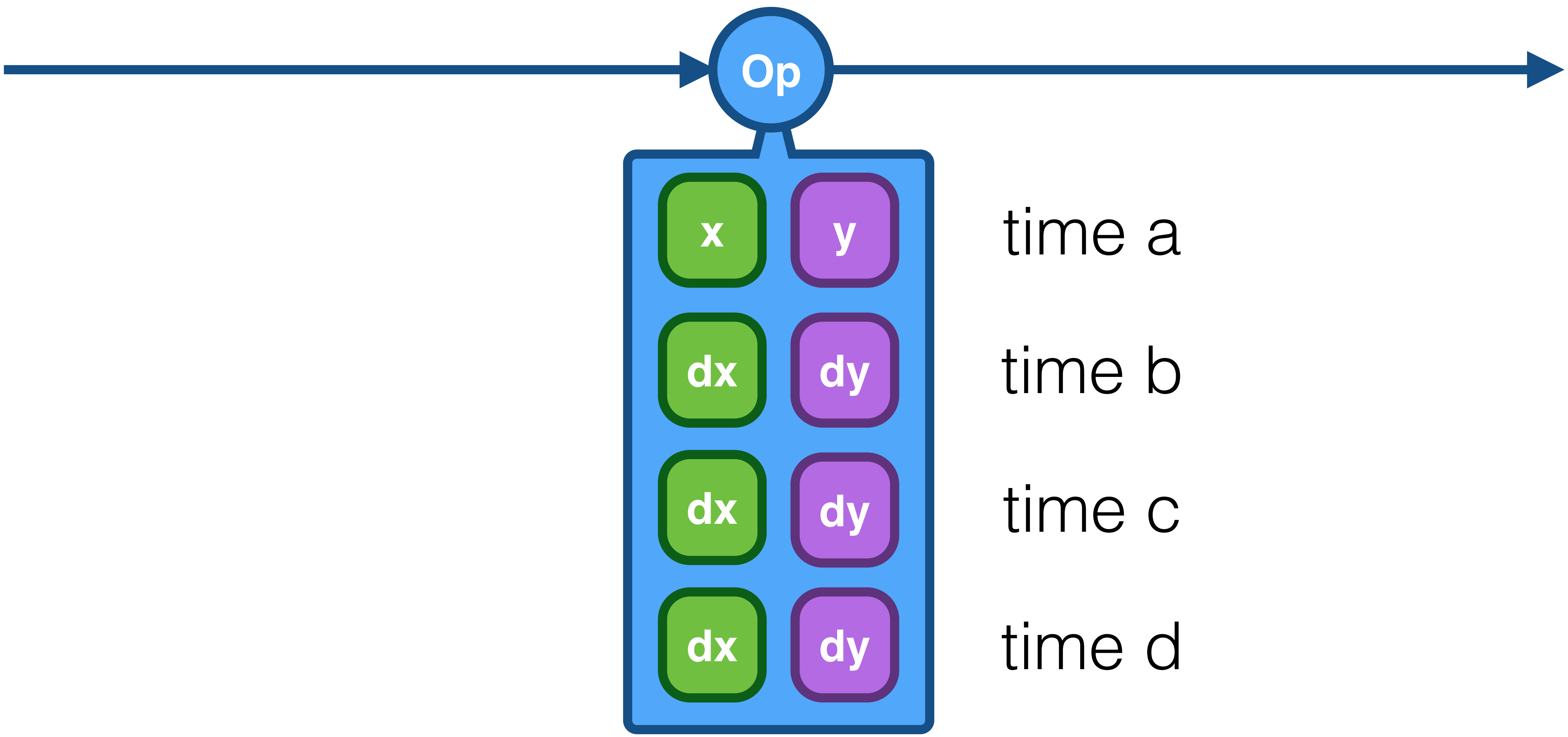
Differential Dataflow

with data-parallel operators

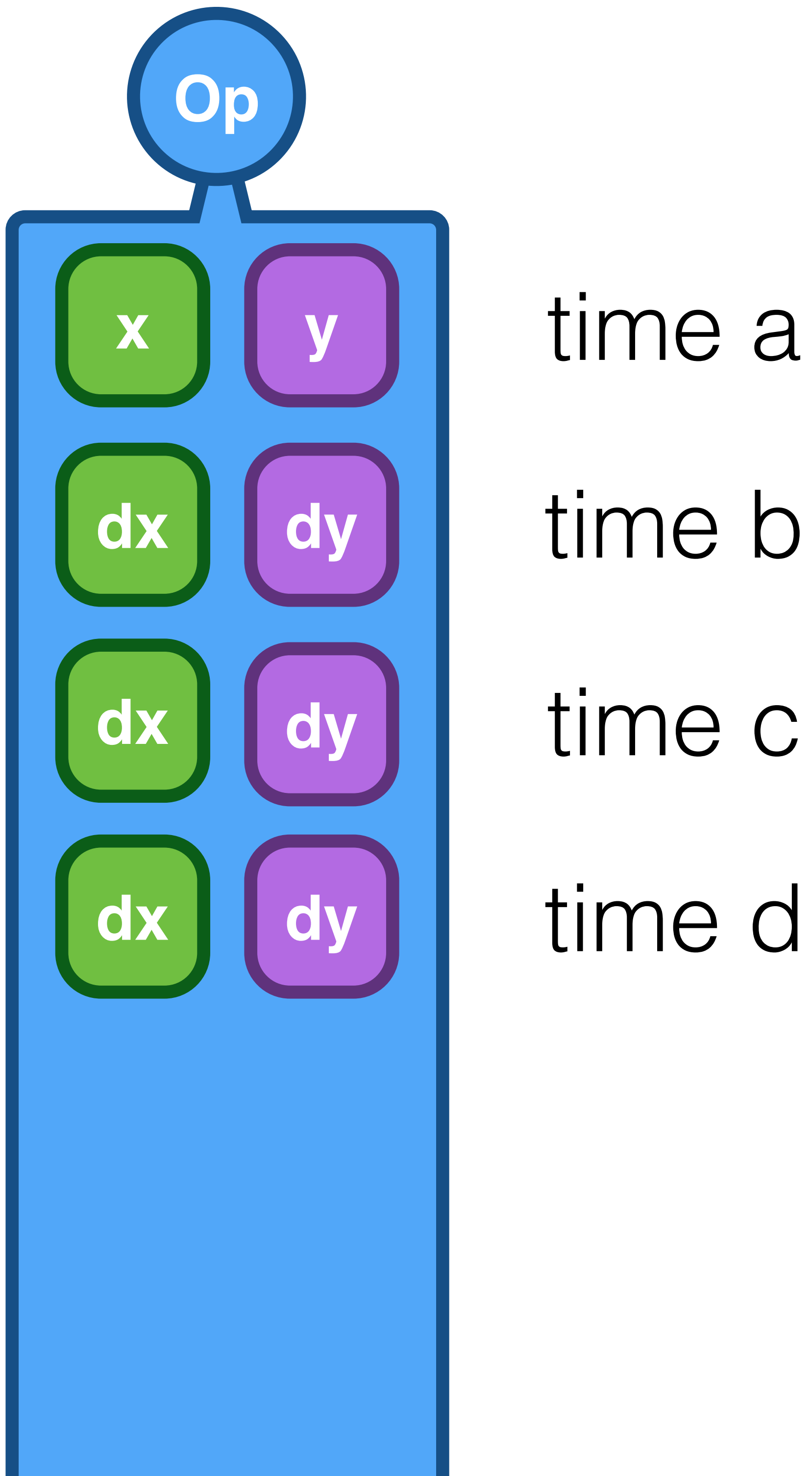


Differential Dataflow

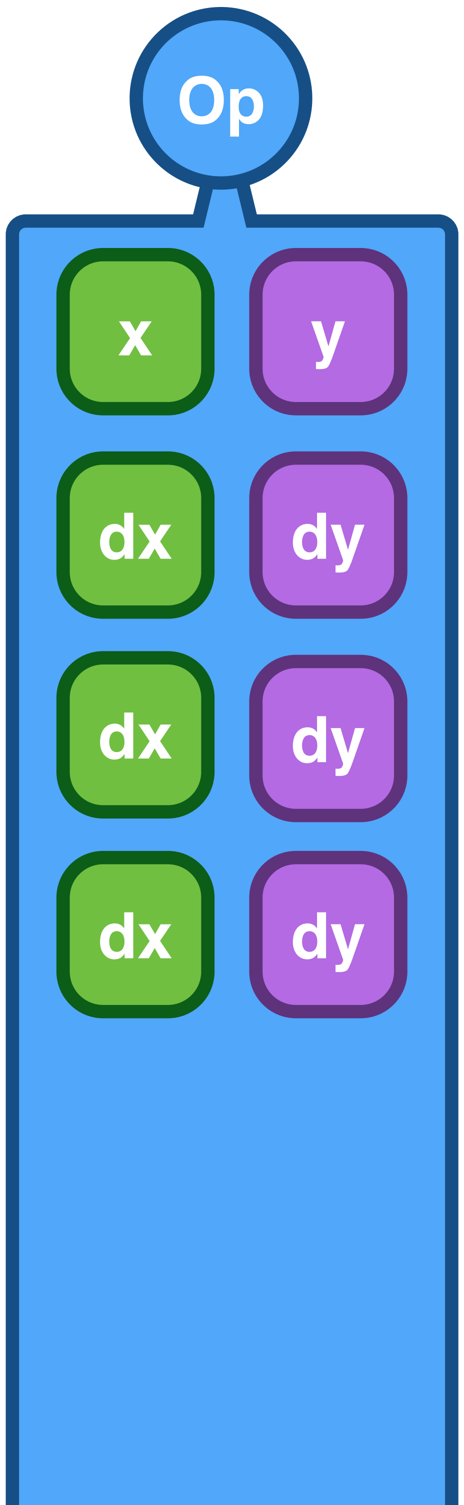
with data-parallel operators



Diversion: Compaction



Diversion: Compaction



time a

time b

time c

time d

Let ***F*** be a set of times lower-bounding those times we might see in the future.

$$t_1 \equiv_F t_2 \;\; \text{when} \;\; \forall_{f \geq F}(t_1 \leq f \text{ iff } t_2 \leq f)$$

Each time ***t*** has a “representative” from this equivalence class, computed as

$$rep_F(t) := \vee_{f \in F}(t \wedge f)$$

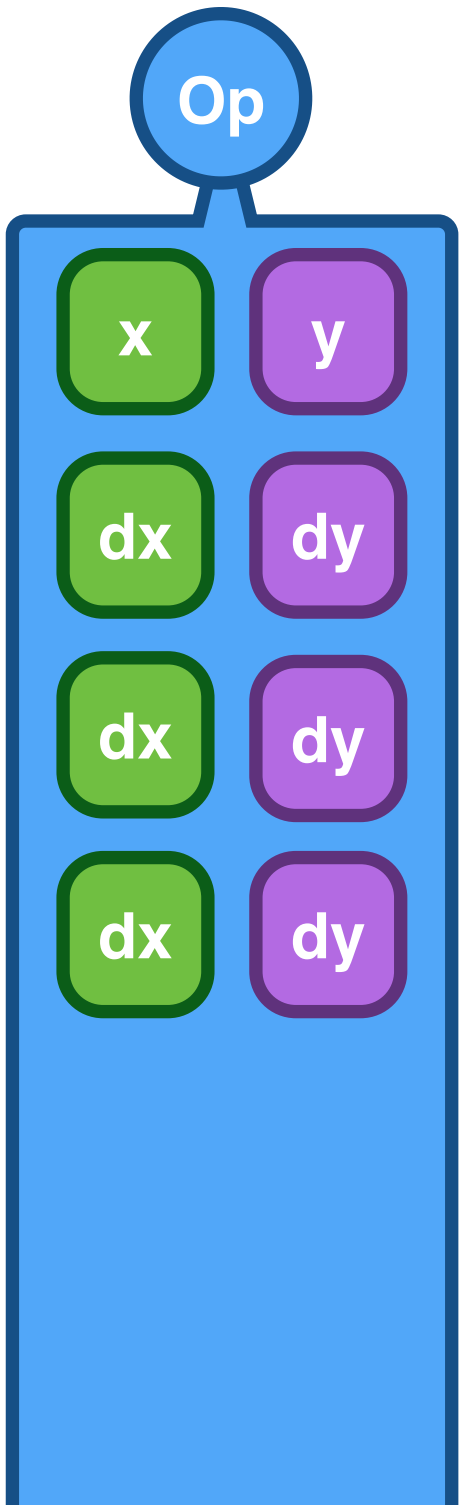
THEOREM A.1 (CORRECTNESS). *For any lattice element t and set F of lattice elements, $t \equiv_F rep_F(t)$.*

PROOF. We prove both directions of the implication in $\equiv_F$ separately, for all $f \geq F$. First assume $t \leq f$. By assumption, f is greater than some element f' of F, and so $t \wedge f' \leq f$ by the (*lub*) property. As a lower bound, $rep_F(t) \leq t \wedge f'$ for each $f' \in F$, and by transitivity $rep_F(t) \leq f$. Second assume $rep_F(t) \leq f$. Because $t \leq (t \wedge f')$ for all $f' \in F$, then $t \leq rep_F(t)$ by the (*glb*) property and by transitivity $t \leq f$. □

THEOREM A.2 (OPTIMALITY). *For any lattice elements* t_1 *and* t_2 *and set* F *of lattice elements, if* $t_1 \equiv_F t_2$ *then* $rep_F(t_1) = rep_F(t_2)$.

PROOF. For all $f \in F$ we have both that $t_1 \leq t_1 \wedge f$ and $f \leq t_1 \wedge f$, the latter implying that $t_1 \wedge f \geq F$. By our assumption, t_2 agrees with t_1 on times greater than F, making $t_2 \leq t_1 \wedge f$ for all $f \in F$. By correctness, $rep_F(t_2)$ agrees with t_2 on times greater than F, which includes $t_1 \wedge f$ for $f \in F$ and so $rep_F(t_2) \leq t_1 \wedge f$ for all $f \in F$. Because $rep_F(t_2)$ is less or equal to each term in the greatest lower bound definition of $rep_F(t_1)$, it is less or equal to $rep_F(t_1)$ itself. The symmetric argument proves that $rep_F(t_1) \leq rep_F(t_2)$, which implies that the two are equal (by antisymmetry). □

Diversion: Compaction



time a

time b

time c

time d

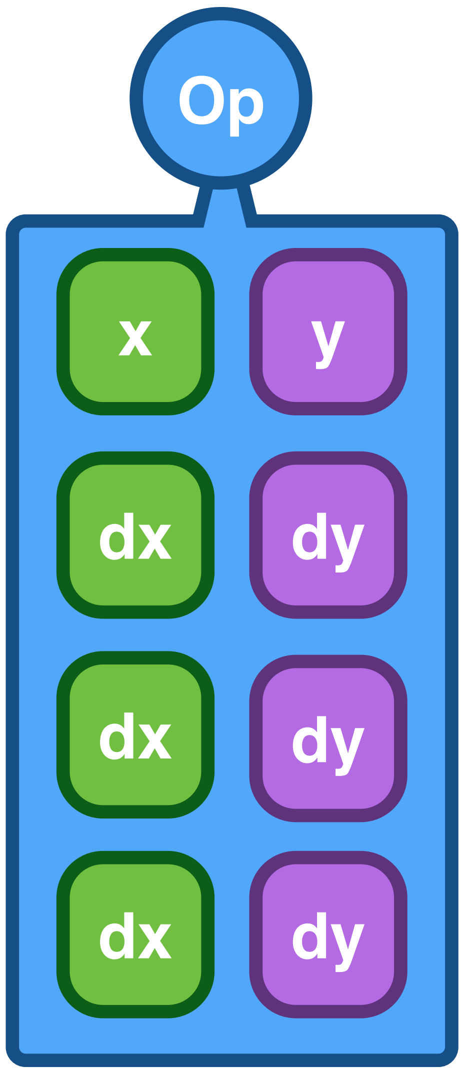
Let ***F*** be a set of times lower-bounding those times we might see in the future.

$$t_1 \equiv_F t_2 \ \text{ when } \ \forall_{f \geq F}(t_1 \leq f \text{ iff } t_2 \leq f)$$

Each time ***t*** has a "representative" from this equivalence class, computed as

$$rep_F(t) := \vee_{f \in F}(t \wedge f)$$

Diversion: Compaction



time a

time b

time c

time d

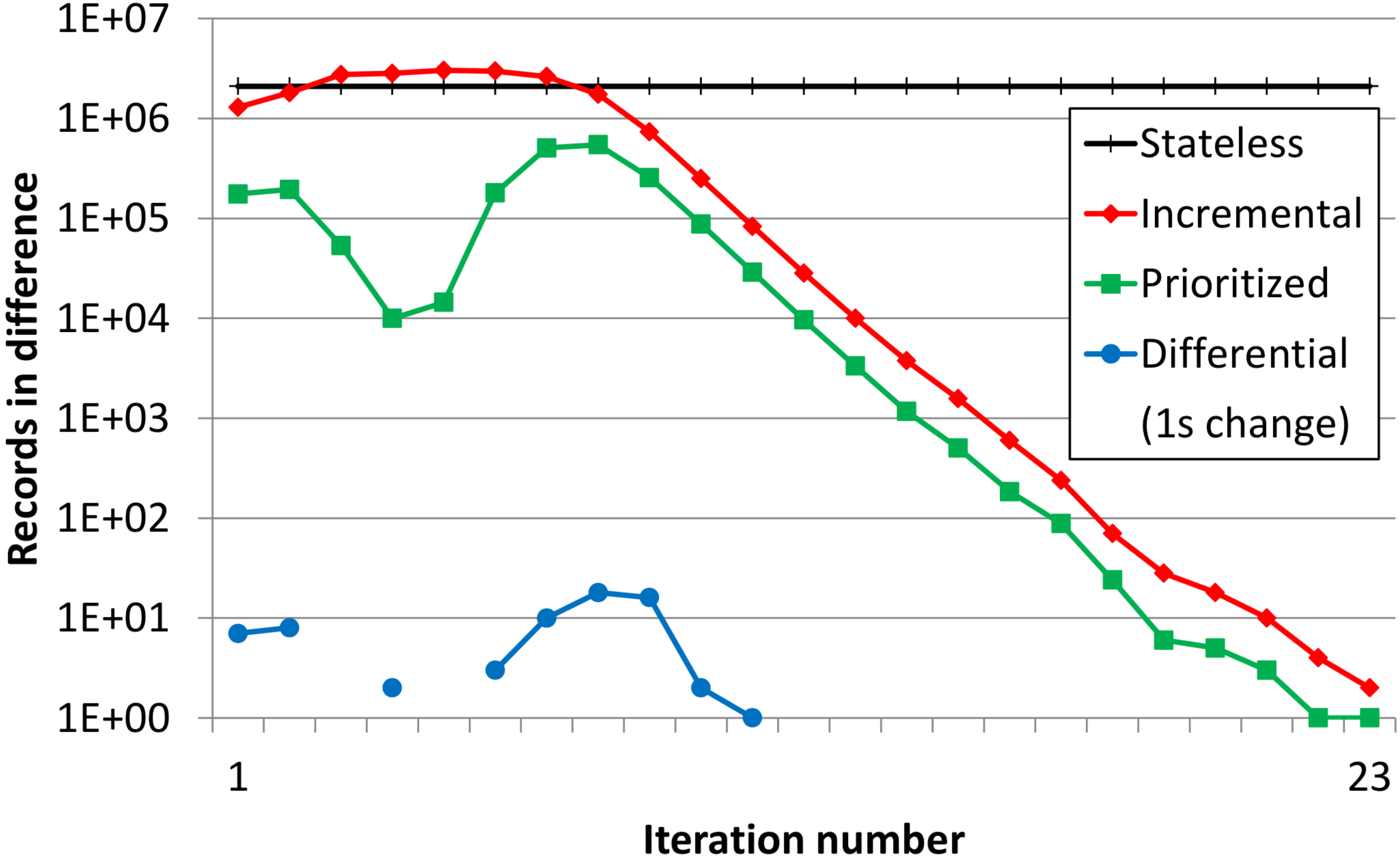
Let ***F*** be a set of times lower-bounding those times we might see in the future.

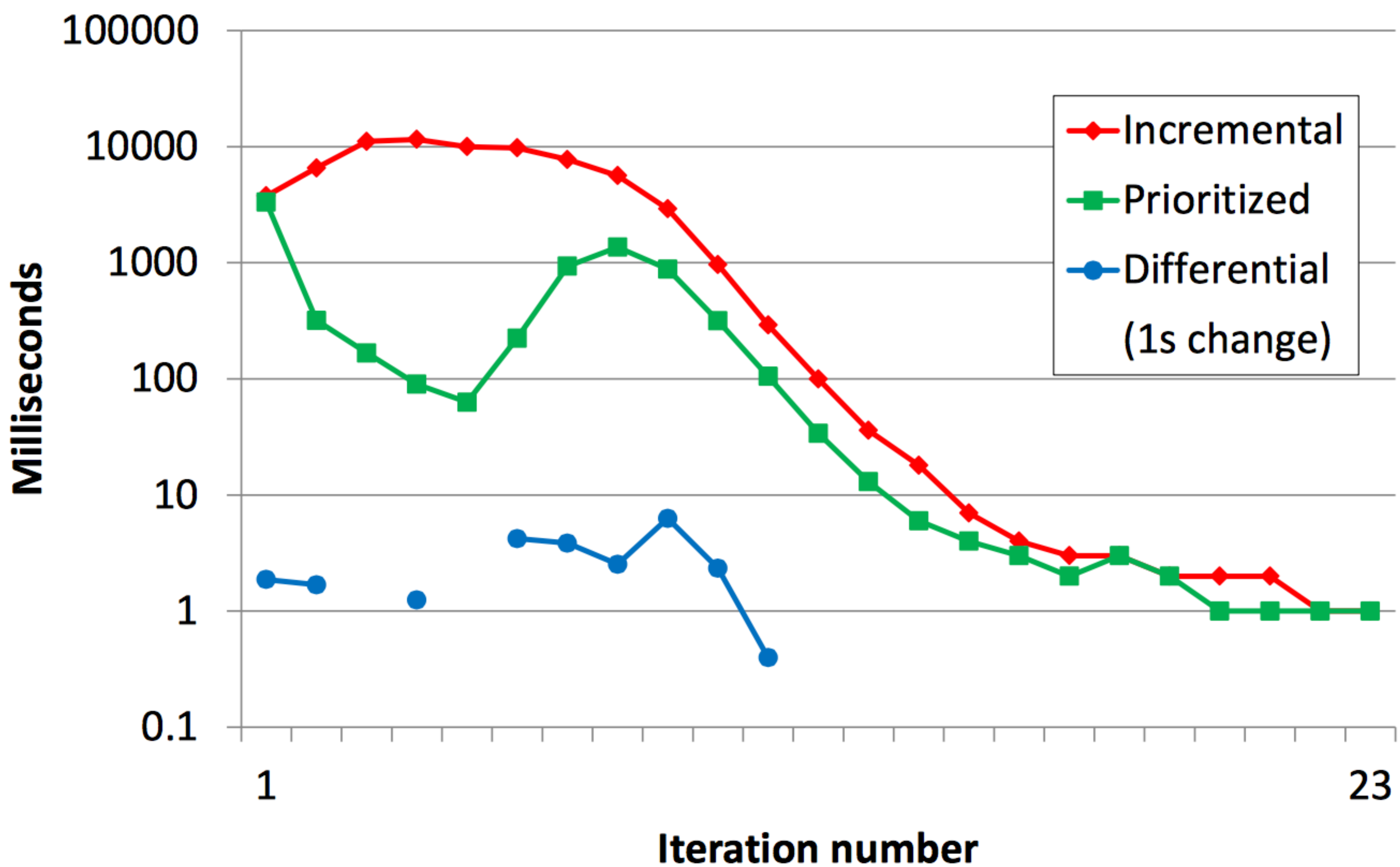
$$t_1 \equiv_F t_2 \;\; \text{when} \;\; \forall_{f \geq F}(t_1 \leq f \text{ iff } t_2 \leq f)$$

Each time ***t*** has a "representative" from this equivalence class, computed as

$$rep_F(t) := \vee_{f \in F}(t \wedge f)$$

Connected components on the Twitter @mention graph


1E+07
1E+06
1E+05
1E+04
1E+03
1E+02
1E+01
1E+00
Records in difference
1
23
Iteration number
Stateless
Incremental
Prioritized
Differential
(1s change)



100000
10000
1000
100
10
1
0.1
Milliseconds
1
23
Iteration number
Incremental
Prioritized
Differential
(1s change)

Pause

Things left unsaid

Arrangements
Operator state can be shared with others.
Credit due to: “declarative programming”.

Open- v. Closed-loop
The world changes even if we are not ready.
Systems should handle batches of changes.

Groups and Semi-groups
All “Diffs” can be arbitrary (semi-)groups.
Types encode the nature of changes.

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

/timely-dataflow

/abomonation

/diagnostics

MATERIALIZE

https://materialize.com

@frankmcsherry