CS 591 K1:
Data Stream Processing and Analytics

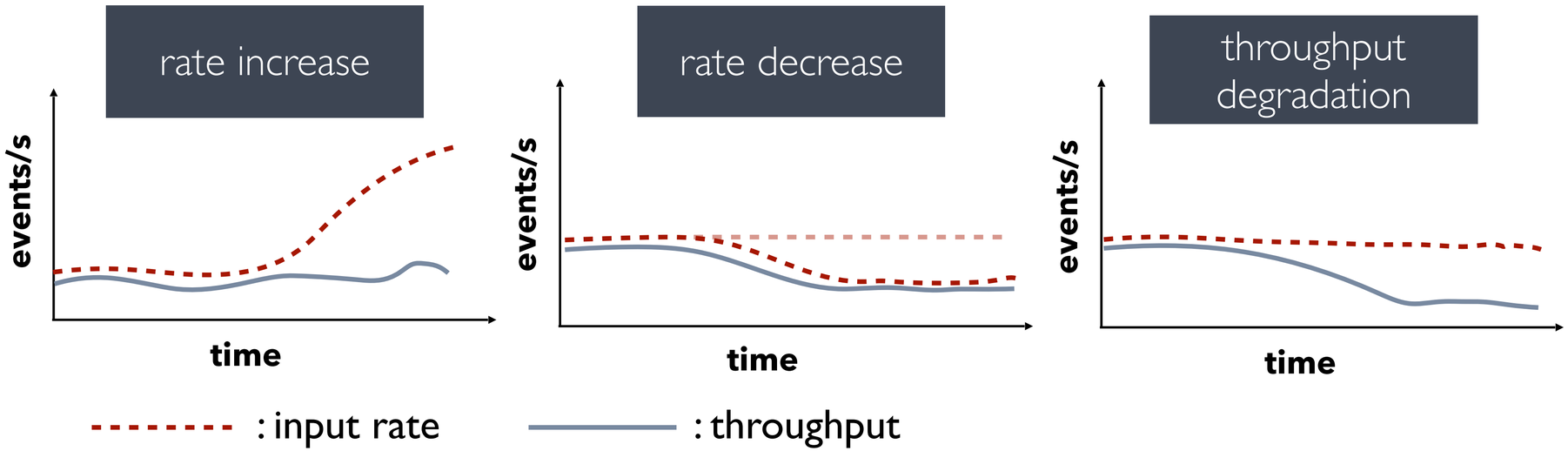
Spring 2021

Elasticity policies and state migration

Vasiliki (Vasia) Kalavri
vkalavri@bu.edu

Streaming applications are long-running

- Workload will change
- Conditions might change
- State is accumulated over time



Control: When and how much to adapt?

- Detect environment changes: external workload and system performance
- Identify bottleneck operators, straggler workers, skew
- Enumerate scaling actions, predict their effects, and decide which and when to apply

Mechanism: How to apply the re-configuration?

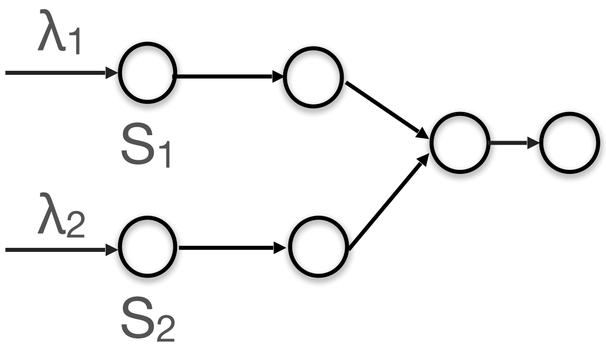
- Allocate new resources, spawn new processes or release unused resources, safely terminate processes
- Adjust dataflow channels and network connections
- Re-partition and migrate state in a consistent manner
- Block and unblock computations to ensure result correctness

Automatic Scaling Control

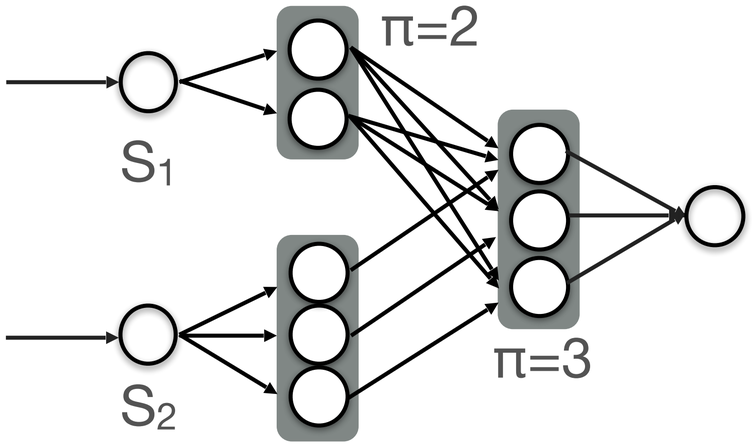
Elastic streaming systems

Adaptive resource scaling according to the workload

logical dataflow



physical dataflow



Given a logical dataflow with sources S_1, S_2, … S_n and rates λ_1, λ_2, … λ_n identify the **minimum parallelism π_i** per operator i, such that the physical dataflow can sustain all source rates.

Scaling Controller

Detect symptoms

metrics

Service time, waiting time, CPU utilization, congestion, back-pressure, throughput

Decide whether to scale

policy

Predictive: Queuing theory, control theory, analytical dataflow-based models

Heuristic: Rule-based models, e.g. if CPU utilization > 70% => scale out

Decide how much to scale

scaling action

Dataflow-wide: at-once for all operators

Speculative: small changes at one operator at a time

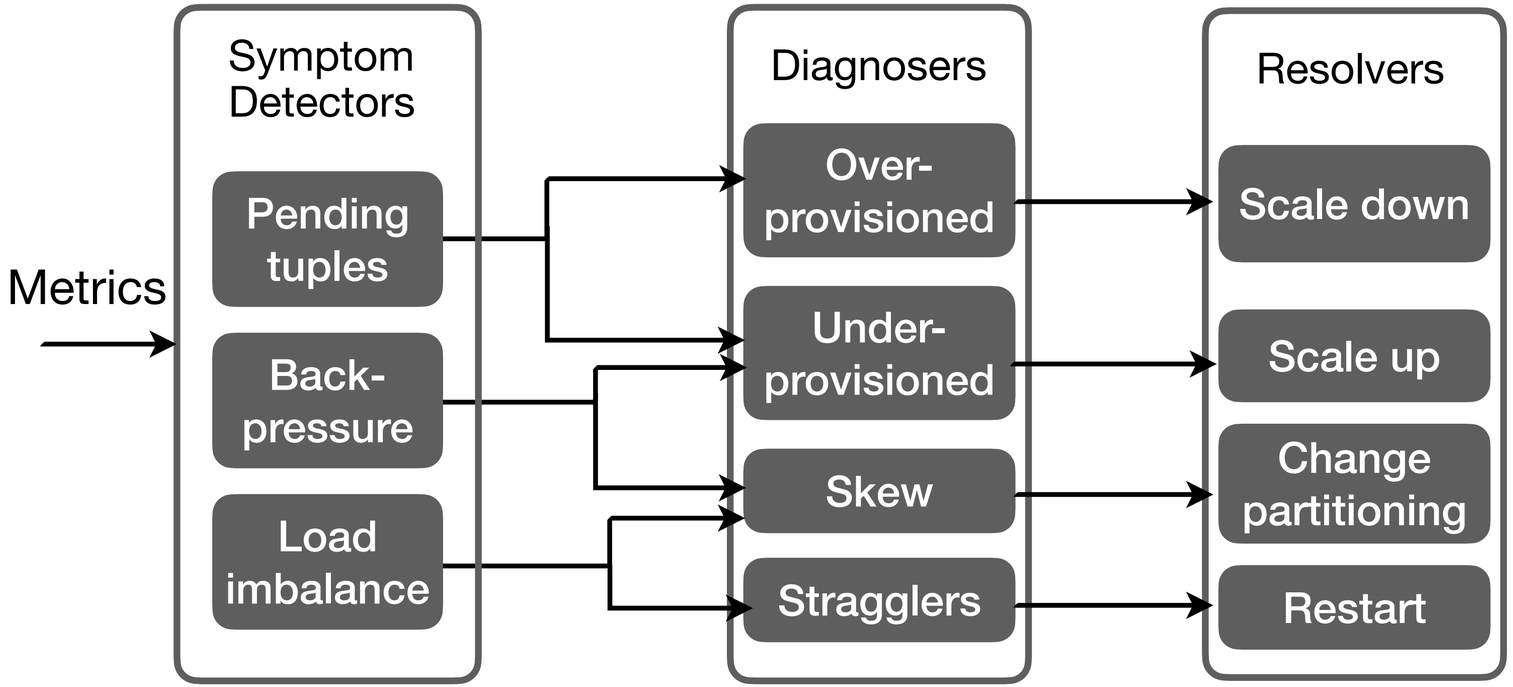
Accuracy: no over/under-provisioning

Stability: no oscillations

Performance: fast convergence

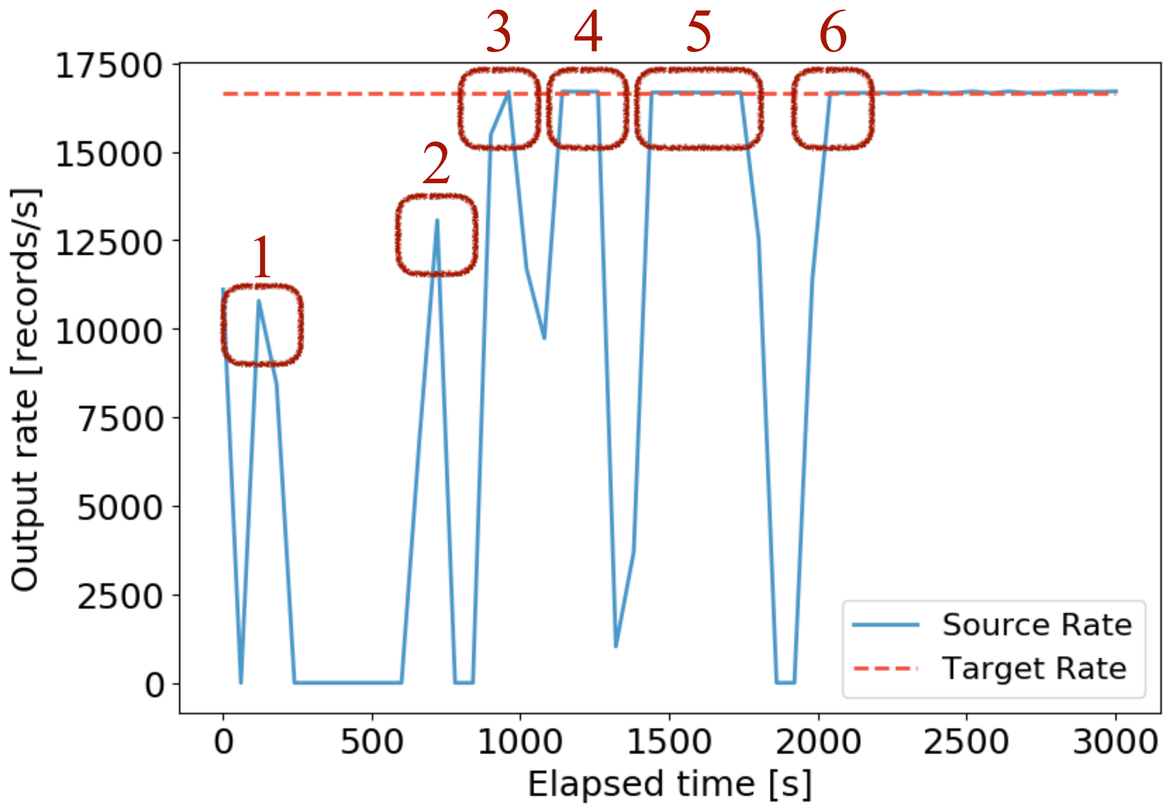
Heuristic Policy

Dhalion (VLDB'17)

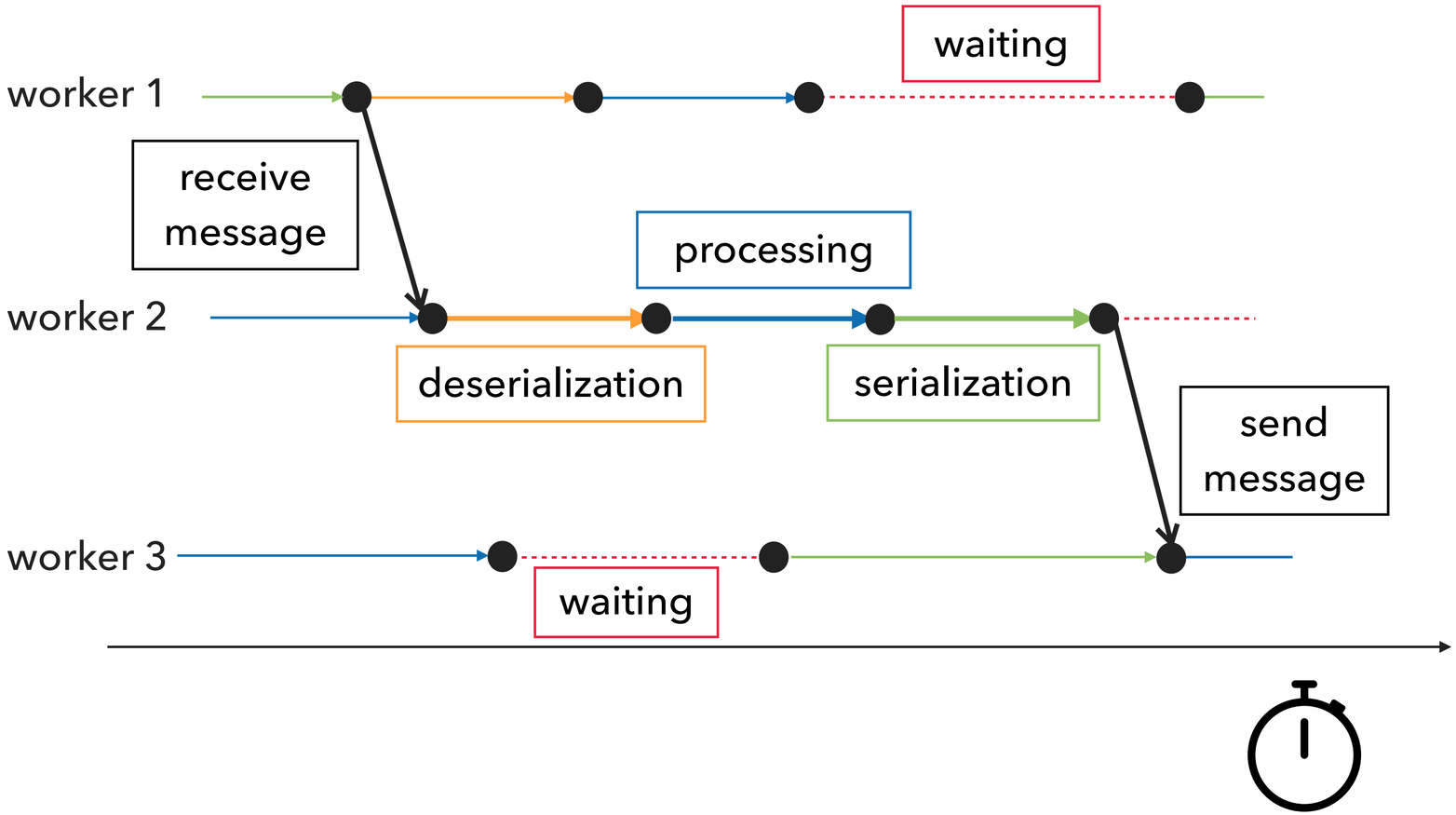


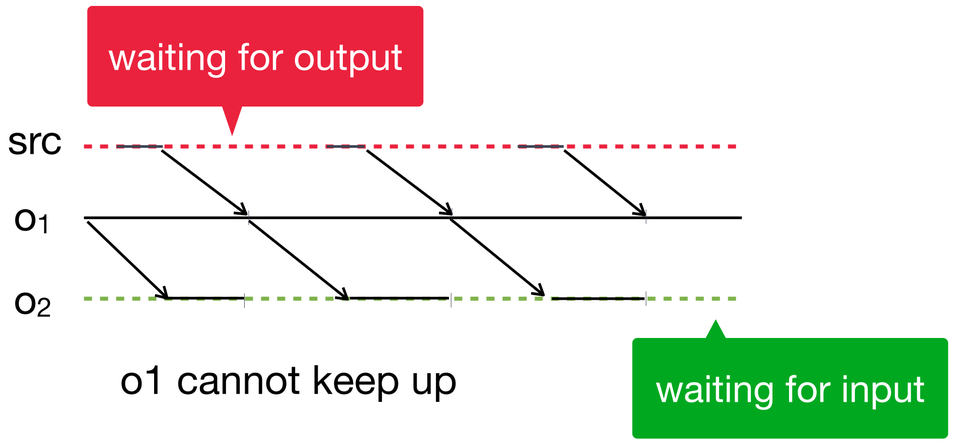
- An **action log** records policies and associated diagnoses
- A **blocklist** records actions that did not produce the expected outcome

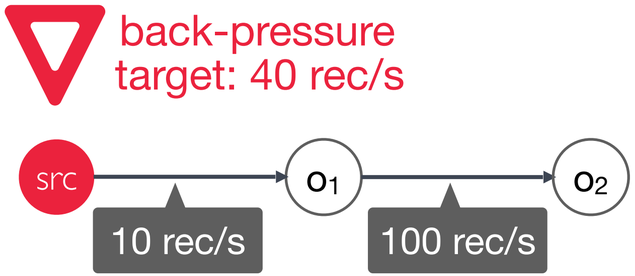
effect of Dhalion's scaling actions
in an initially under-provisioned wordcount dataflow

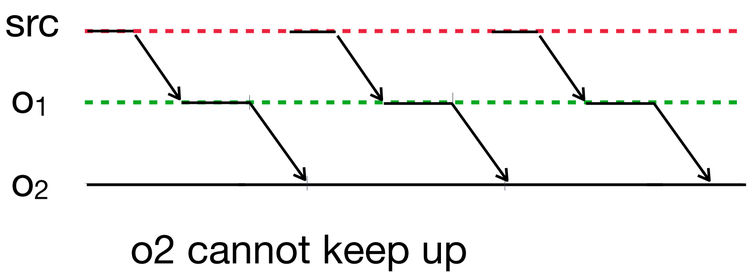


Dataflow worker activities









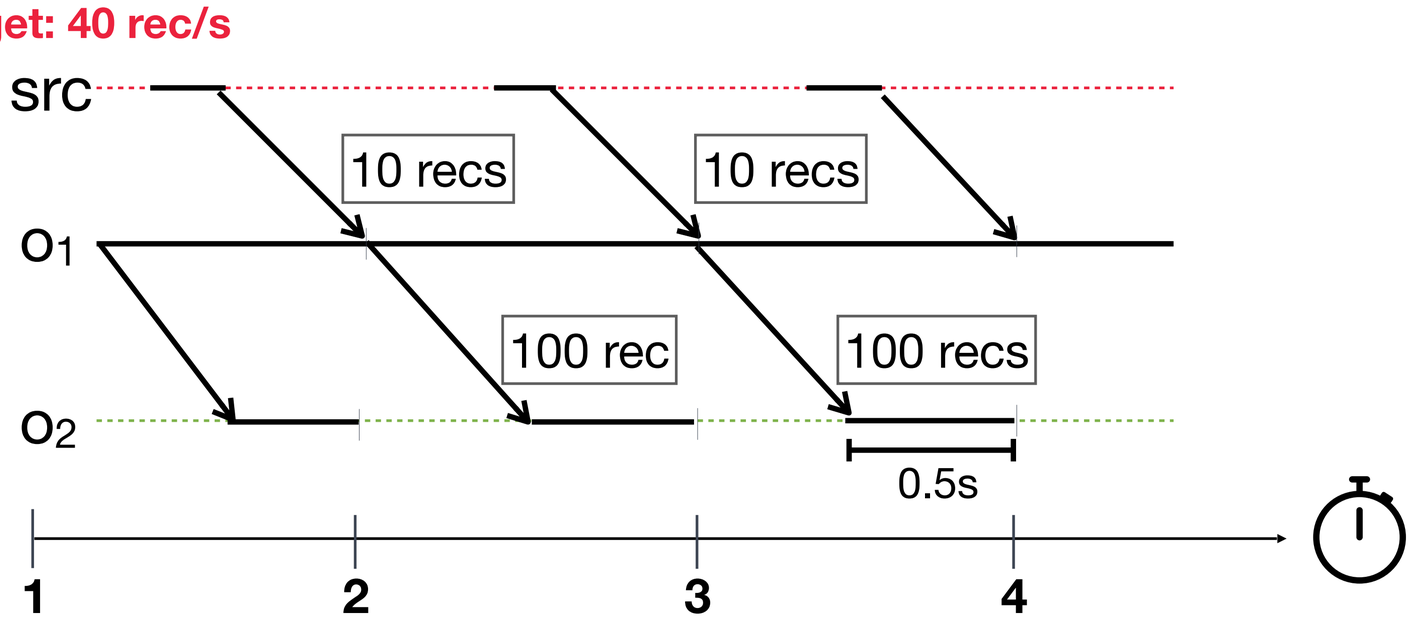
Which operator is the bottleneck?

What if we scale o_1 x 4?

How much to scale o_2?

Predictive Policy

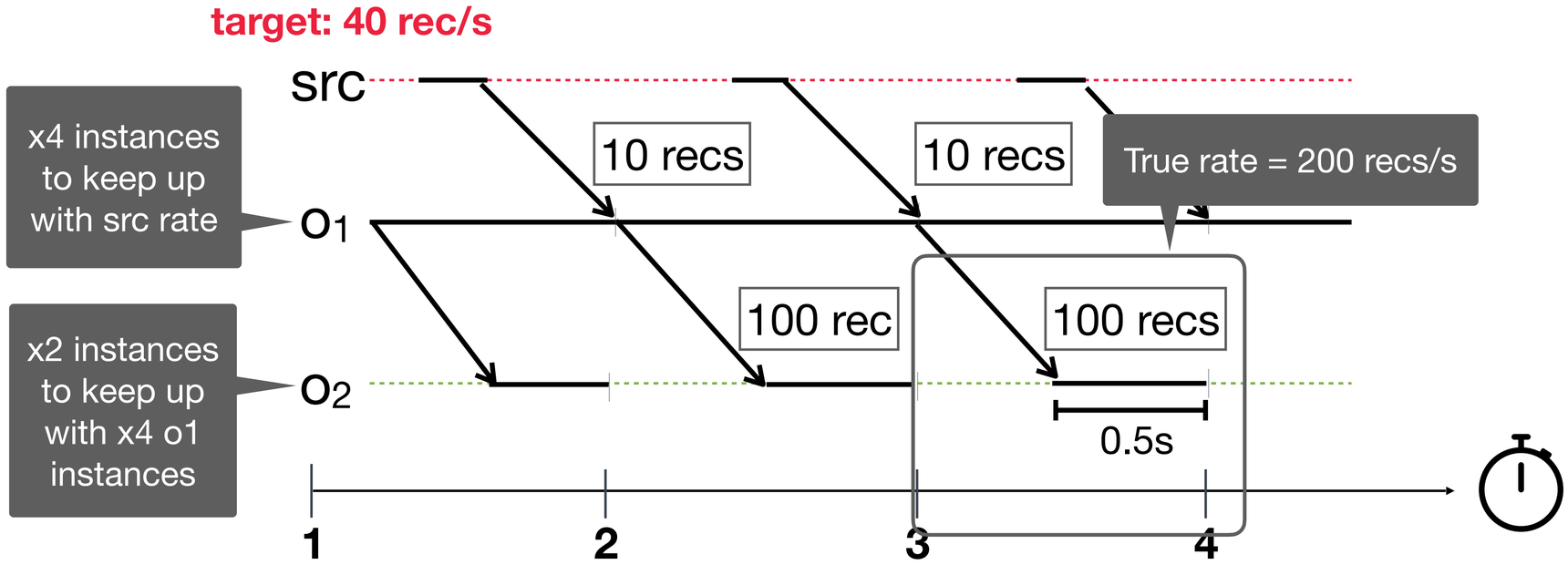
Three steps is all you need - DS2 (OSDI'18)



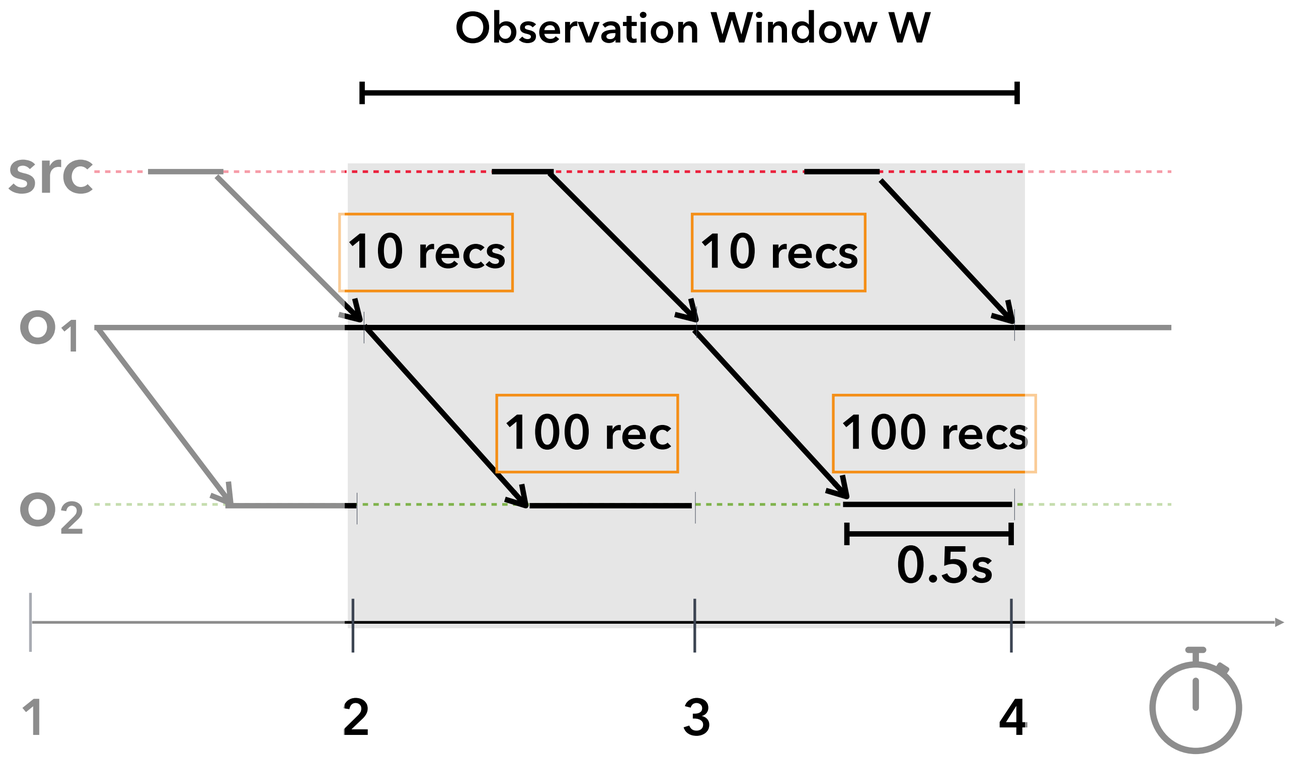
- It uses a **linear model** of operator dependencies as defined by the dataflow graph.
- It relies on **system instrumentation** to collect accurate, representative metrics.
- It computes rates as if operator instances are executed in an **ideal** setting.

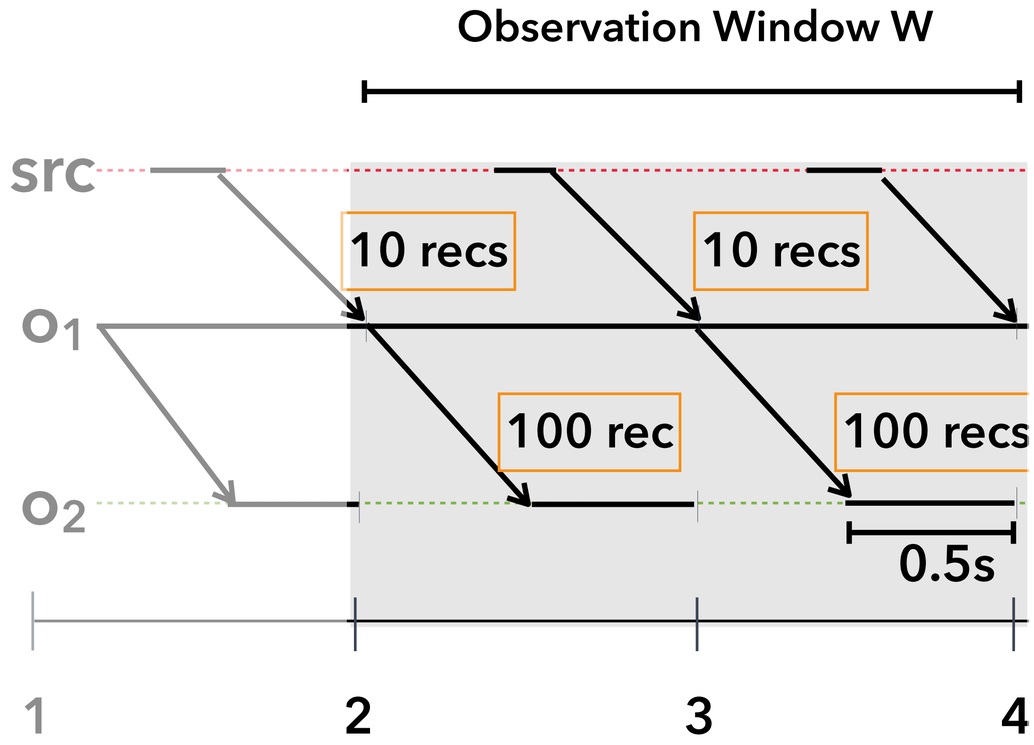
Predictive Policy

Three steps is all you need - DS2 (OSDI'18)



- It uses a **linear model** of operator dependencies as defined by the dataflow graph.
- It relies on **system instrumentation** to collect accurate, representative metrics.
- It computes rates as if operator instances are executed in an **ideal** setting.


Observation Window W
src
10 recs
10 recs
o1
100 rec
100 recs
o2
0.5s
1
2
3
4



Instrumentation Metrics

	o_1	o_2
Records processed R_{pcd}	20	200
Records pushed R_{psd}	200	-
Useful time W_u	2s	1s

The DS2 model

The DS2 model

- Collect metrics per configurable **observation** window **W**
  - **activity durations** per worker
  - records processed $\mathbf{R_{prc}}$ and records pushed to output $\mathbf{R_{psd}}$

The DS2 model

- Collect metrics per configurable **observation** window **W**
  - **activity durations** per worker
  - records processed $\mathbf{R_{prc}}$ and records pushed to output $\mathbf{R_{psd}}$
- Capture **dependencies** through the dataflow graph
  - assign an increasing **sequential id** to all operators in topological order, starting from the sources
  - represent as an **adjacency** matrix **A**
    - $A_{ij} = 1$ iff operator i is upstream neighbor of j

Useful time W_u

The time spent by an operator instance in **deserialization**, **processing**, and **serialization** activities.

- excludes any time spent waiting on input or on output
- amounts to the time an operator instance runs for if executed in an *ideal* setting
  - when there is no waiting the useful time is equal to the **observed time**

True processing / output rates

$$\lambda_p = \frac{R_{\text{prc}}}{W_u} \qquad \lambda_o = \frac{R_{\text{psd}}}{W_u}$$

Aggregated true processing / output rates

$$o_i[\lambda_p] = \sum_{k=1}^{k=p_i} \lambda_p^k \qquad o_i[\lambda_o] = \sum_{k=1}^{k=p_i} \lambda_o^k$$

Optimal parallelism per operator

$$\pi_i = \left\lceil \sum_{\forall j: j<i} A_{ji} \cdot o_j[\lambda_o]^* \cdot \left(\frac{o_i[\lambda_p]}{p_i} \right)^{-1} \right\rceil, n \leq i < m$$

Optimal parallelism per operator

$$\pi_i = \left\lceil \sum_{\forall j: j<i} A_{ji} \cdot o_j[\lambda_o]^* \cdot \left(\frac{o_i[\lambda_p]}{p_i} \right)^{-1} \right\rceil, n \leq i < m$$

captures upstream operators

Optimal parallelism per operator

$$\pi_i = \left\lceil \sum_{\forall j: j<i} A_{ji} \cdot o_j[\lambda_o]^* \cdot \left(\frac{o_i[\lambda_p]}{p_i} \right)^{-1} \right\rceil, n \leq i < m$$

captures upstream operators

Aggregated true output rate of operator o_j , when o_j itself and all upstream ops are deployed with optimal parallelism

Optimal parallelism per operator

$$\pi_i = \left\lceil \sum_{\forall j: j<i} A_{ji} \cdot o_j[\lambda_o]^* \cdot \left(\frac{o_i[\lambda_p]}{p_i} \right)^{-1} \right\rceil, n \leq i < m$$

captures upstream operators

Aggregated true output rate of operator o_j , when o_j itself and all upstream ops are deployed with optimal parallelism

current parallelism of operator i

Recursively computed as:

$$o_j[\lambda_o]^* = \begin{cases} o_j[\lambda_o] = \lambda_{\text{src}}^j, & 0 \leq j < n \\ \dfrac{o_j[\lambda_o]}{o_j[\lambda_p]} \cdot \sum\limits_{\forall u : u < j} A_{uj} \cdot o_u[\lambda_o]^*, & n \leq j < m \end{cases}$$

Recursively computed as:

True output rate of source j

$$o_j[\lambda_o]^* = \begin{cases} o_j[\lambda_o] = \lambda^j_{\text{src}}, & 0 \le j < n \\ \dfrac{o_j[\lambda_o]}{o_j[\lambda_p]} \cdot \sum\limits_{\forall u : u < j} A_{uj} \cdot o_u[\lambda_o]^*, & n \le j < m \end{cases}$$

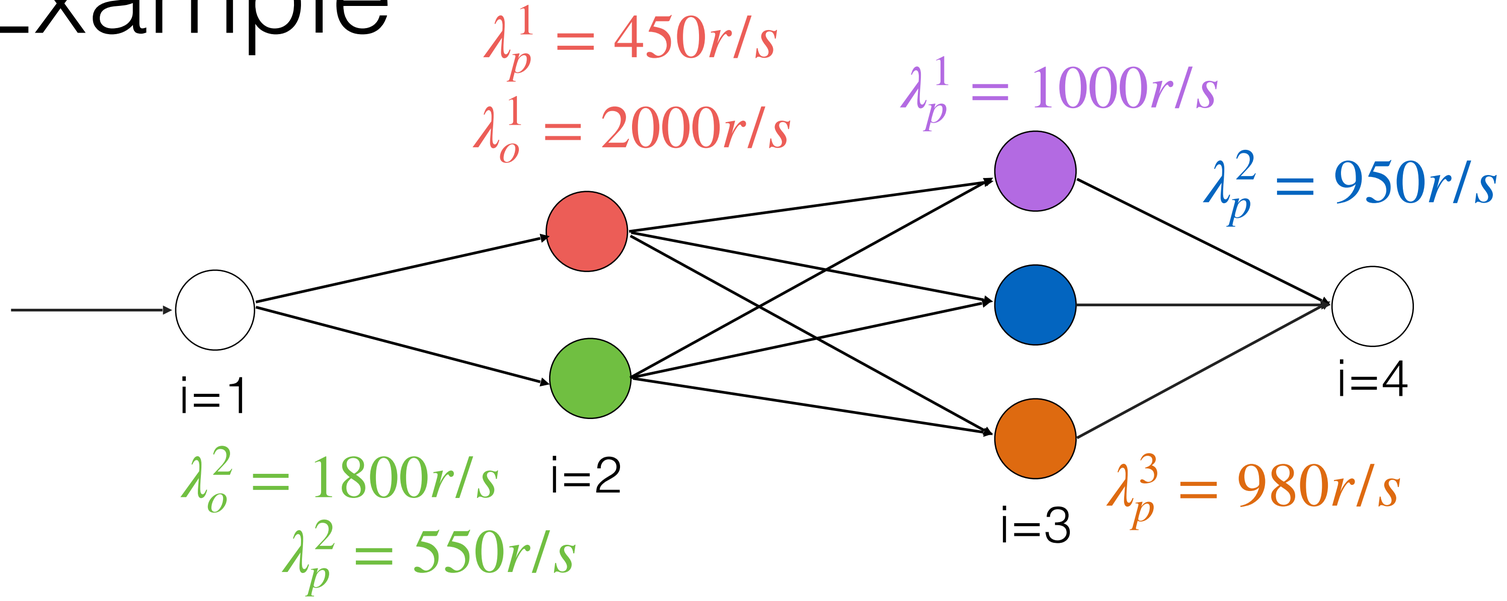
Recursively computed as:

$$o_j[\lambda_o]^* = \begin{cases} o_j[\lambda_o] = \lambda^j_{\text{src}}, & 0 \leq j < n \\ \frac{o_j[\lambda_o]}{o_j[\lambda_p]} \cdot \sum_{\forall u : u < j} A_{uj} \cdot o_u[\lambda_o]^*, & n \leq j < m \end{cases}$$

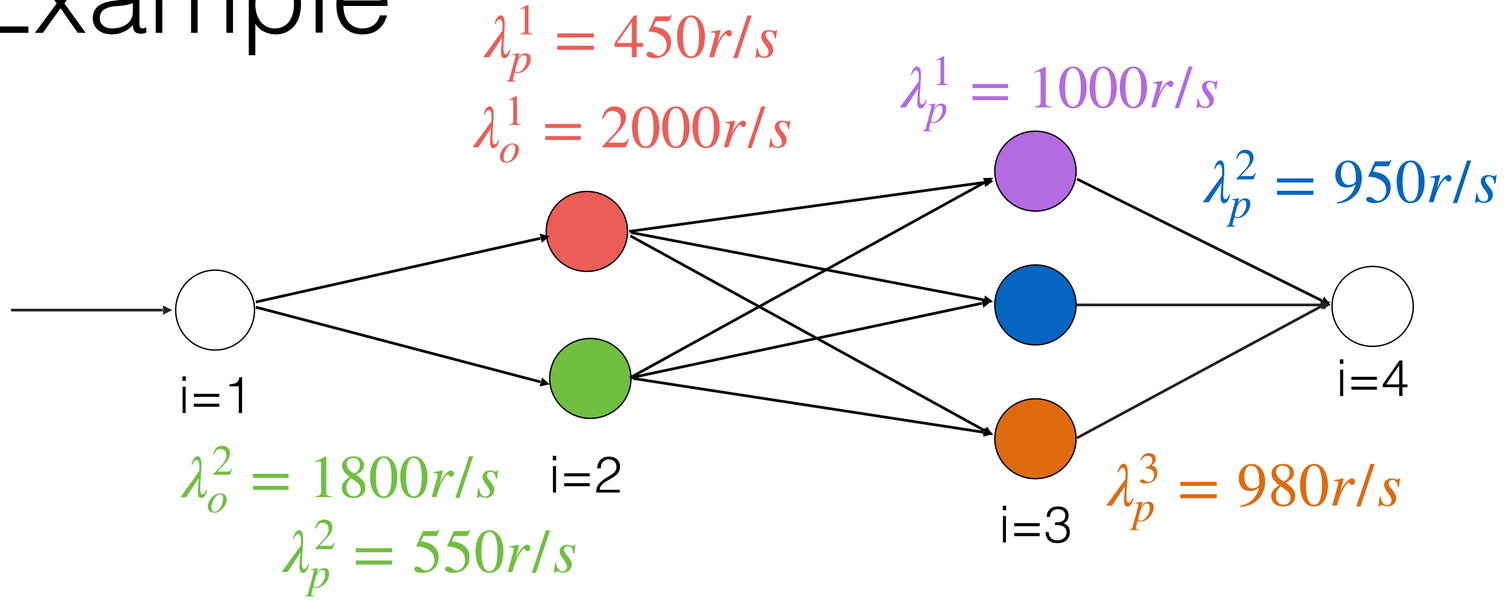
True output rate of source j

It can be computed **for all operators** by traversing the dataflow from left to right **once**

Example


$\lambda_p^1 = 450r/s$
$\lambda_o^1 = 2000r/s$
$\lambda_p^1 = 1000r/s$
$\lambda_p^2 = 950r/s$
i=1
i=2
i=3
i=4
$\lambda_o^2 = 1800r/s$
$\lambda_p^2 = 550r/s$
$\lambda_p^3 = 980r/s$

Example

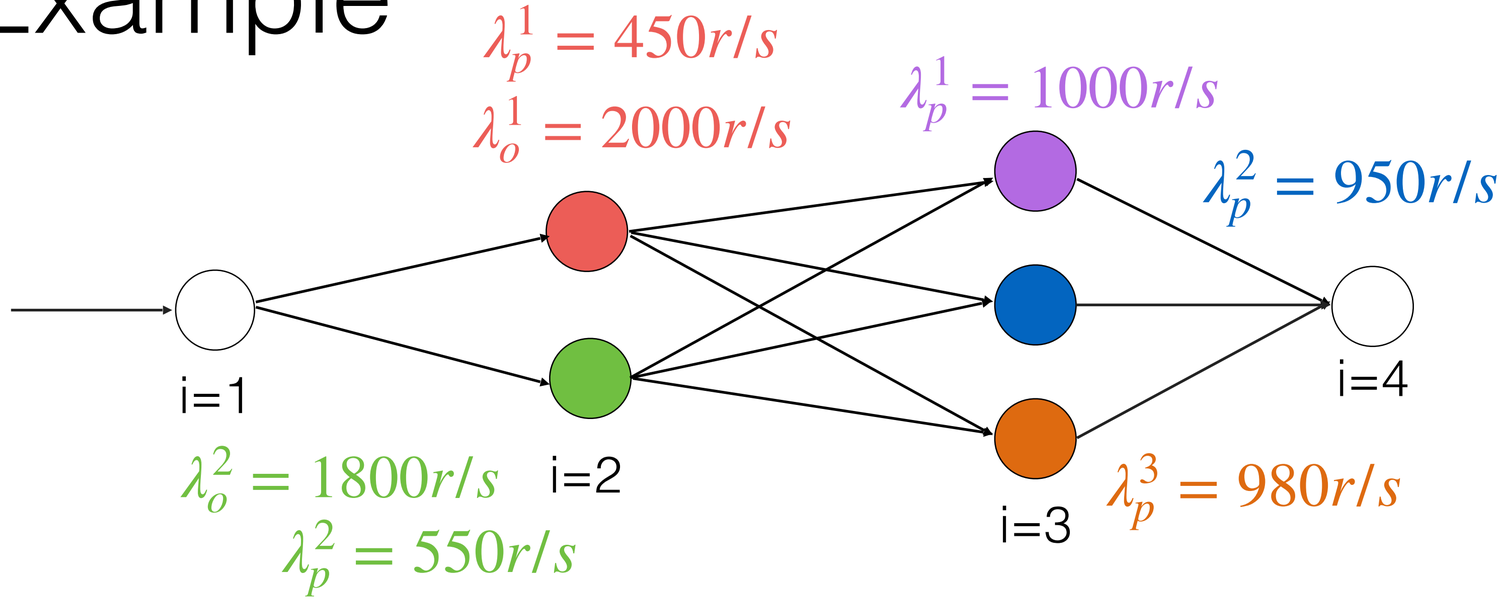


$o_1[\lambda_p] = 0$
$o_1[\lambda_o] = 2000$ r/s

$o_2[\lambda_p] = 1000$ r/s
$o_2[\lambda_o] = 3800$ r/s

$o_3[\lambda_p] = 2930$ r/s
$o_3[\lambda_o] = 600$ r/s

Example



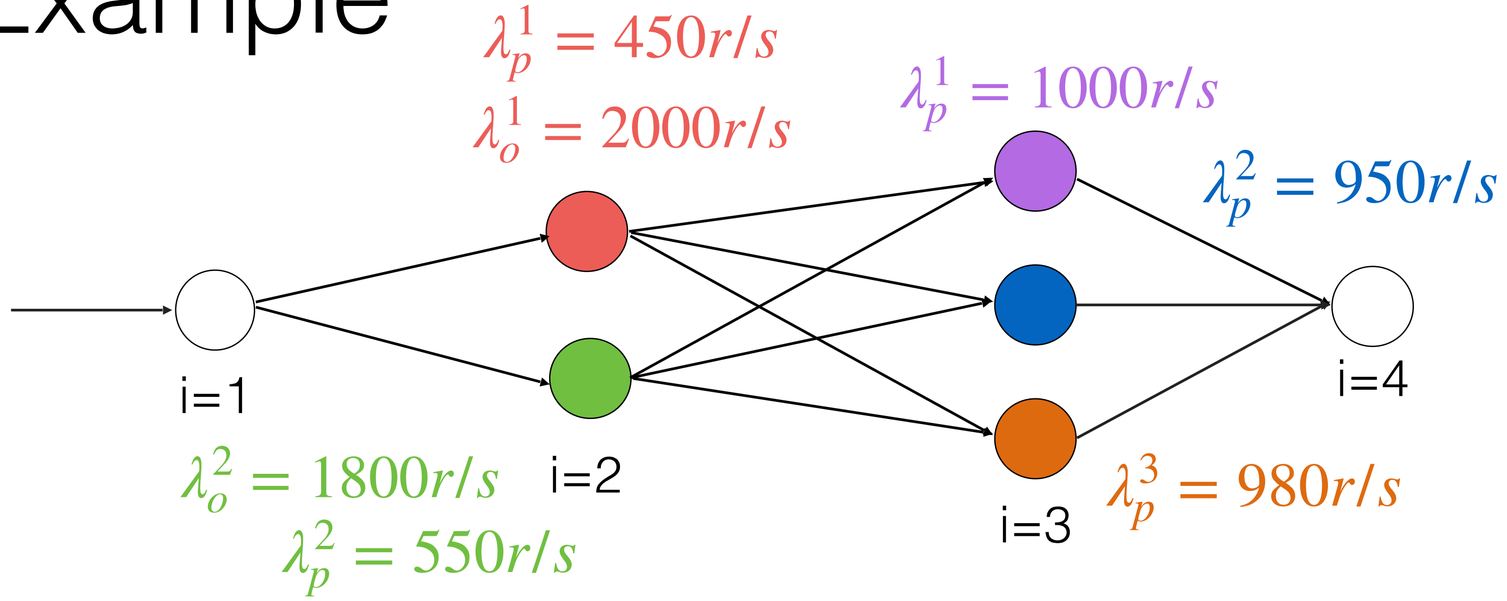
$o_1[\lambda_p] = 0$
$o_1[\lambda_o] = 2000$ r/s

$o_2[\lambda_p] = 1000$ r/s
$o_2[\lambda_o] = 3800$ r/s

$o_3[\lambda_p] = 2930$ r/s
$o_3[\lambda_o] = 600$ r/s

$$\pi_2 = o_1[\lambda_o^*] * \frac{p_2}{o_2[\lambda_p]} = 2000 * \frac{2}{1000} = 4$$

Example



$o_1[\lambda_p] = 0$
$o_1[\lambda_o] = 2000$ r/s

$o_2[\lambda_p] = 1000$ r/s
$o_2[\lambda_o] = 3800$ r/s

$o_3[\lambda_p] = 2930$ r/s
$o_3[\lambda_o] = 600$ r/s

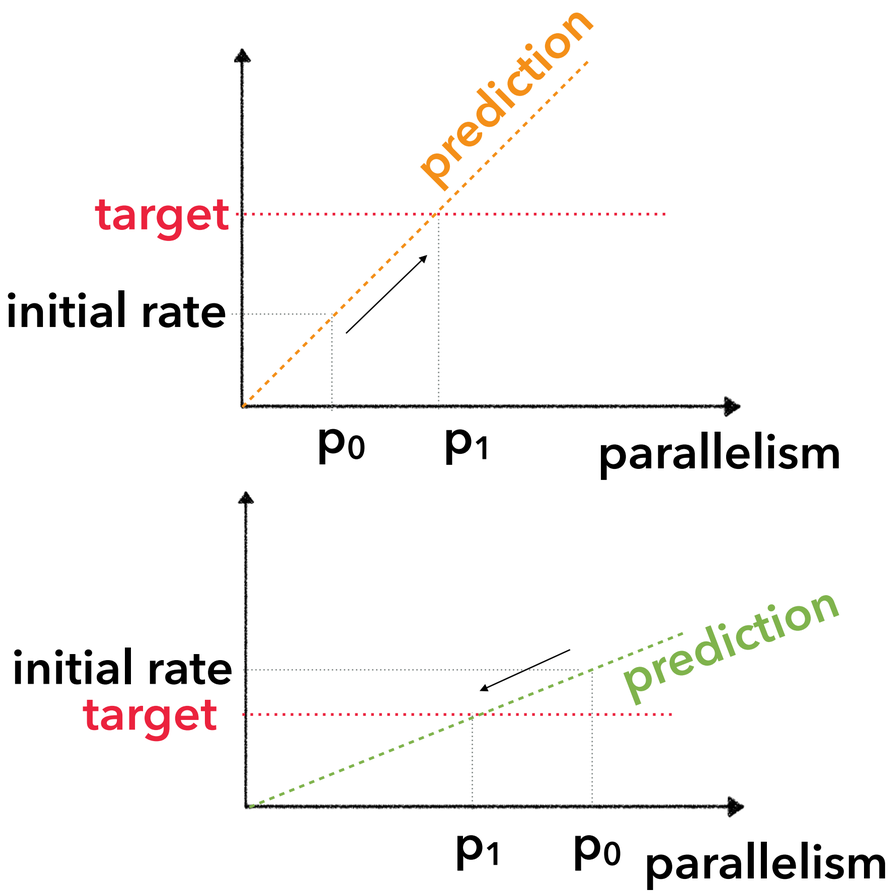
$$\pi_2 = o_1[\lambda_o^*] * \frac{p_2}{o_2[\lambda_p]} = 2000 * \frac{2}{1000} = 4$$

$$\pi_3 = o_2[\lambda_o^*] * \frac{p_3}{o_3[\lambda_p]} = 7600 * \frac{3}{2930} \approx 7.78 \rightarrow 8$$

DS2 model properties

If operator scaling is **linear**, then:

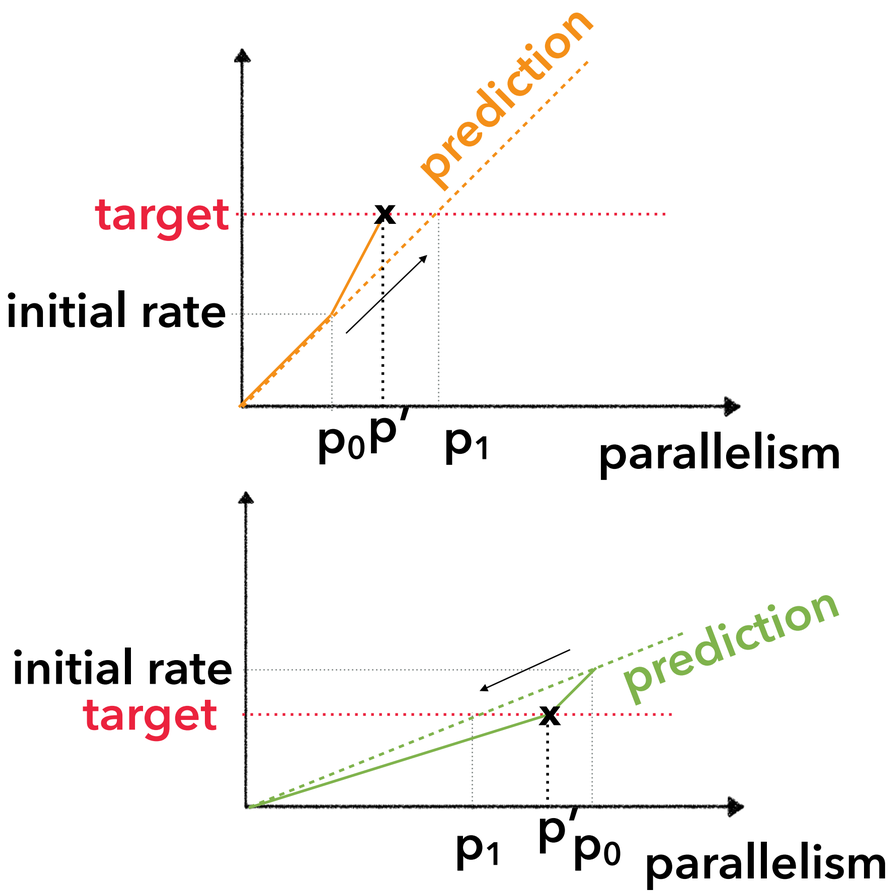
- **no overshoot** when scaling up
- **no undershoot** when scaling down



DS2 model properties

If operator scaling is **linear**, then:

- **no overshoot** when scaling up
- **no undershoot** when scaling down



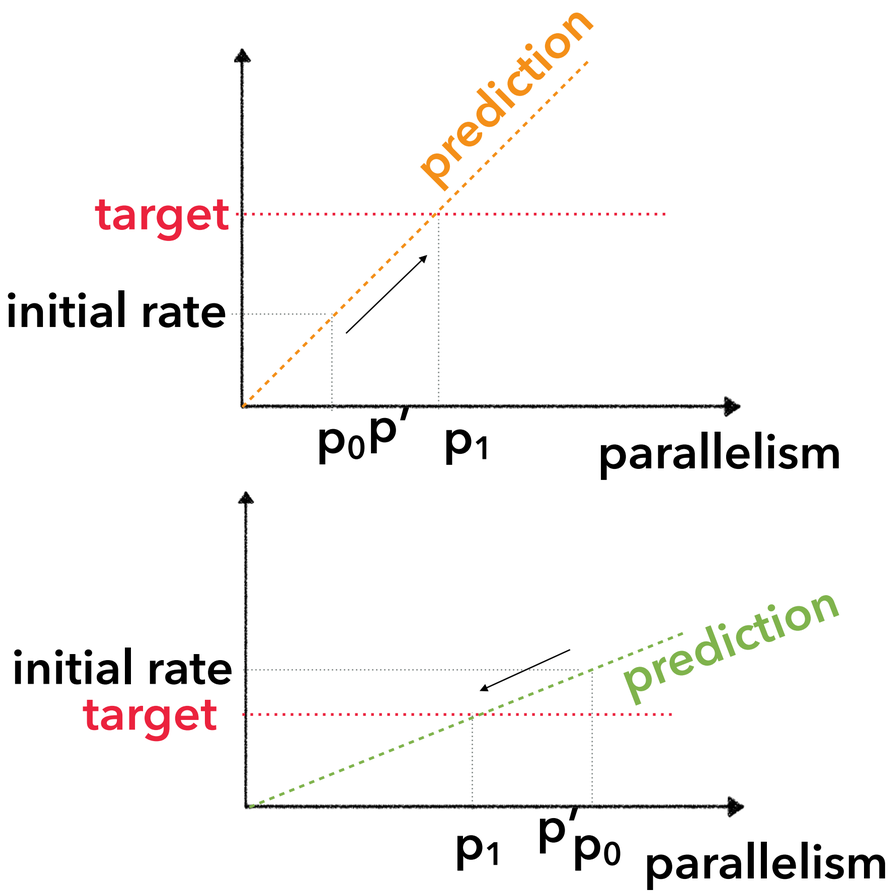
DS2 model properties

If operator scaling is **linear**, then:

- **no overshoot** when scaling up
- **no undershoot** when scaling down

Ideal rates act as un **upper bound** when scaling up and as a **lower bound** when scaling down:

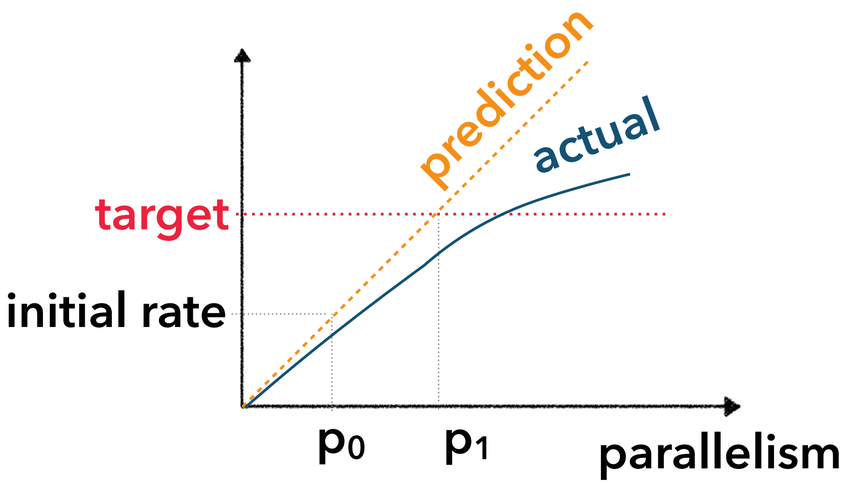
DS2 will **converge monotonically** to the target rate



DS2 model properties

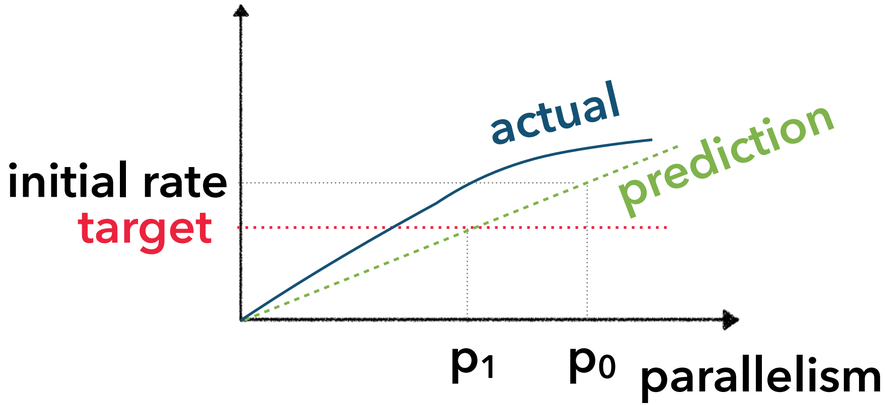
If operator scaling is **linear**, then:

- **no overshoot** when scaling up
- **no undershoot** when scaling down

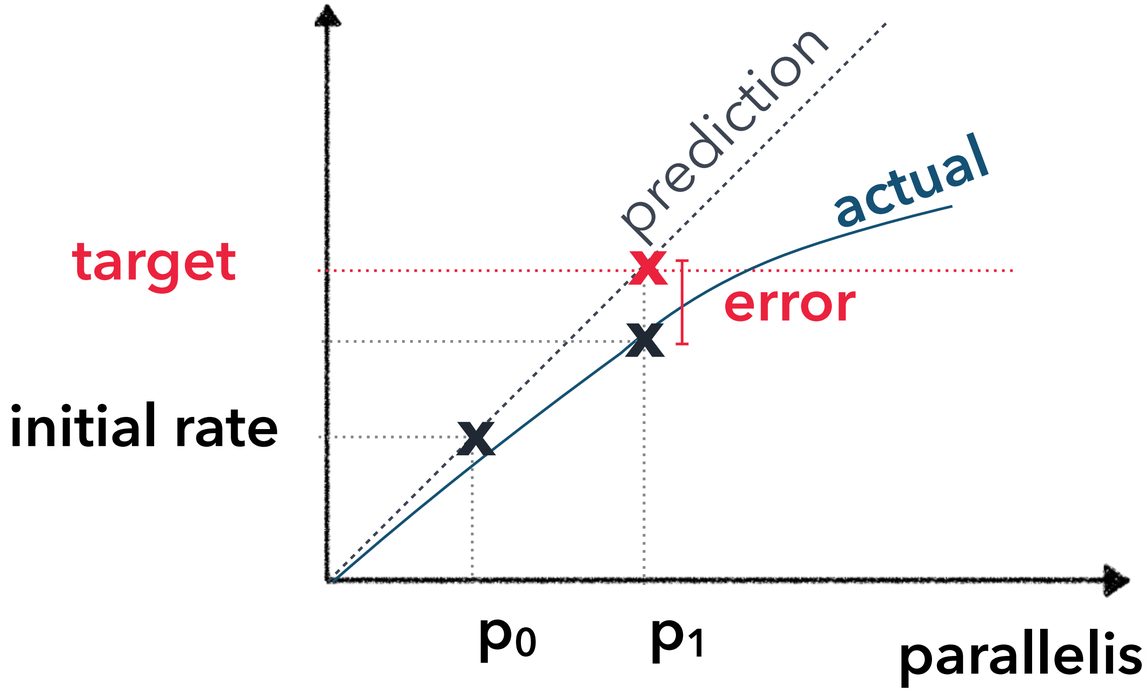


Ideal rates act as un **upper bound** when scaling up and as a **lower bound** when scaling down:

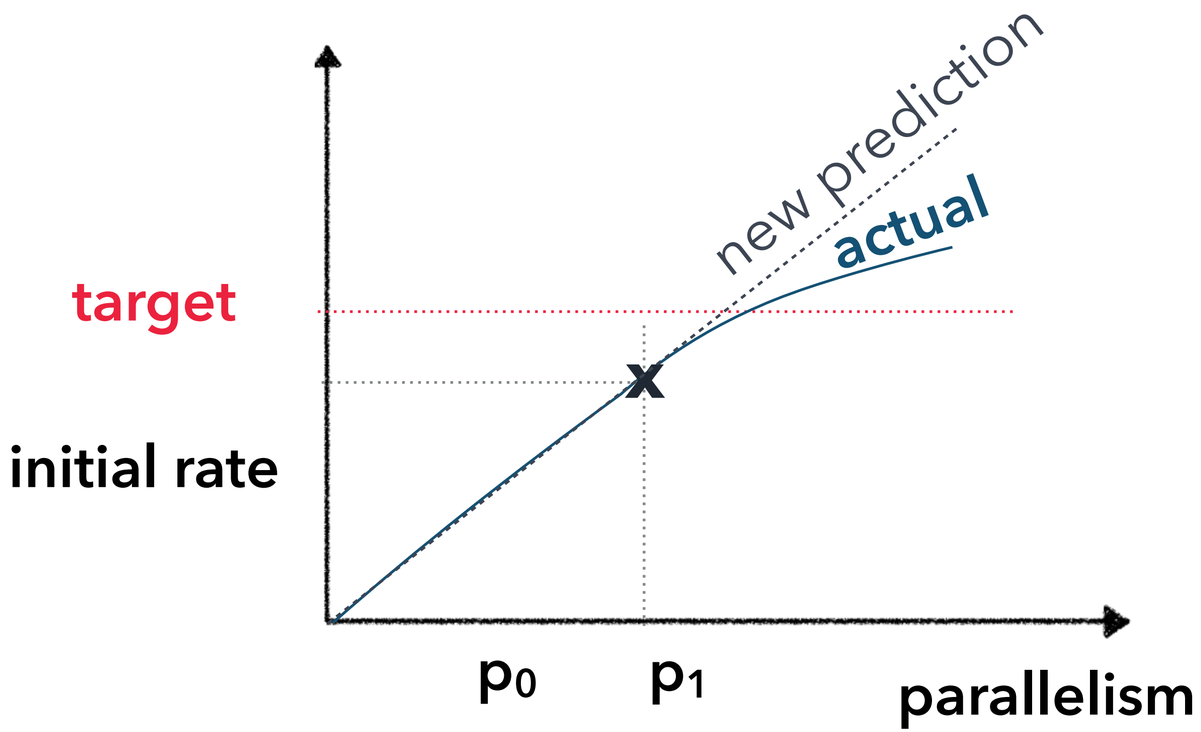
DS2 will **converge monotonically** to the target rate



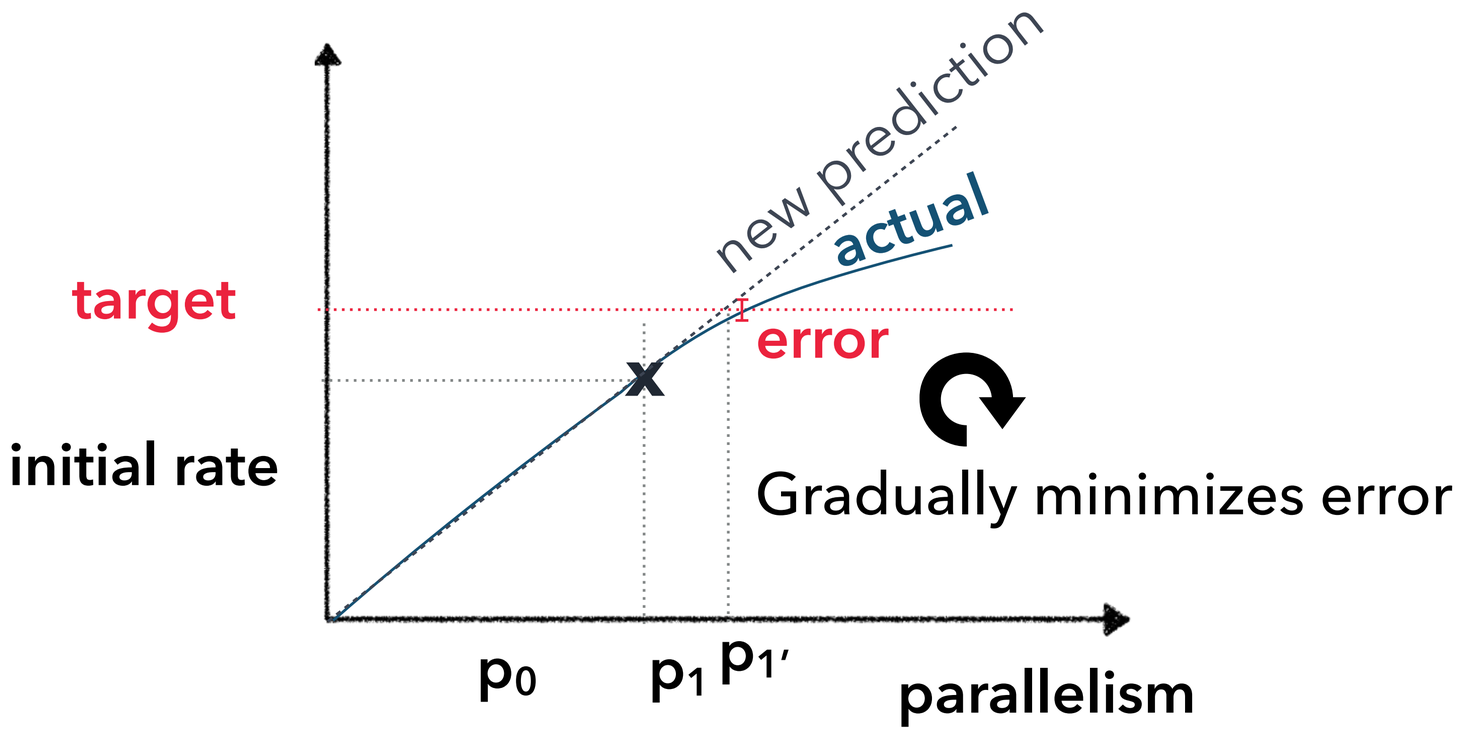
DS2 model properties

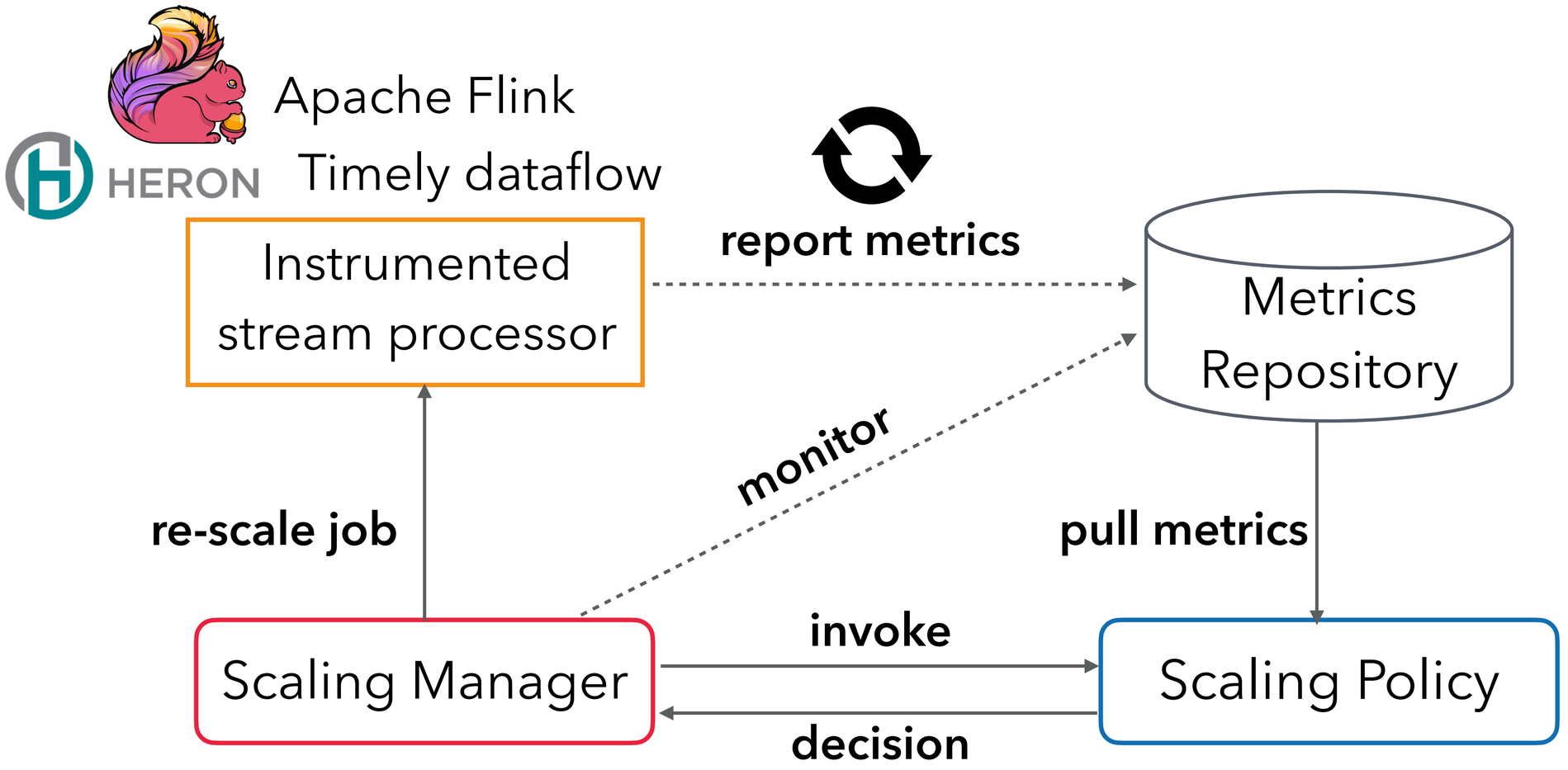


DS2 model properties



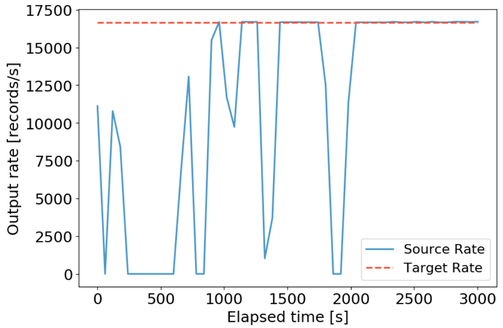
DS2 model properties

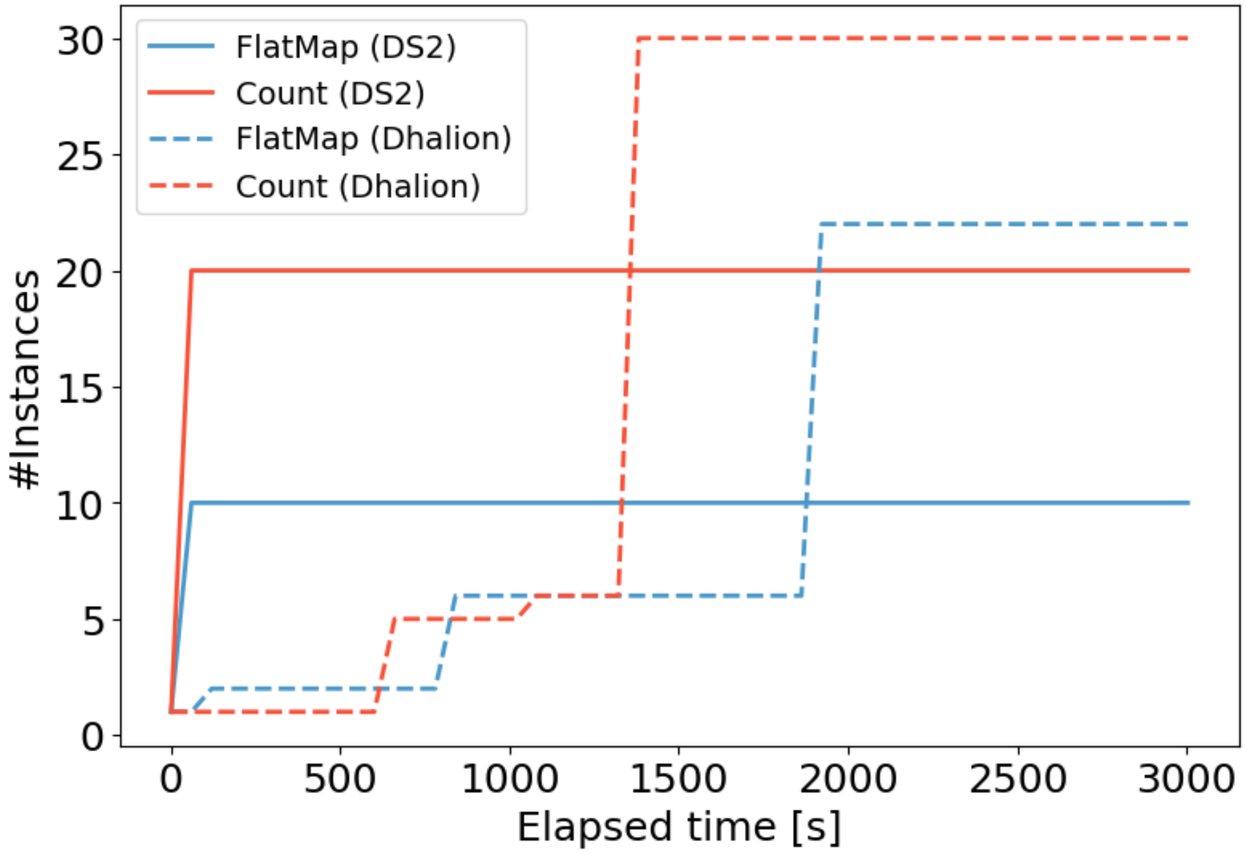



Apache Flink
Timely dataflow
HERON
report metrics
Instrumented
stream processor
Metrics
Repository
monitor
re-scale job
pull metrics
invoke
Scaling Manager
Scaling Policy
decision

Initially under-provisioned wordcount dataflow

Target rate: 16.700 rec/s

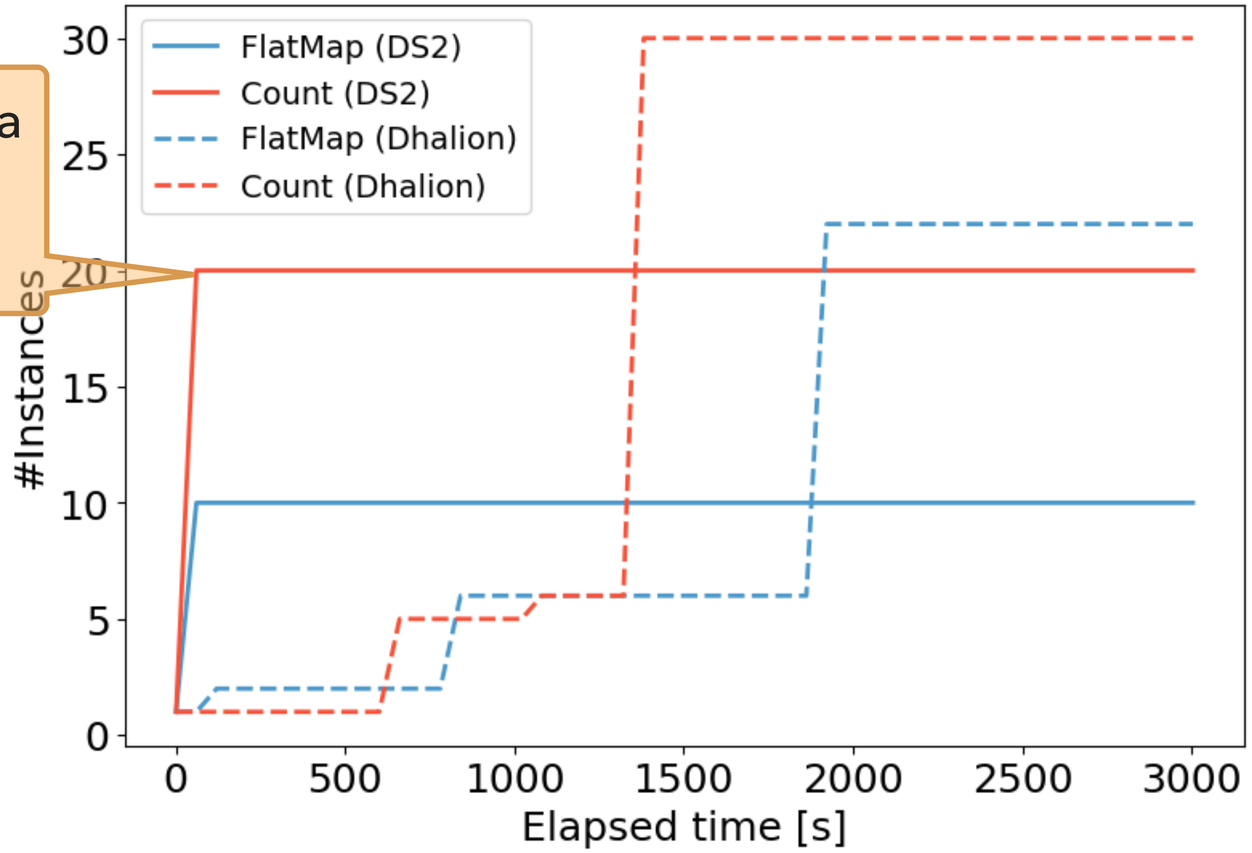




Initially under-provisioned wordcount dataflow

Target rate: 16.700 rec/s

DS2 converges in a **single step** for both operators

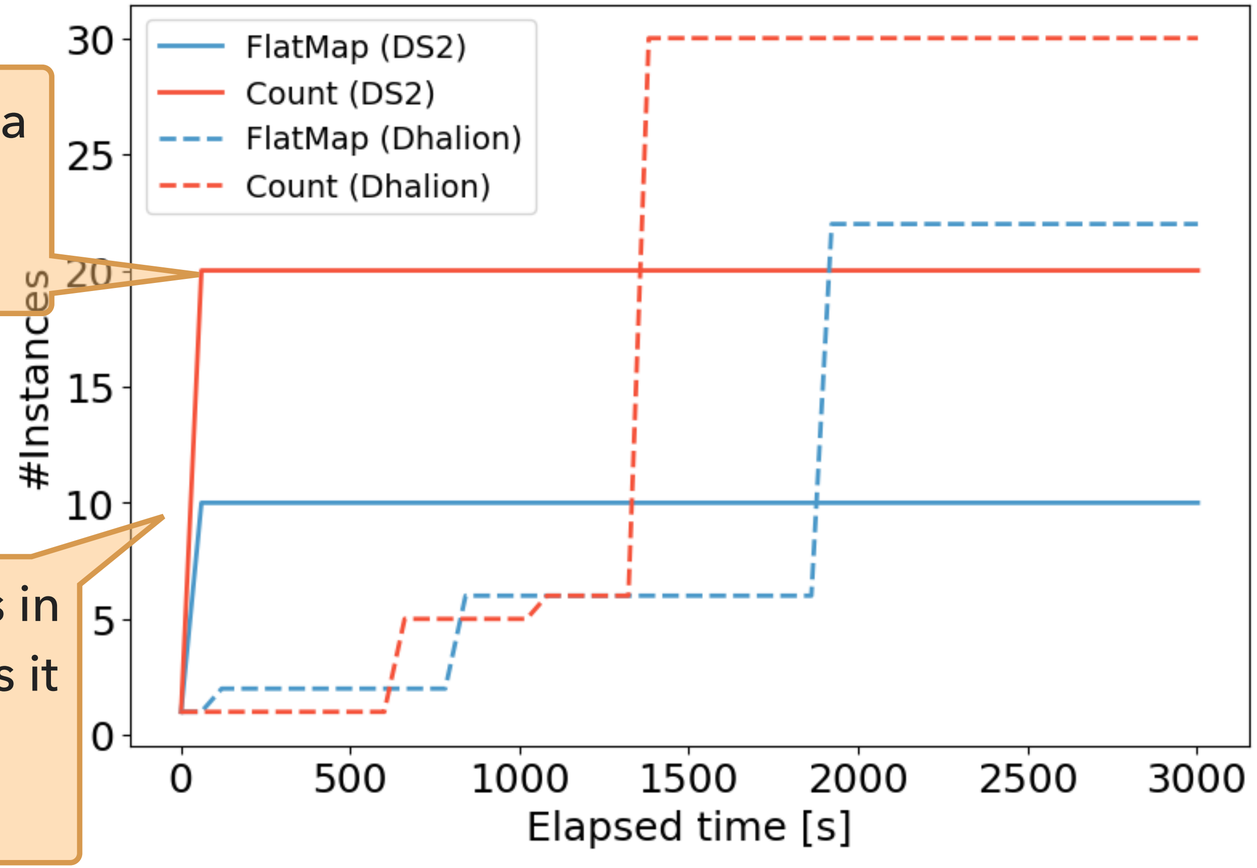


Initially under-provisioned wordcount dataflow

Target rate: 16.700 rec/s

DS2 converges in a **single step** for both operators

and converges in **60s**, as soon as it receives the Heron metrics

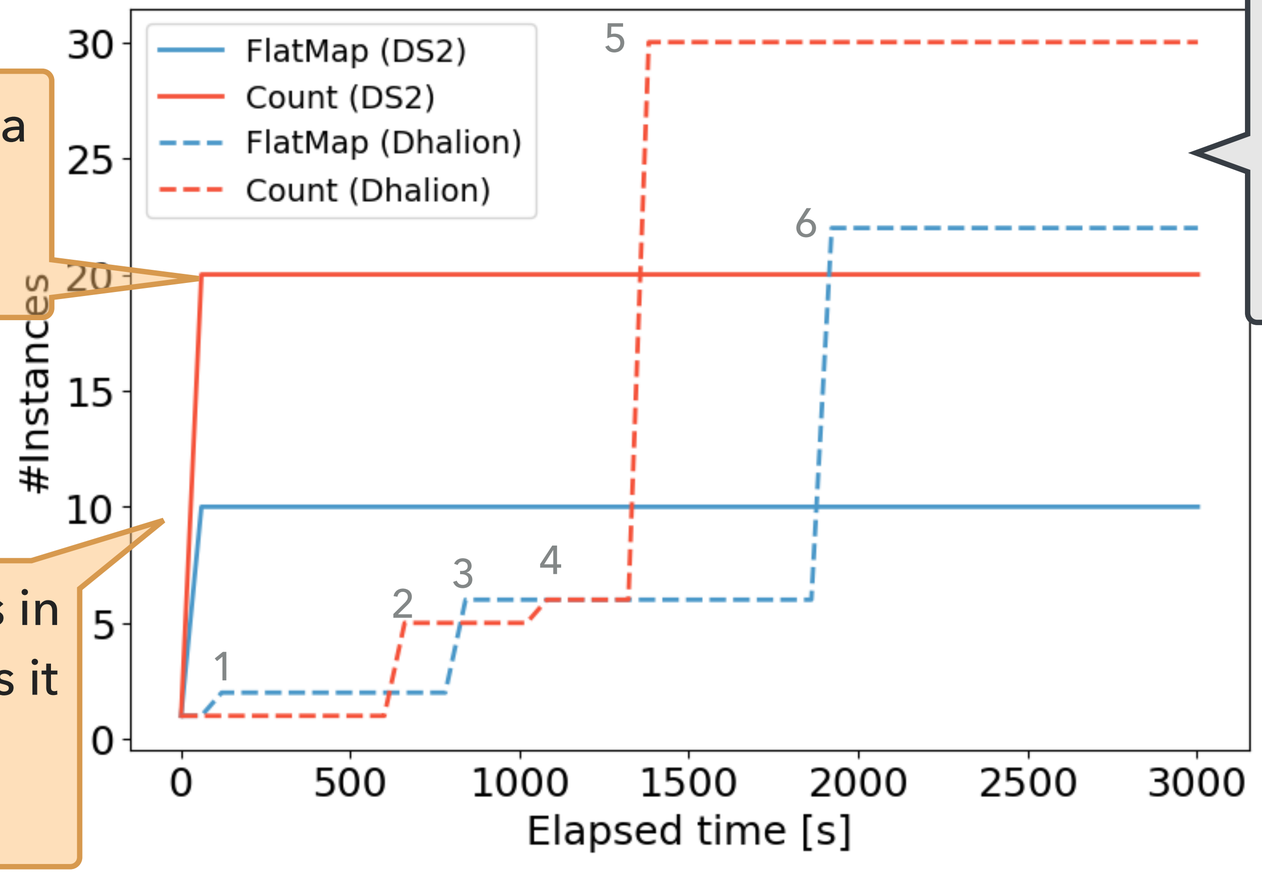


Initially under-provisioned wordcount dataflow

Target rate: 16.700 rec/s

DS2 converges in a **single step** for both operators

and converges in **60s**, as soon as it receives the Heron metrics



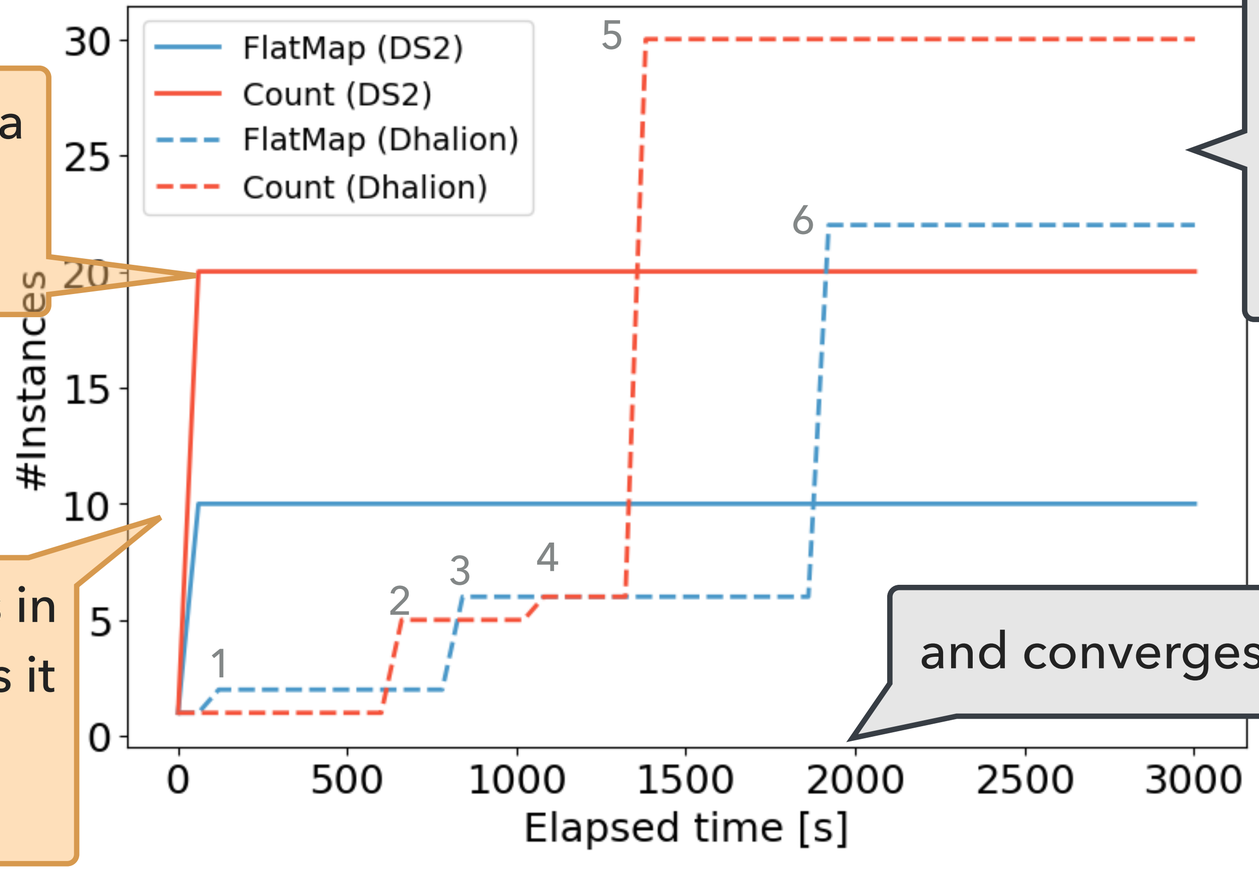
Dhalion scales one operator at a time, and needs **six steps** in total

Initially under-provisioned wordcount dataflow

Target rate: 16.700 rec/s

DS2 converges in a **single step** for both operators

and converges in **60s**, as soon as it receives the Heron metrics



Dhalion scales one operator at a time, and needs **six steps** in total

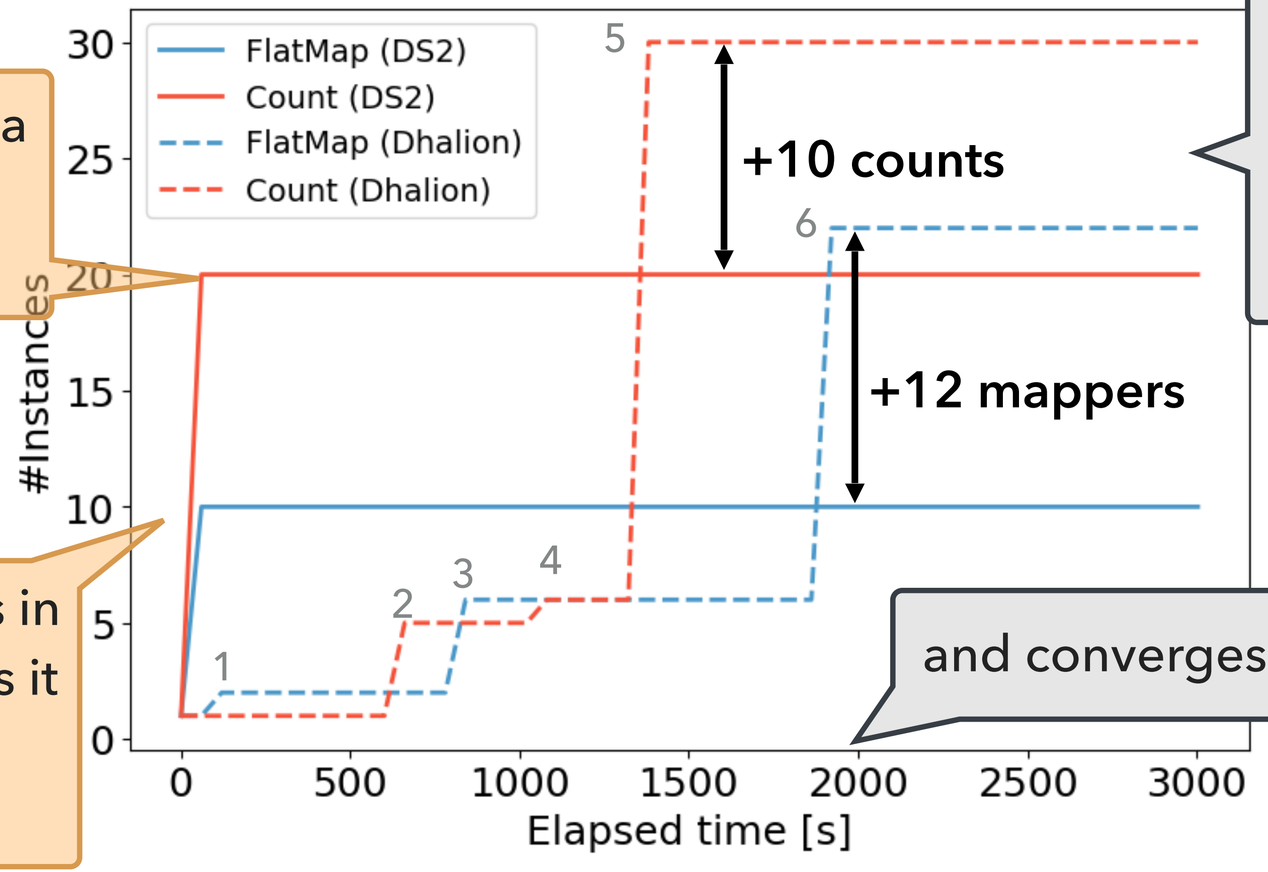
and converges in **2000s**

Initially under-provisioned wordcount dataflow

Target rate: 16.700 rec/s

DS2 converges in a **single step** for both operators

and converges in **60s**, as soon as it receives the Heron metrics



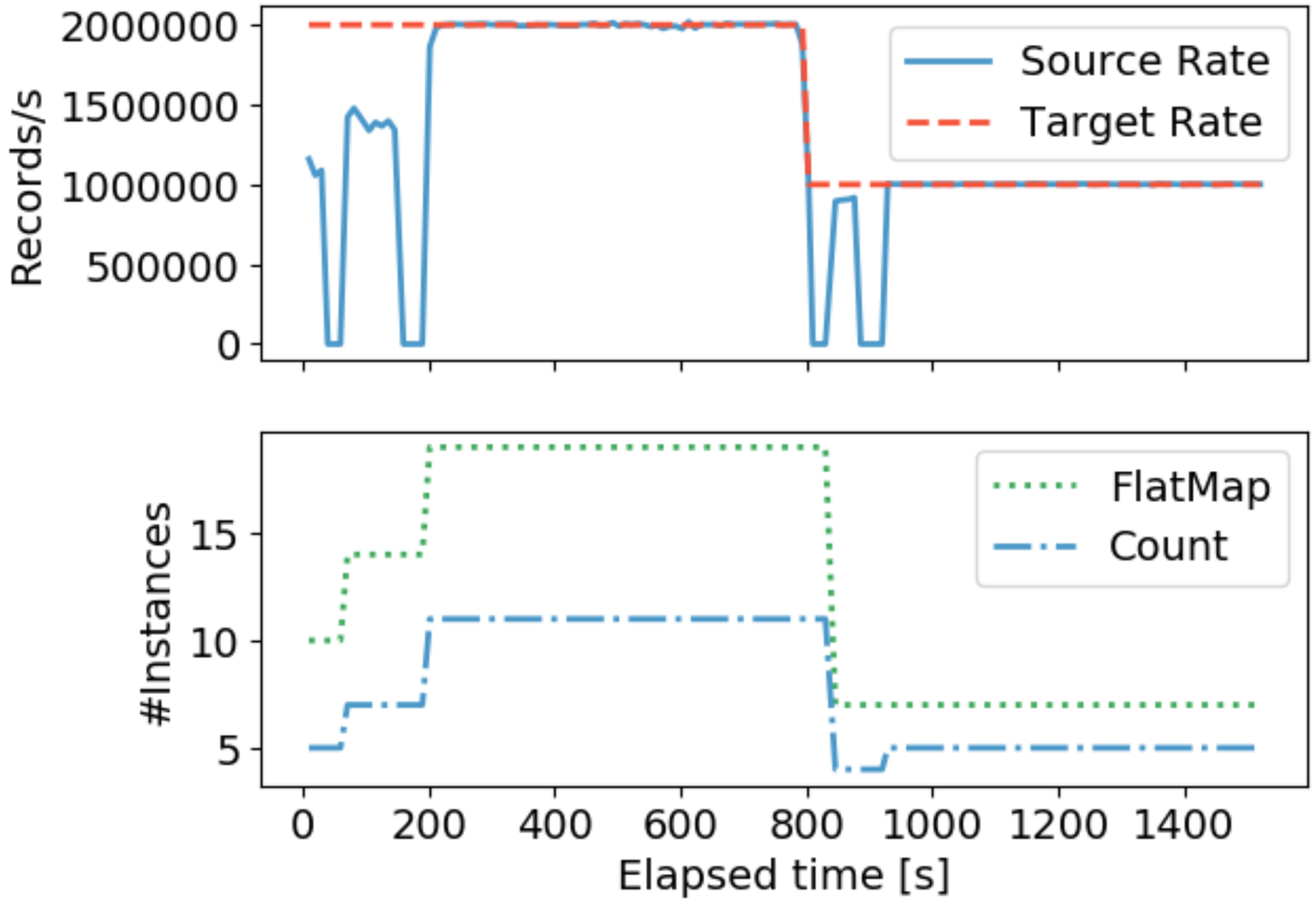
Dhalion scales one operator at a time, and needs **six steps** in total

and converges in **2000s**

DS2 on Flink

Initially under-provisioned wordcount

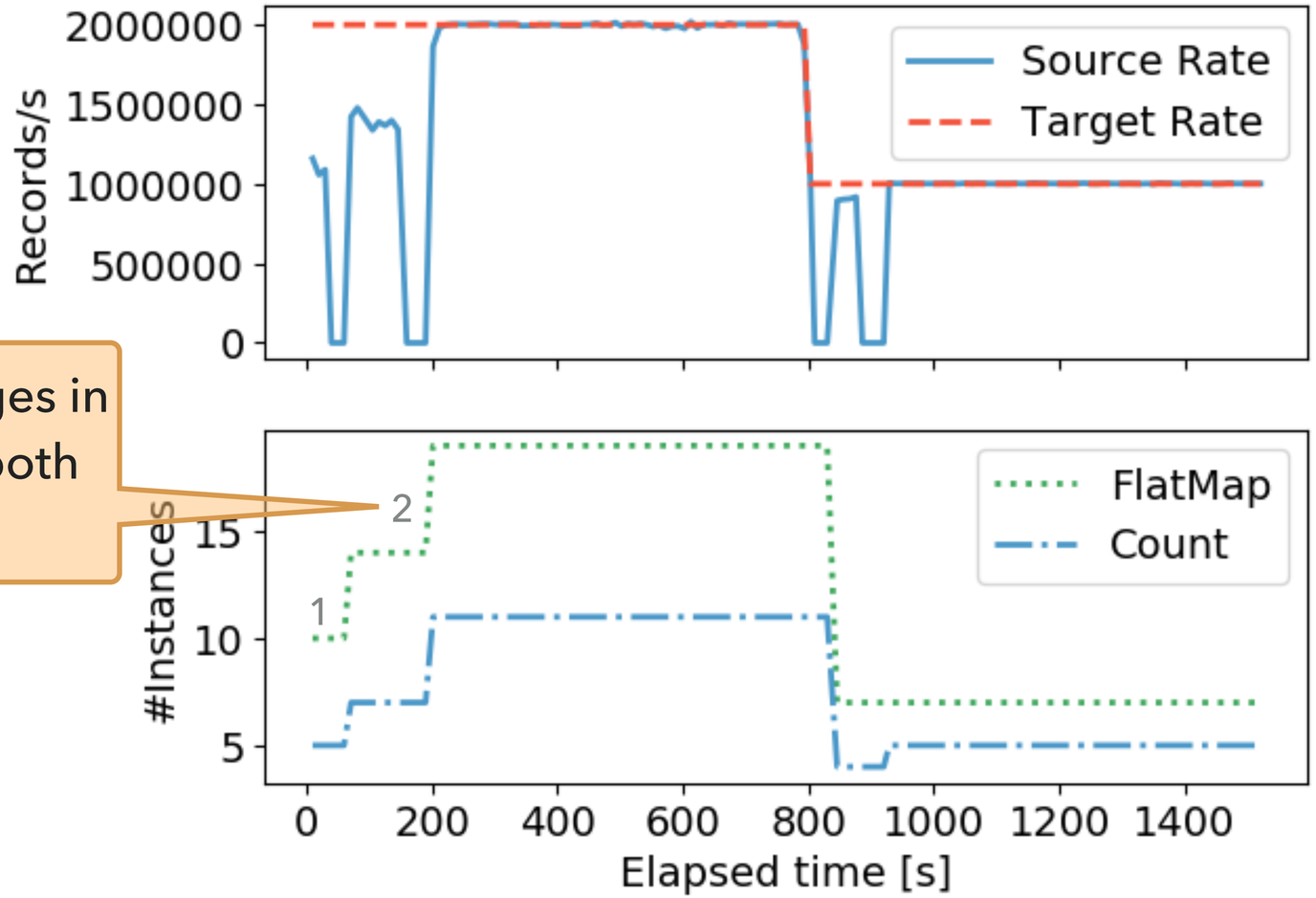
Target rate: 2.000.000 rec/s, drops to half at 800s



DS2 on Flink

Initially under-provisioned wordcount

Target rate: 2.000.000 rec/s, drops to half at 800s

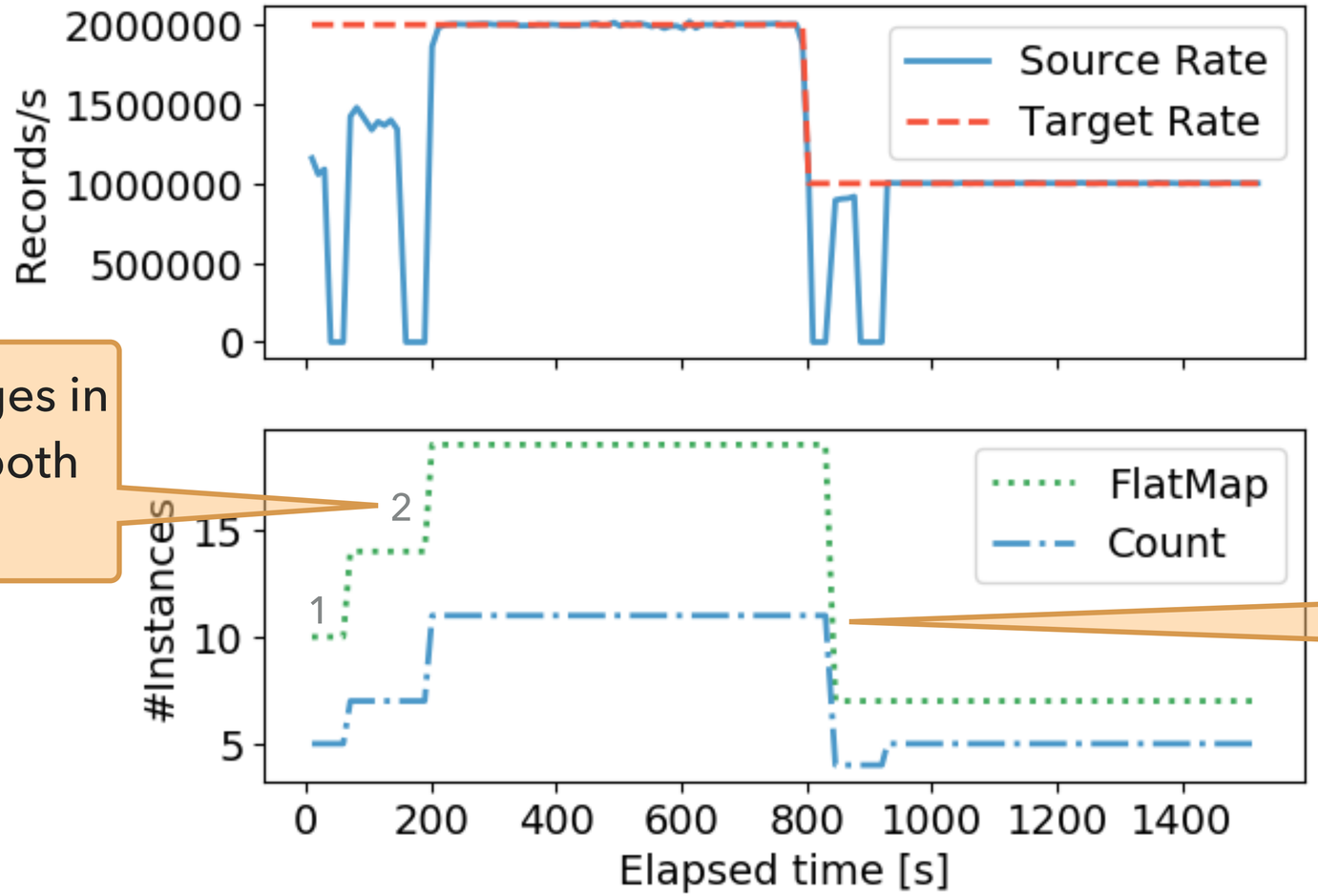


DS2 converges in **2 steps** for both operators

DS2 on Flink

Initially under-provisioned wordcount

Target rate: 2.000.000 rec/s, drops to half at 800s



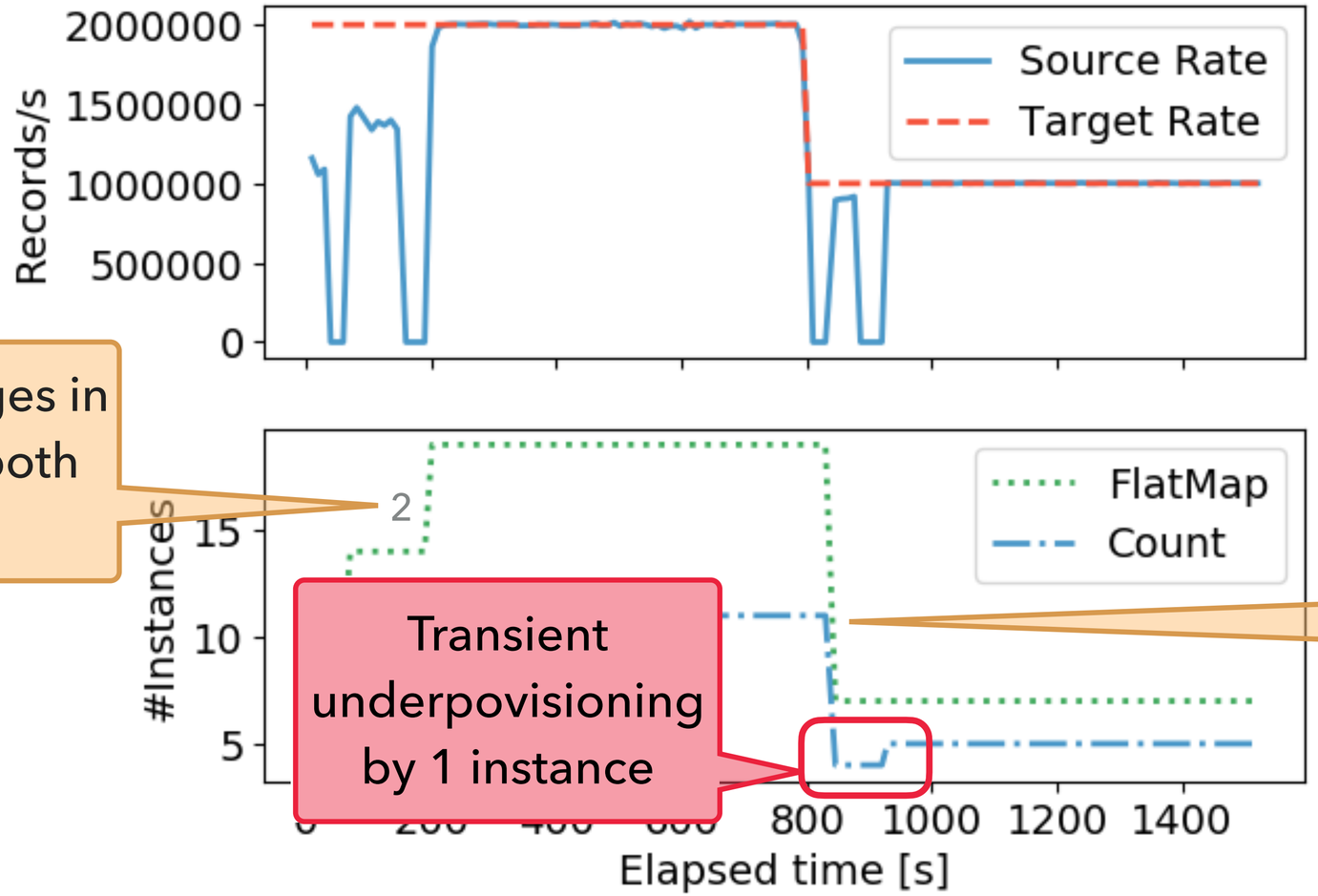
DS2 converges in **2 steps** for both operators

DS2 reacts **within 3s** when the target rate drops

DS2 on Flink

Initially under-provisioned wordcount

Target rate: 2.000.000 rec/s, drops to half at 800s

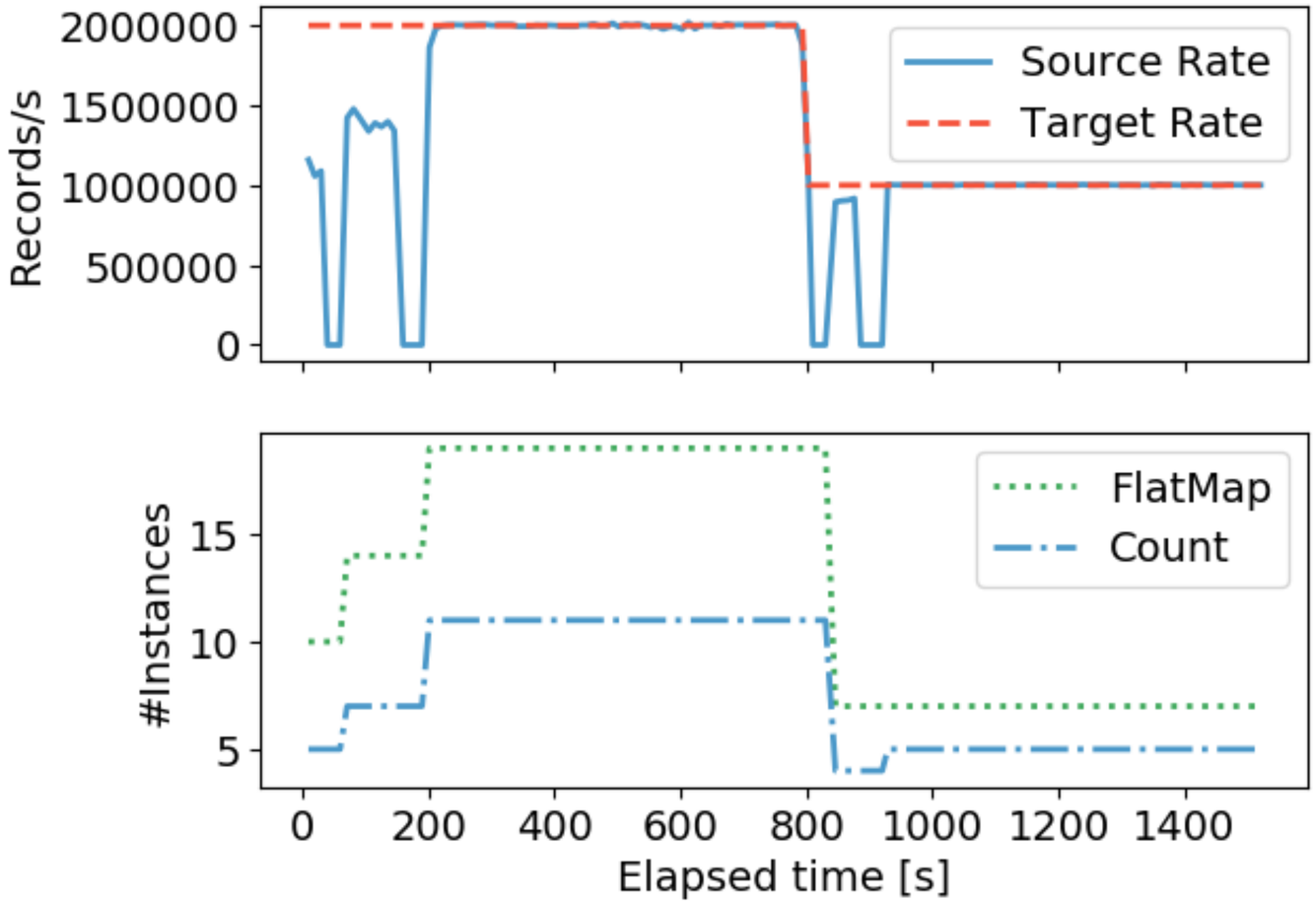


DS2 converges in **2 steps** for both operators

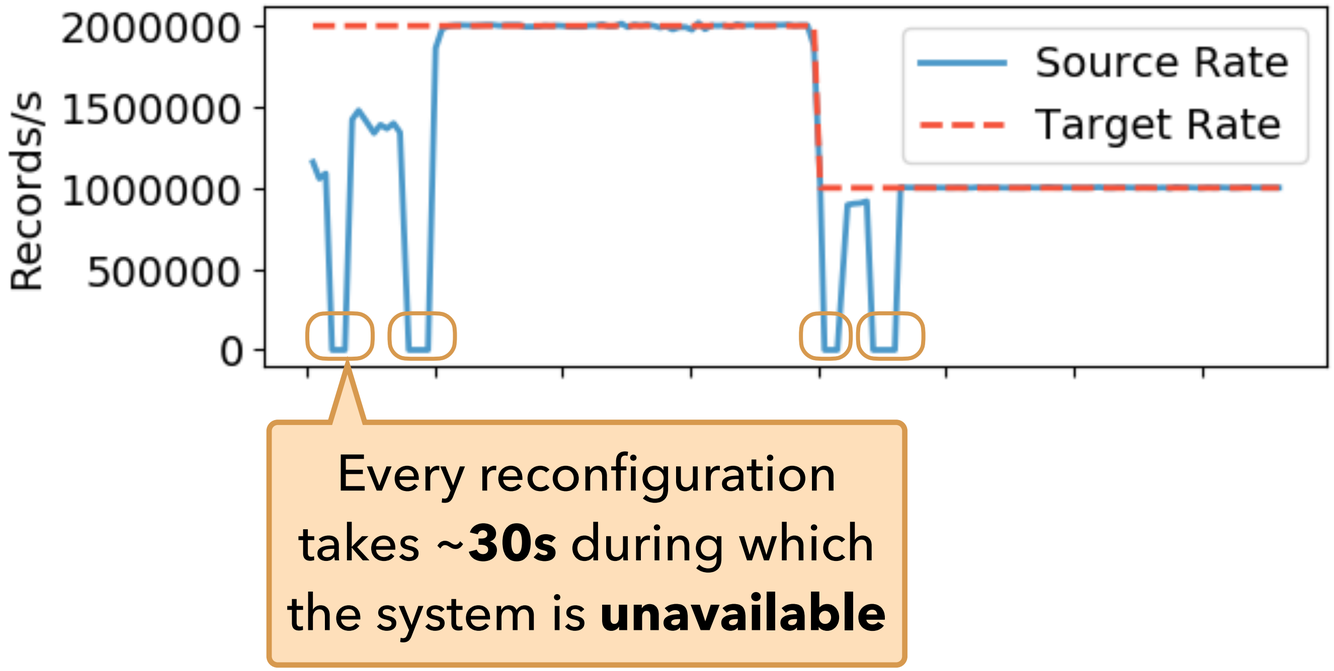
Transient underpovisioning by 1 instance

DS2 reacts **within 3s** when the target rate drops

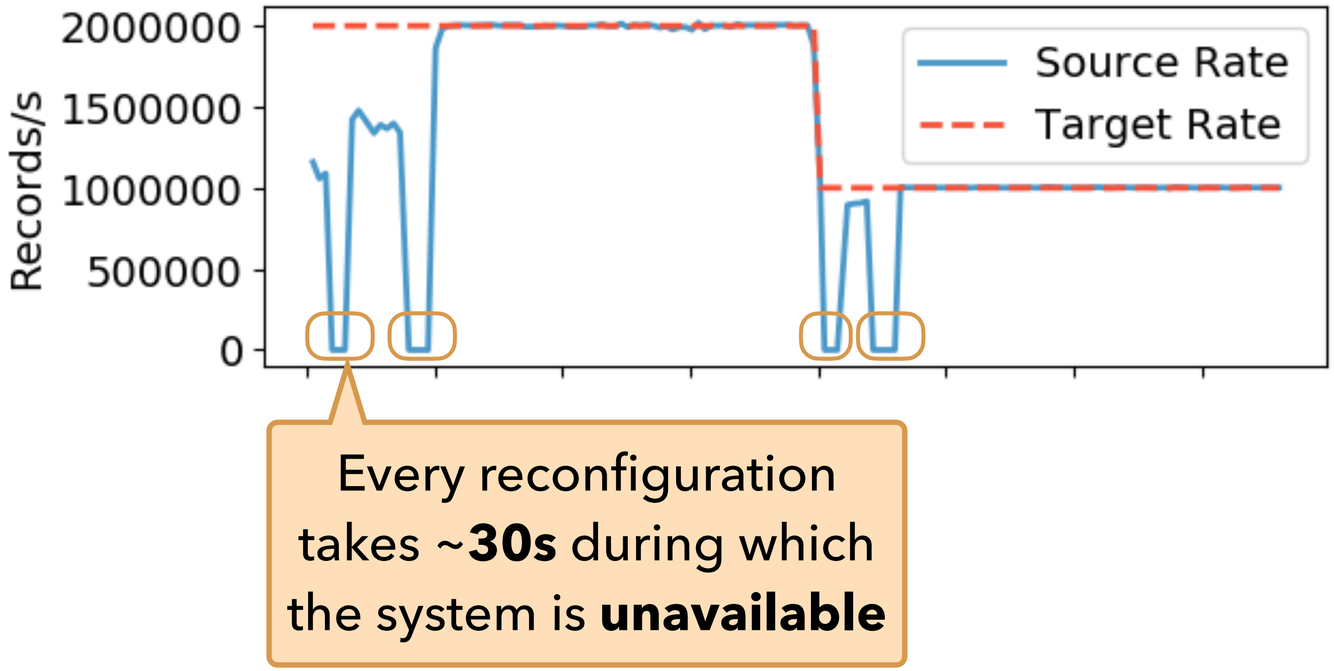
DS2 scaling actions on Apache Flink wordcount



DS2 scaling actions on Apache Flink wordcount



DS2 scaling actions on Apache Flink wordcount



Re-configuration requires **state migration** with correctness guarantees.

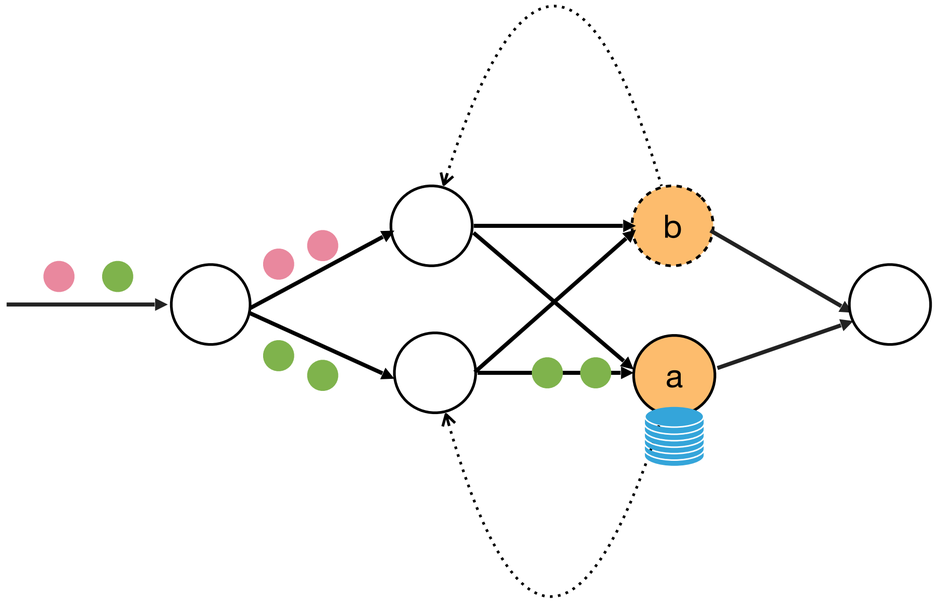
Elasticity mechanisms

Applying the reconfiguration

- **Stop-and-restart** - Dhalion (VLDB'17), DS2 (OSDI'18), Turbine (ICDE'20)
  - Halt the whole computation, take a state snapshot of all operators, restart
- **Partial pause and restart**
  - Temporarily block the affected dataflow subgraph only
- **Pro-active replication**
  - Maintain state replicas in multiple nodes for reconfiguration purposes

Partial-pause-and-restart

FUGU(DEBS'14), Seep(SIGMOD'13)

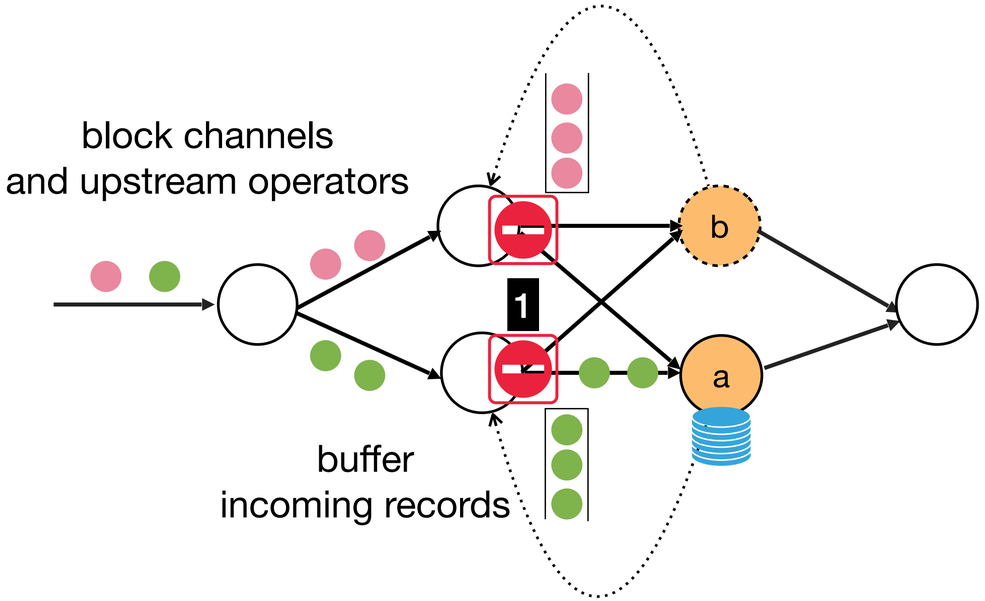


Migrating state from a to b

1. Pause a's upstream operators and start buffering events in their input channels.
2. Process all remaining events in a's input buffers and then extract its state.
3. Move a's state to b.
4. Operator b loads state and sends "restart" signal to upstream operators.

Partial-pause-and-restart

FUGU(DEBS'14), Seep(SIGMOD'13)

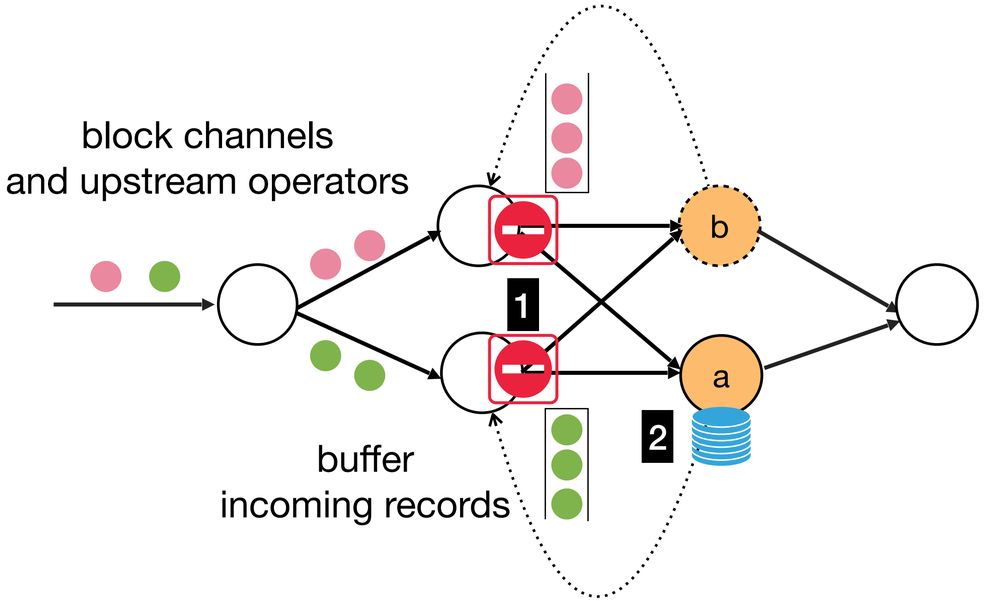


Migrating state from a to b

1. Pause a's upstream operators and start buffering events in their input channels.
2. Process all remaining events in a's input buffers and then extract its state.
3. Move a's state to b.
4. Operator b loads state and sends "restart" signal to upstream operators.

Partial-pause-and-restart

FUGU(DEBS'14), Seep(SIGMOD'13)

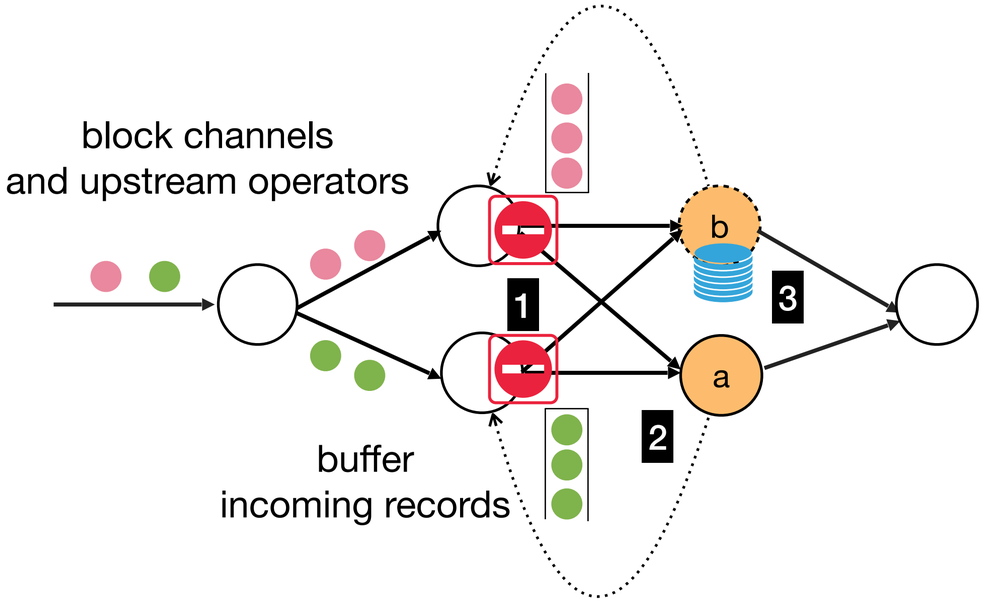


Migrating state from a to b

1. Pause a's upstream operators and start buffering events in their input channels.
2. Process all remaining events in a's input buffers and then extract its state.
3. Move a's state to b.
4. Operator b loads state and sends "restart" signal to upstream operators.

Partial-pause-and-restart

FUGU(DEBS'14), Seep(SIGMOD'13)

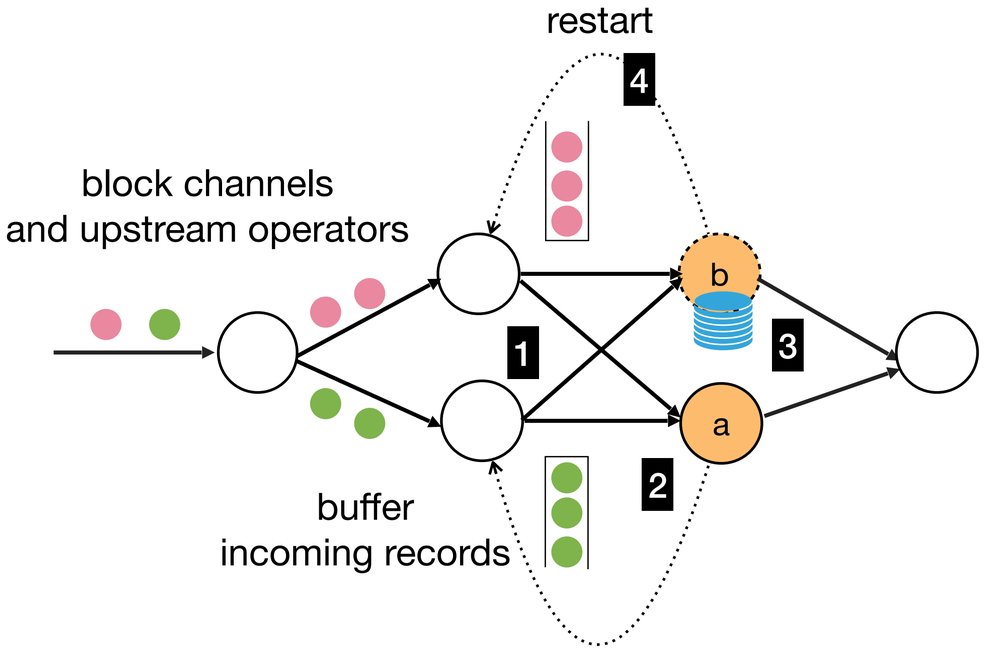


Migrating state from a to b

1. Pause a's upstream operators and start buffering events in their input channels.
2. Process all remaining events in a's input buffers and then extract its state.
3. Move a's state to b.
4. Operator b loads state and sends "restart" signal to upstream operators.

Partial-pause-and-restart

FUGU(DEBS'14), Seep(SIGMOD'13)

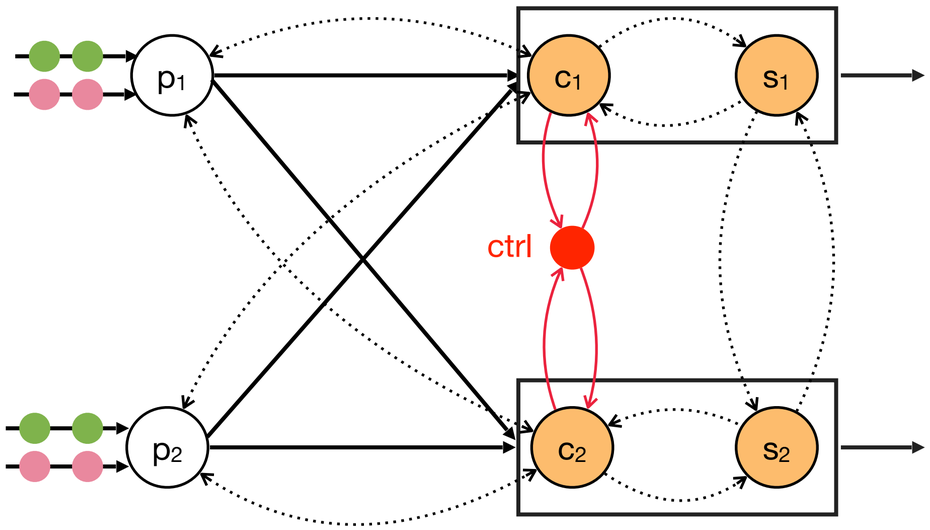


Migrating state from a to b

1. Pause a's upstream operators and start buffering events in their input channels.
2. Process all remaining events in a's input buffers and then extract its state.
3. Move a's state to b.
4. Operator b loads state and sends "restart" signal to upstream operators.

Partial-pause-and-restart

FLUX (ICDE'03)

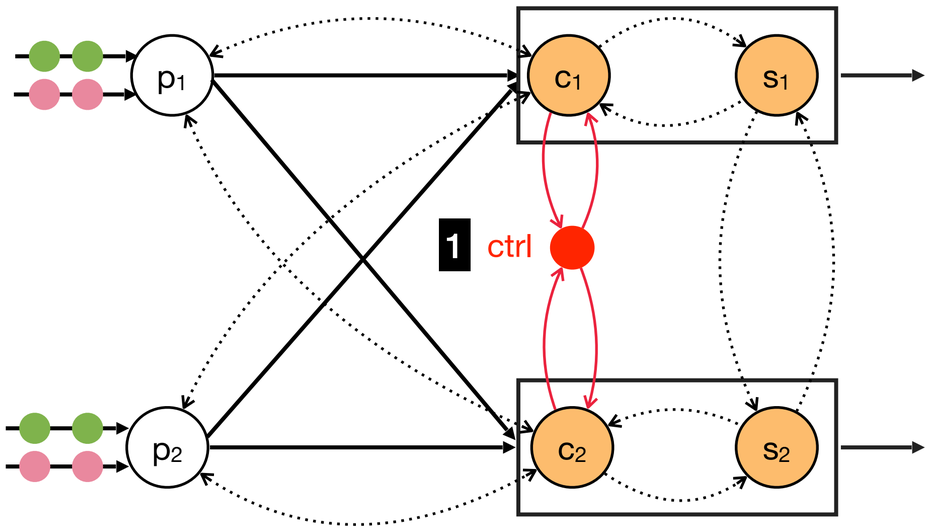


Migrating state from c_1 to c_2

1. The controller generates a list of partitions to move and informs c_1, c_2.
2. Upon receiving a move request:
   a. c_2 queues a receive-request with s_2
   b. c_1 broadcasts a pause to all upstream instances p_i
3. p_i receives the pause and marks partitions as stalled, stops consuming from its corresponding input buffer, and sends an ack to c_1
4. After c_1 receives all acks:
   a. s_1 transfers the partitions to s_2
   b. s_2 notifies c_2
   c. c_2 sends a restart signal upstream and to the controller

Partial-pause-and-restart

FLUX (ICDE'03)

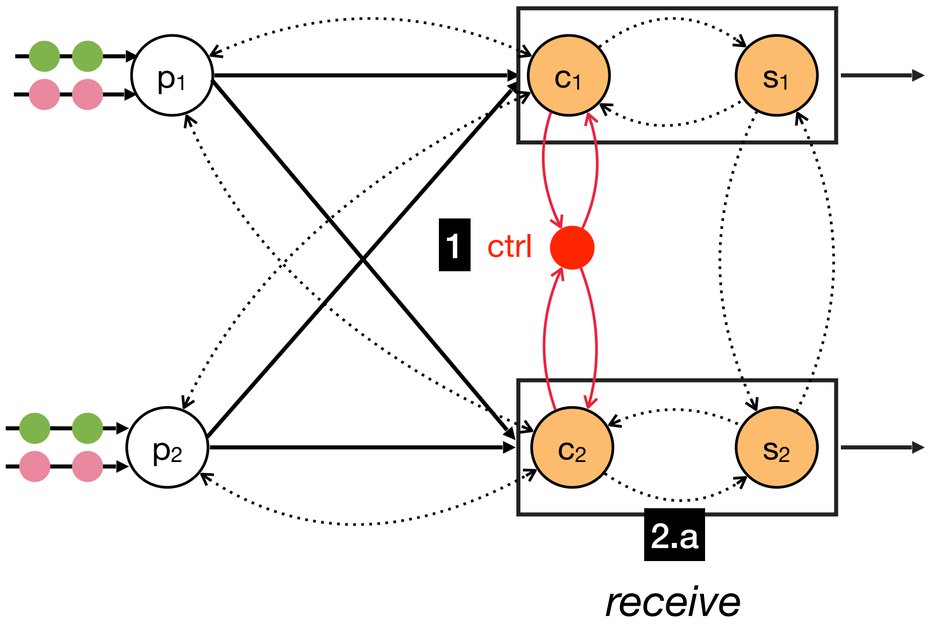


Migrating state from c_1 to c_2

1. The controller generates a list of partitions to move and informs c_1, c_2.
2. Upon receiving a move request:
   a. c_2 queues a receive-request with s_2
   b. c_1 broadcasts a pause to all upstream instances p_i
3. p_i receives the pause and marks partitions as stalled, stops consuming from its corresponding input buffer, and sends an ack to c_1
4. After c_1 receives all acks:
   a. s_1 transfers the partitions to s_2
   b. s_2 notifies c_2
   c. c_2 sends a restart signal upstream and to the controller

Partial-pause-and-restart

FLUX (ICDE'03)

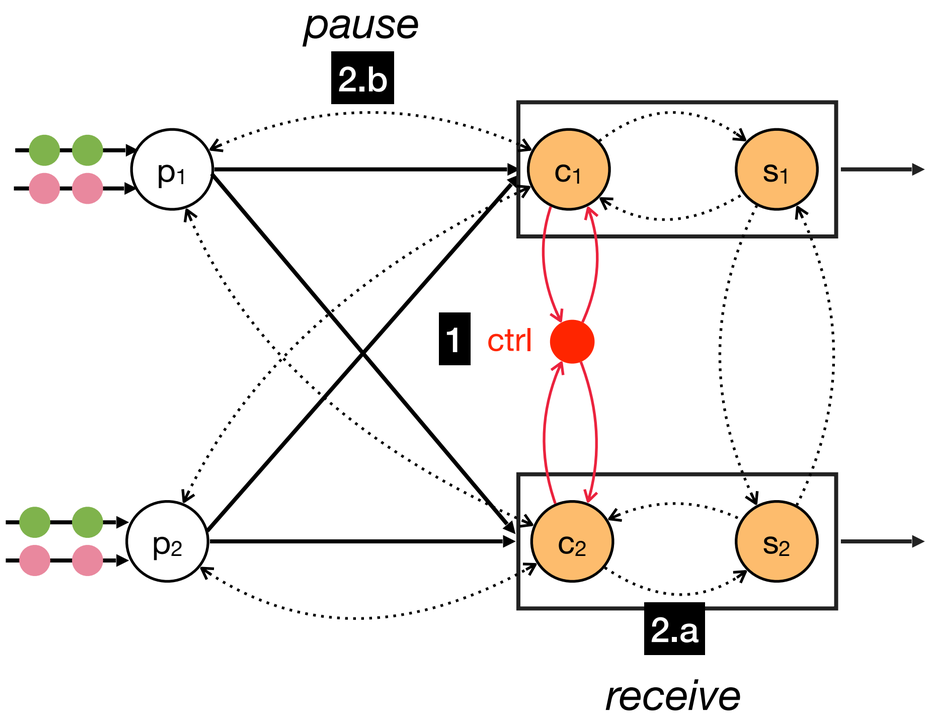


Migrating state from c_1 to c_2

1. The controller generates a list of partitions to move and informs c_1, c_2.
2. Upon receiving a move request:
   a. c_2 queues a receive-request with s_2
   b. c_1 broadcasts a pause to all upstream instances p_i
3. p_i receives the pause and marks partitions as stalled, stops consuming from its corresponding input buffer, and sends an ack to c_1
4. After c_1 receives all acks:
   a. s_1 transfers the partitions to s_2
   b. s_2 notifies c_2
   c. c_2 sends a restart signal upstream and to the controller

Partial-pause-and-restart

FLUX (ICDE'03)

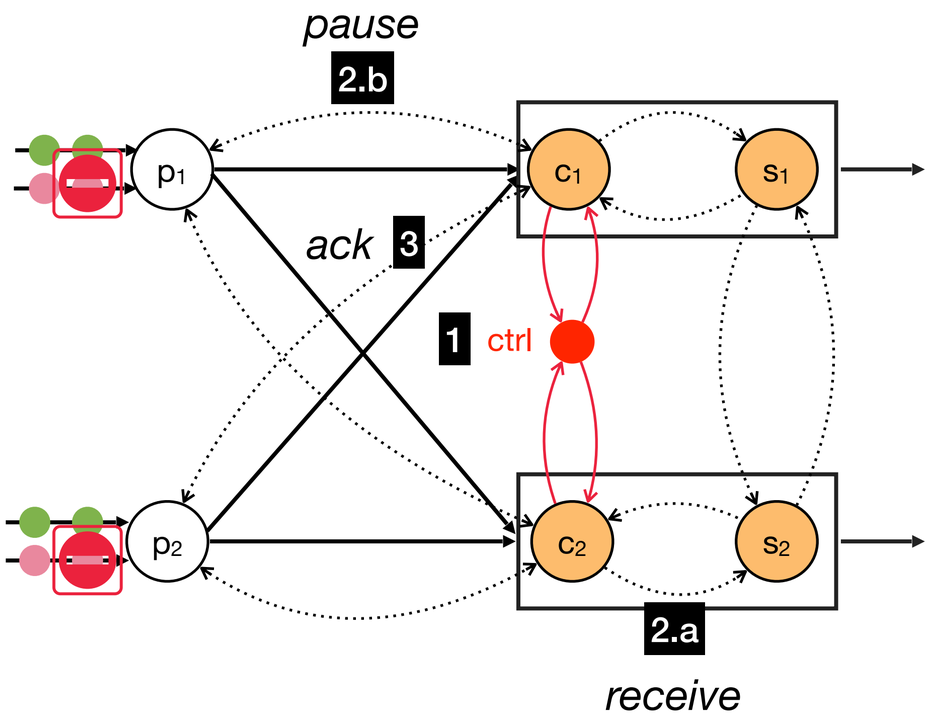


Migrating state from c_1 to c_2

1. The controller generates a list of partitions to move and informs c_1, c_2.
2. Upon receiving a move request:
   a. c_2 queues a receive-request with s_2
   b. c_1 broadcasts a pause to all upstream instances p_i
3. p_i receives the pause and marks partitions as stalled, stops consuming from its corresponding input buffer, and sends an ack to c_1
4. After c_1 receives all acks:
   a. s_1 transfers the partitions to s_2
   b. s_2 notifies c_2
   c. c_2 sends a restart signal upstream and to the controller

Partial-pause-and-restart

FLUX (ICDE'03)

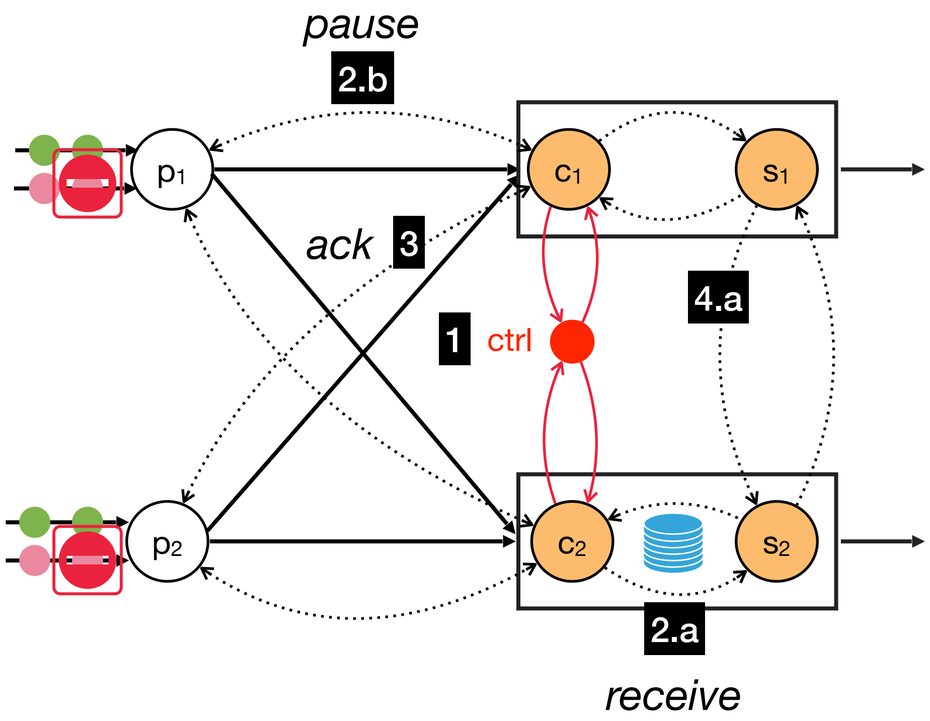


Migrating state from c_1 to c_2

1. The controller generates a list of partitions to move and informs c_1, c_2.
2. Upon receiving a move request:
   a. c_2 queues a receive-request with s_2
   b. c_1 broadcasts a pause to all upstream instances p_i
3. p_i receives the pause and marks partitions as stalled, stops consuming from its corresponding input buffer, and sends an ack to c_1
4. After c_1 receives all acks:
   a. s_1 transfers the partitions to s_2
   b. s_2 notifies c_2
   c. c_2 sends a restart signal upstream and to the controller

Partial-pause-and-restart

FLUX (ICDE'03)

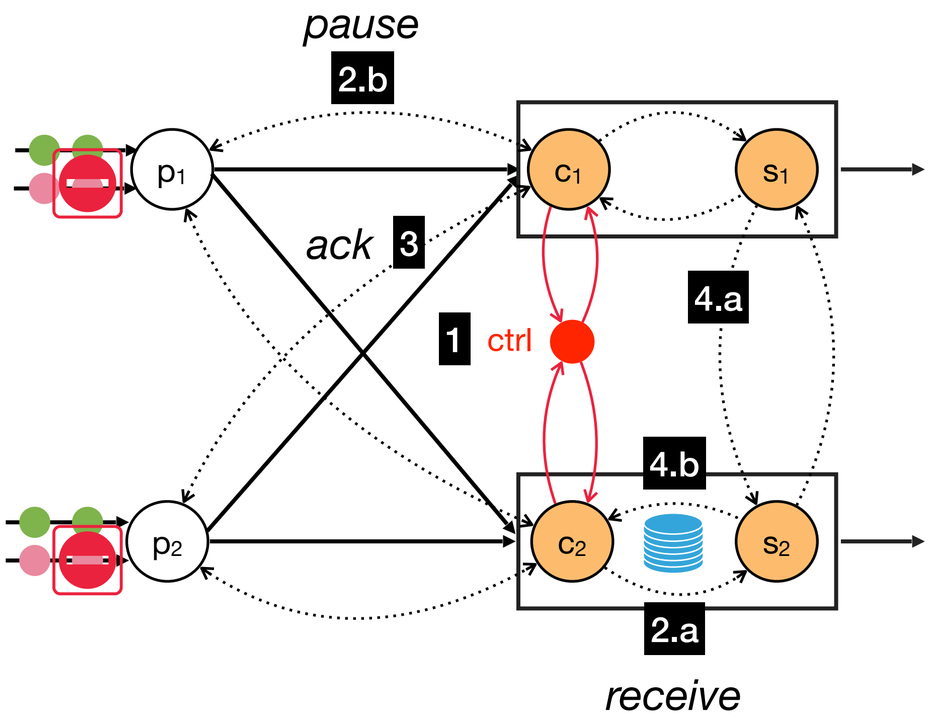


Migrating state from c_1 to c_2

1. The controller generates a list of partitions to move and informs c_1, c_2.
2. Upon receiving a move request:
   a. c_2 queues a receive-request with s_2
   b. c_1 broadcasts a pause to all upstream instances p_i
3. p_i receives the pause and marks partitions as stalled, stops consuming from its corresponding input buffer, and sends an ack to c_1
4. After c_1 receives all acks:
   a. s_1 transfers the partitions to s_2
   b. s_2 notifies c_2
   c. c_2 sends a restart signal upstream and to the controller

Partial-pause-and-restart

FLUX (ICDE'03)

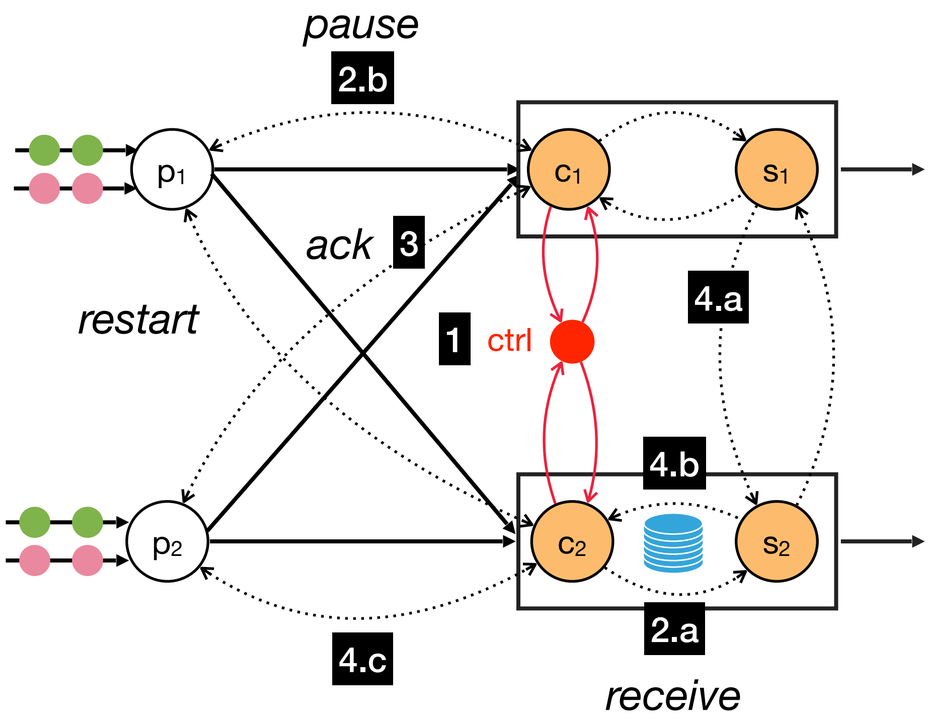


Migrating state from c_1 to c_2

1. The controller generates a list of partitions to move and informs c_1, c_2.
2. Upon receiving a move request:
   a. c_2 queues a receive-request with s_2
   b. c_1 broadcasts a pause to all upstream instances p_i
3. p_i receives the pause and marks partitions as stalled, stops consuming from its corresponding input buffer, and sends an ack to c_1
4. After c_1 receives all acks:
   a. s_1 transfers the partitions to s_2
   b. s_2 notifies c_2
   c. c_2 sends a restart signal upstream and to the controller

Partial-pause-and-restart

FLUX (ICDE'03)

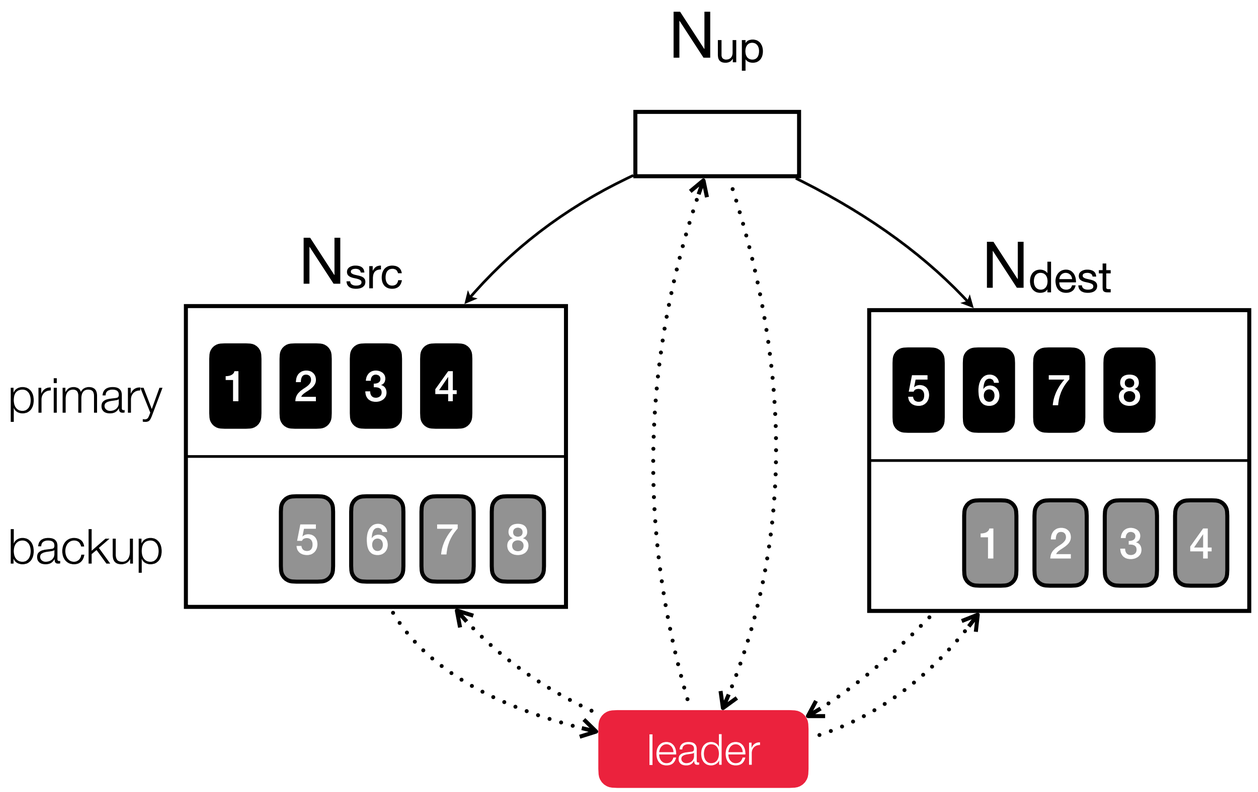


Migrating state from c_1 to c_2

1. The controller generates a list of partitions to move and informs c_1, c_2.
2. Upon receiving a move request:
   a. c_2 queues a receive-request with s_2
   b. c_1 broadcasts a pause to all upstream instances p_i
3. p_i receives the pause and marks partitions as stalled, stops consuming from its corresponding input buffer, and sends an ack to c_1
4. After c_1 receives all acks:
   a. s_1 transfers the partitions to s_2
   b. s_2 notifies c_2
   c. c_2 sends a restart signal upstream and to the controller

Pro-active Replication

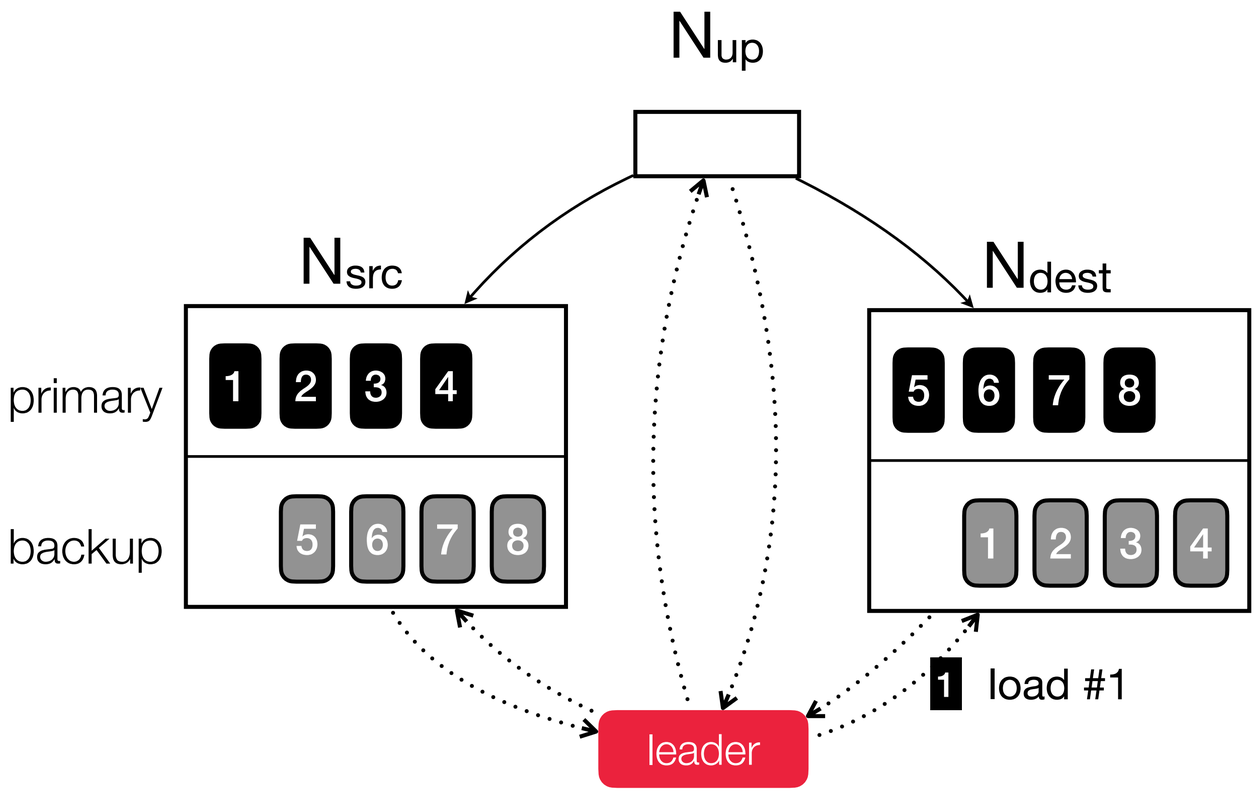
ChronoStream (ICDE'15), Rhino (SIGMOD'20)



1. The leader instructs N_{dest} to load slice #1.
2. N_{dest} loads slice #1 and sends back ack and the slice progress. In the meantime, N_{src} keeps processing events destined for slice #1.
3. The leader notifies upstream operators to re-play events according to the progress metric provided by N_{dest}.
4. Upstream nodes receive the message and re-route events to N_{dest}.
5. The leader notifies N_{src} that the transfer is complete.
6. N_{src} consumes remaining data in its buffers and moves slice #1 to the backup group.

Pro-active Replication

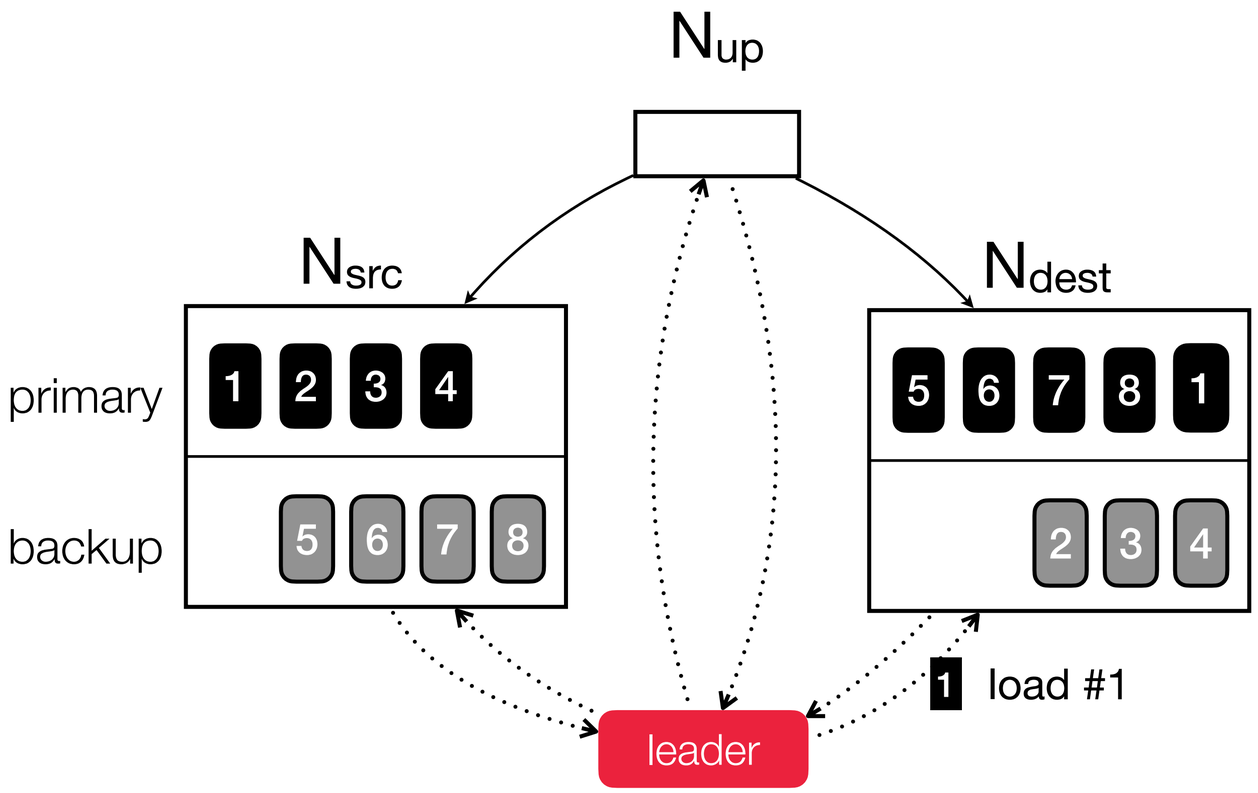
ChronoStream (ICDE'15), Rhino (SIGMOD'20)



1. The leader instructs N_{dest} to load slice #1.
2. N_{dest} loads slice #1 and sends back ack and the slice progress. In the meantime, N_{src} keeps processing events destined for slice #1.
3. The leader notifies upstream operators to re-play events according to the progress metric provided by N_{dest}.
4. Upstream nodes receive the message and re-route events to N_{dest}.
5. The leader notifies N_{src} that the transfer is complete.
6. N_{src} consumes remaining data in its buffers and moves slice #1 to the backup group.

Pro-active Replication

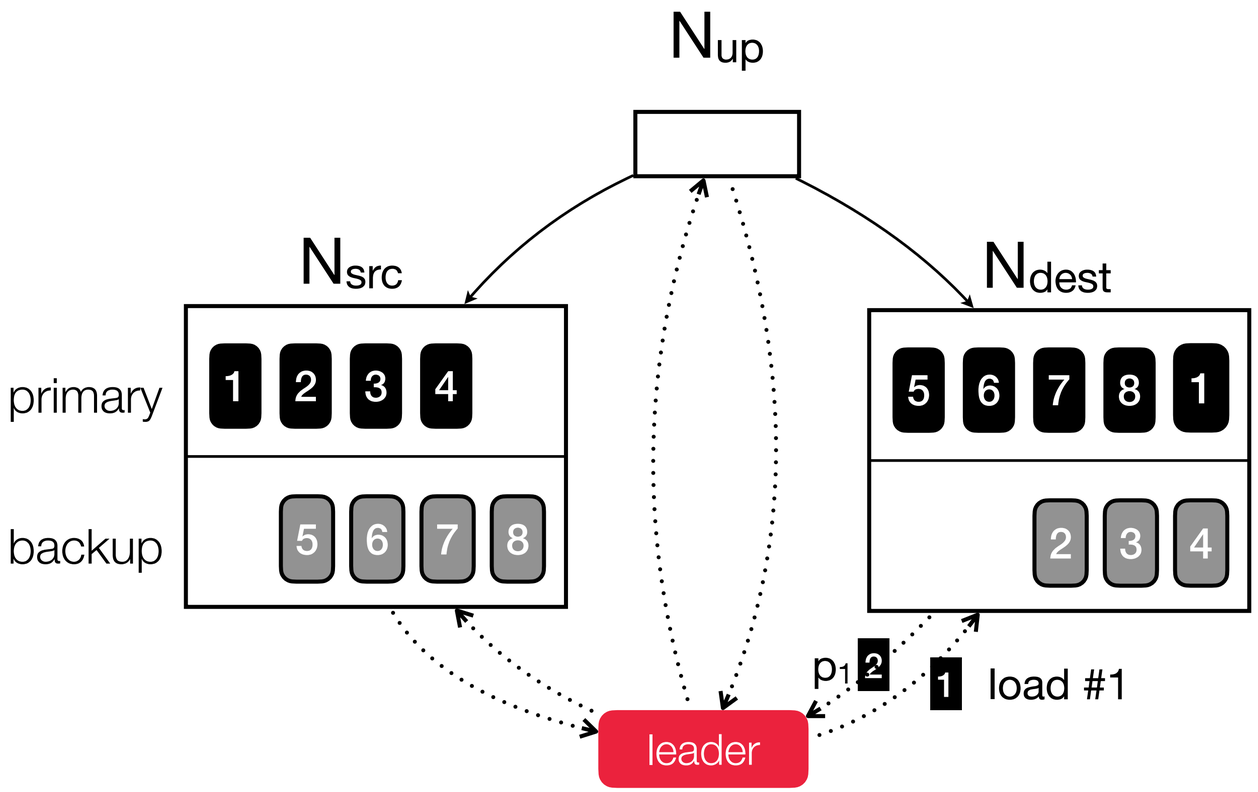
ChronoStream (ICDE'15), Rhino (SIGMOD'20)



1. The leader instructs N_{dest} to load slice #1.
2. N_{dest} loads slice #1 and sends back ack and the slice progress. In the meantime, N_{src} keeps processing events destined for slice #1.
3. The leader notifies upstream operators to re-play events according to the progress metric provided by N_{dest}.
4. Upstream nodes receive the message and re-route events to N_{dest}.
5. The leader notifies N_{src} that the transfer is complete.
6. N_{src} consumes remaining data in its buffers and moves slice #1 to the backup group.

Pro-active Replication

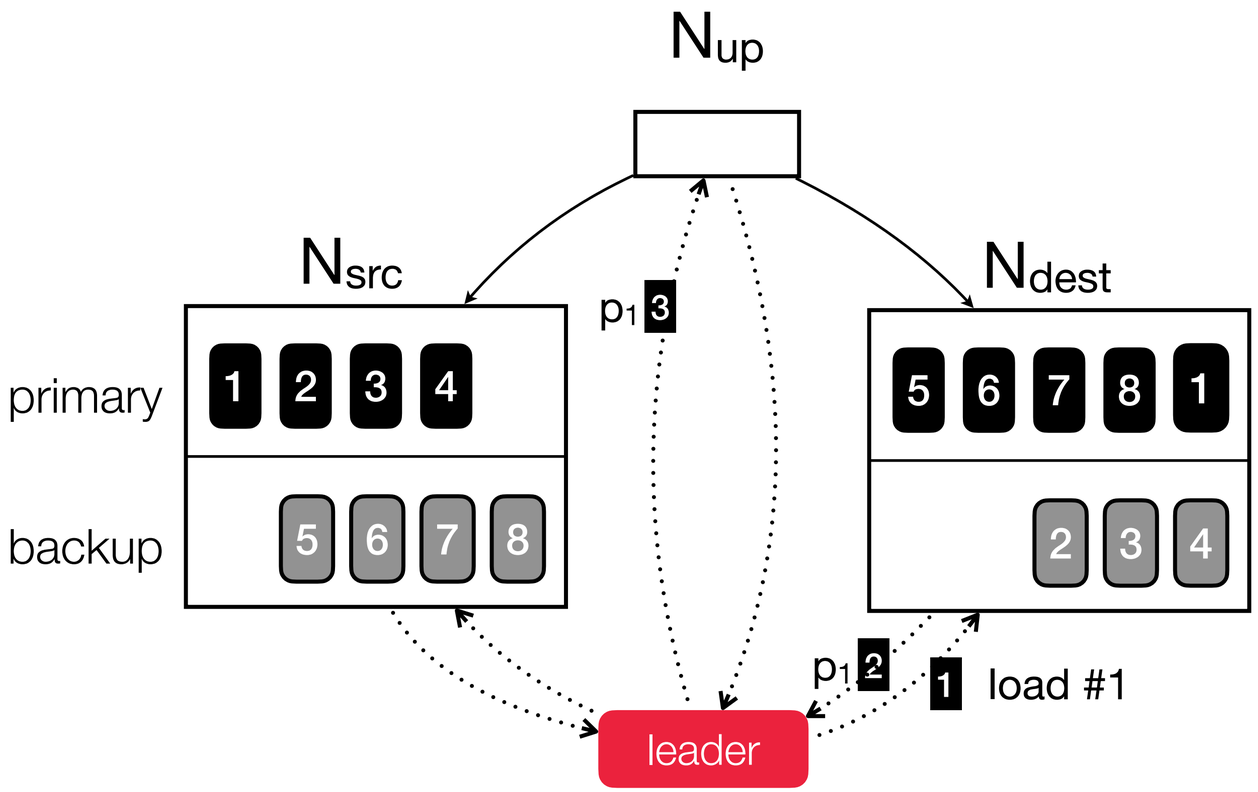
ChronoStream (ICDE'15), Rhino (SIGMOD'20)



1. The leader instructs N_{dest} to load slice #1.
2. N_{dest} loads slice #1 and sends back ack and the slice progress. In the meantime, N_{src} keeps processing events destined for slice #1.
3. The leader notifies upstream operators to re-play events according to the progress metric provided by N_{dest}.
4. Upstream nodes receive the message and re-route events to N_{dest}.
5. The leader notifies N_{src} that the transfer is complete.
6. N_{src} consumes remaining data in its buffers and moves slice #1 to the backup group.

Pro-active Replication

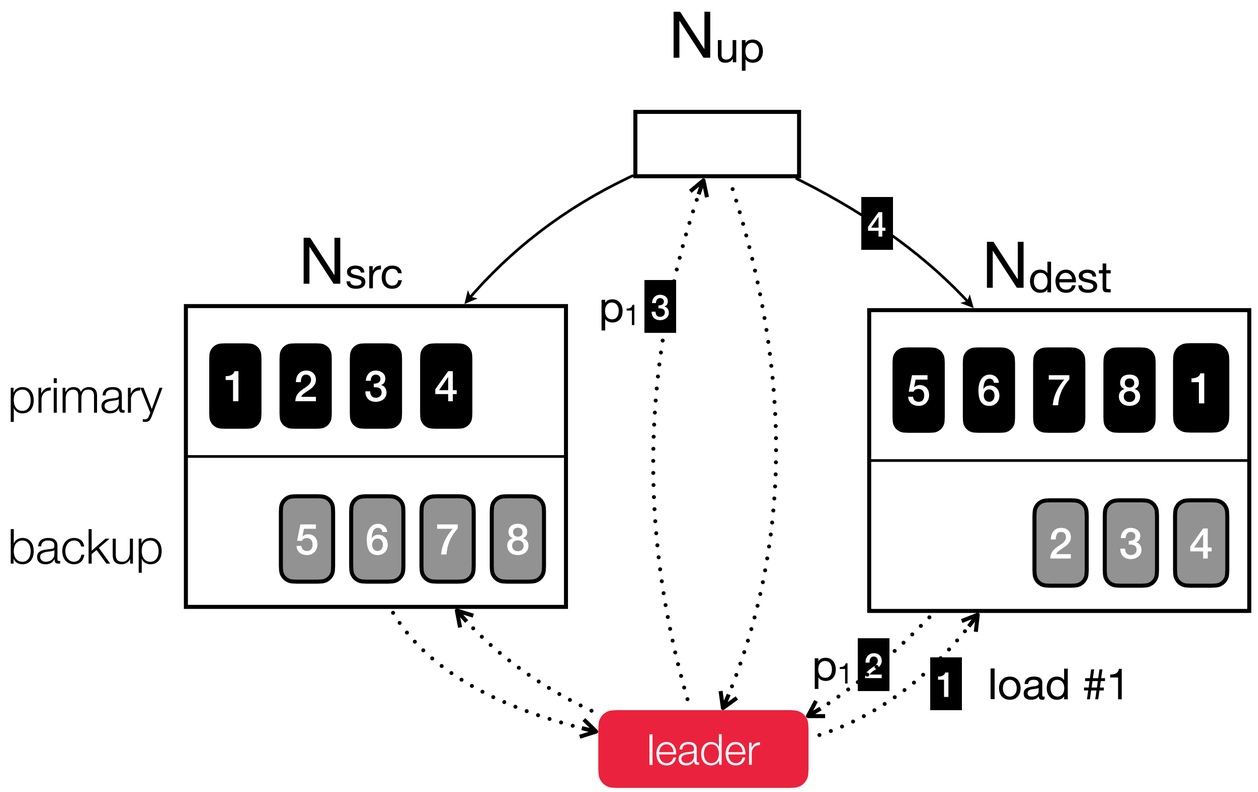
ChronoStream (ICDE'15), Rhino (SIGMOD'20)



1. The leader instructs N_{dest} to load slice #1.
2. N_{dest} loads slice #1 and sends back ack and the slice progress. In the meantime, N_{src} keeps processing events destined for slice #1.
3. The leader notifies upstream operators to re-play events according to the progress metric provided by N_{dest}.
4. Upstream nodes receive the message and re-route events to N_{dest}.
5. The leader notifies N_{src} that the transfer is complete.
6. N_{src} consumes remaining data in its buffers and moves slice #1 to the backup group.

Pro-active Replication

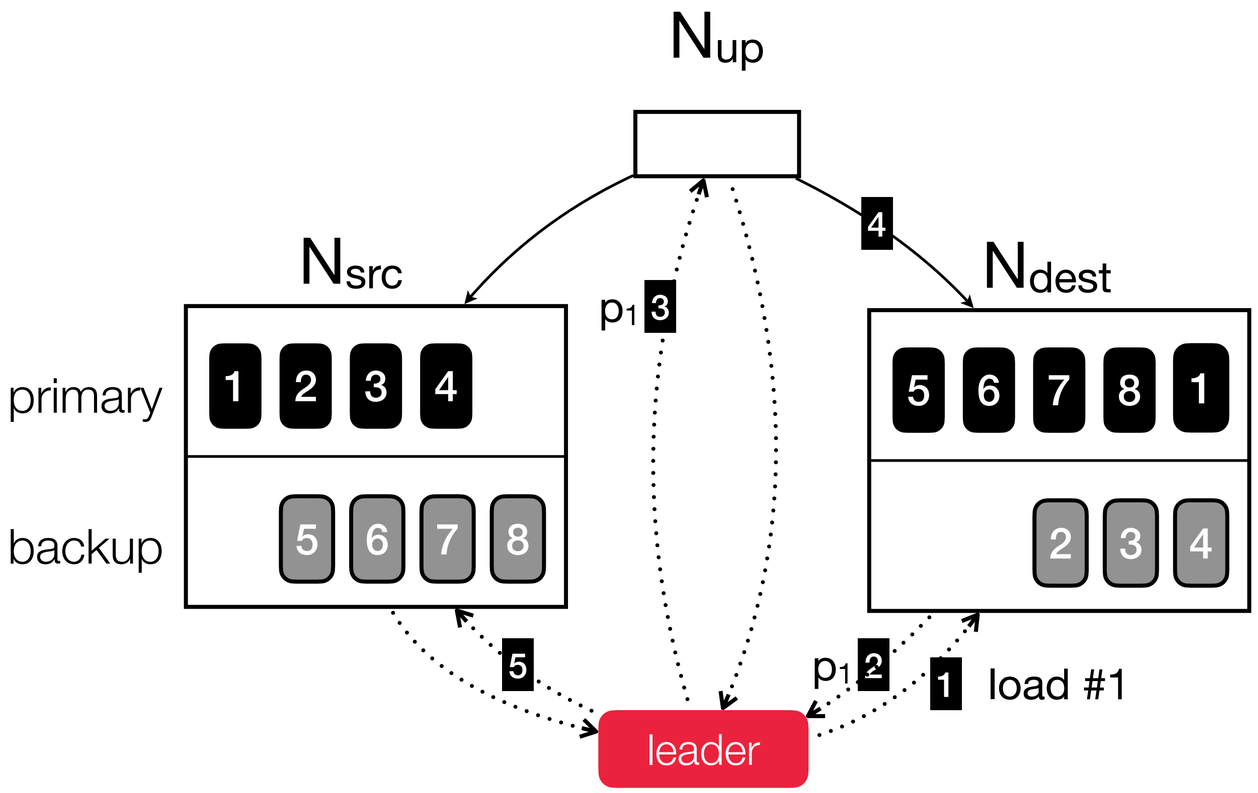
ChronoStream (ICDE'15), Rhino (SIGMOD'20)



1. The leader instructs N_{dest} to load slice #1.
2. N_{dest} loads slice #1 and sends back ack and the slice progress. In the meantime, N_{src} keeps processing events destined for slice #1.
3. The leader notifies upstream operators to re-play events according to the progress metric provided by N_{dest}.
4. Upstream nodes receive the message and re-route events to N_{dest}.
5. The leader notifies N_{src} that the transfer is complete.
6. N_{src} consumes remaining data in its buffers and moves slice #1 to the backup group.

Pro-active Replication

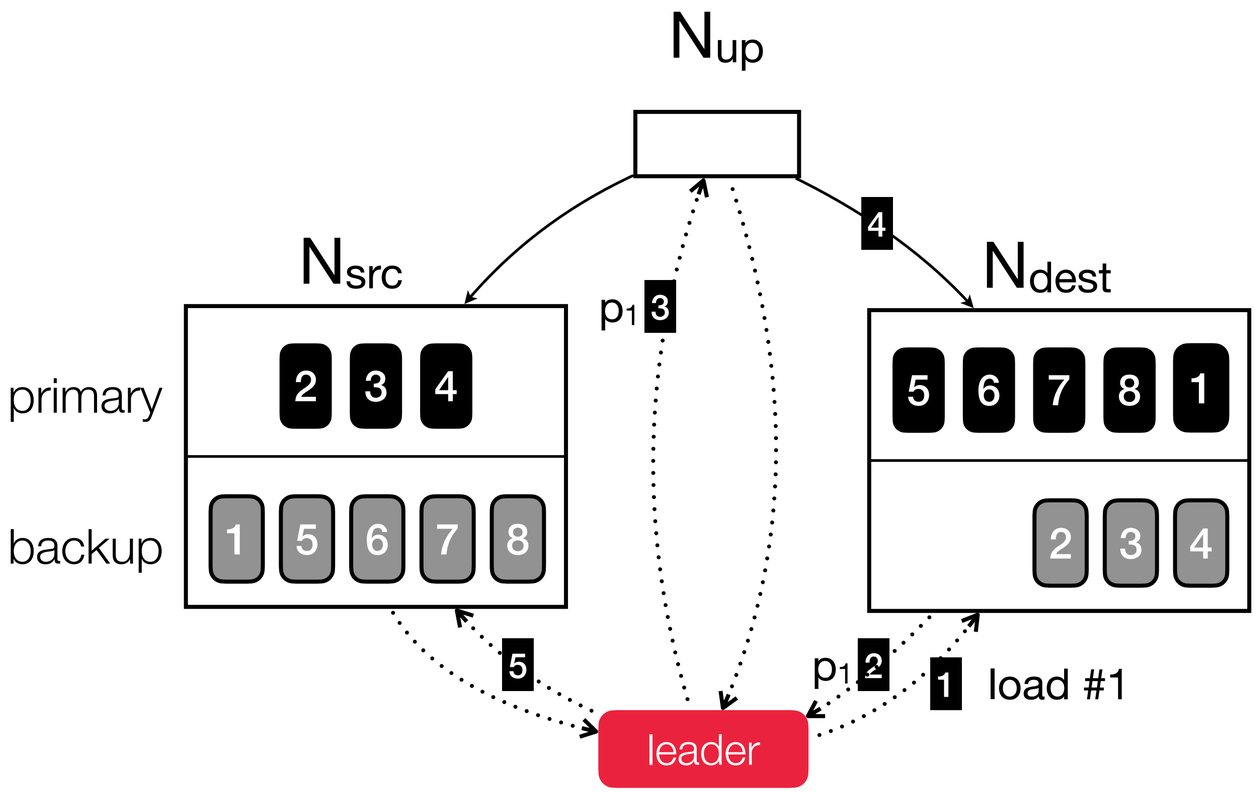
ChronoStream (ICDE'15), Rhino (SIGMOD'20)



1. The leader instructs N_{dest} to load slice #1.
2. N_{dest} loads slice #1 and sends back ack and the slice progress. In the meantime, N_{src} keeps processing events destined for slice #1.
3. The leader notifies upstream operators to re-play events according to the progress metric provided by N_{dest}.
4. Upstream nodes receive the message and re-route events to N_{dest}.
5. The leader notifies N_{src} that the transfer is complete.
6. N_{src} consumes remaining data in its buffers and moves slice #1 to the backup group.

Pro-active Replication

ChronoStream (ICDE'15), Rhino (SIGMOD'20)



1. The leader instructs N_{dest} to load slice #1.
2. N_{dest} loads slice #1 and sends back ack and the slice progress. In the meantime, N_{src} keeps processing events destined for slice #1.
3. The leader notifies upstream operators to re-play events according to the progress metric provided by N_{dest}.
4. Upstream nodes receive the message and re-route events to N_{dest}.
5. The leader notifies N_{src} that the transfer is complete.
6. N_{src} consumes remaining data in its buffers and moves slice #1 to the backup group.

State transfer strategies

All-at-once

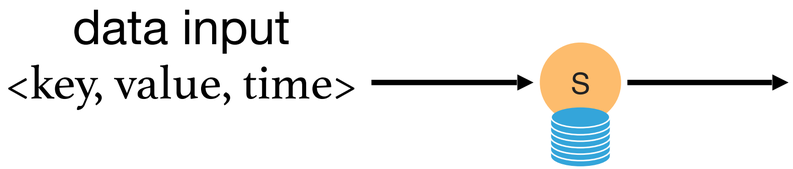
- Move state to be migrated in one operation
- High latency during migration if the state is large

Progressive

- Move state to be migrated in smaller pieces, e.g. key-by-key
- It enables interleaving state transfer with processing
- Migration duration might increase

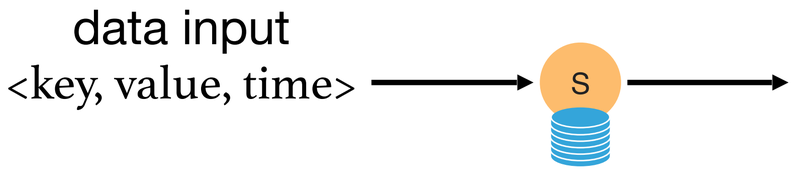
Progressive State migration

Megaphone (VLDB'19)

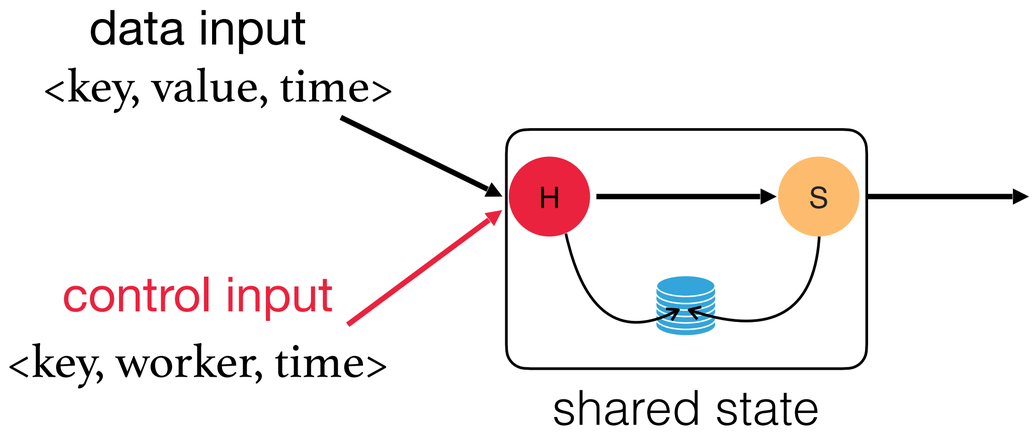


Progressive State migration

Megaphone (VLDB'19)



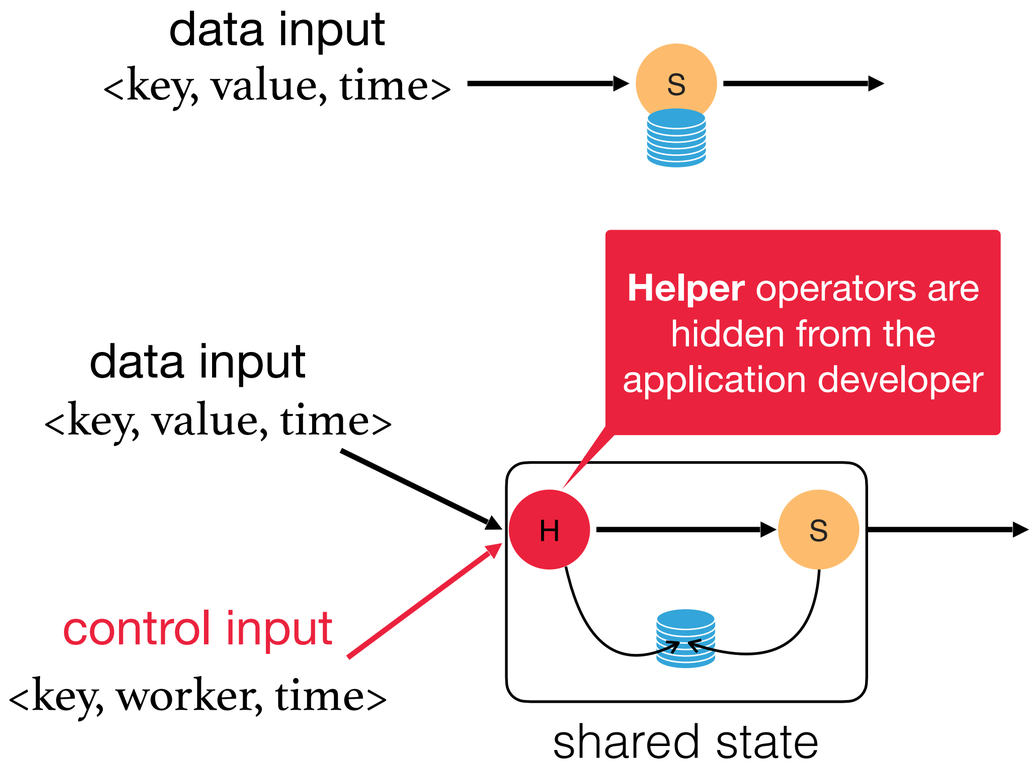
Each stateful operator is augmented with a **helper upstream operator** which accepts a control stream as input



Control inputs have **timestamps** and participate in the progress protocol (e.g. advance and propagate watermarks)

Progressive State migration

Megaphone (VLDB'19)

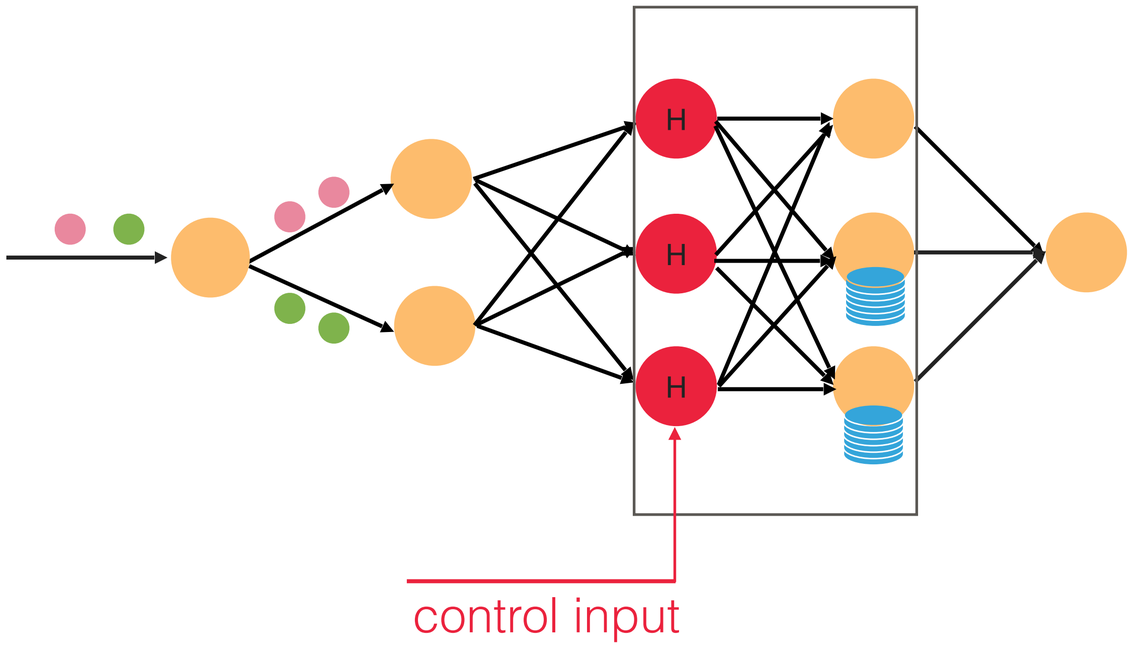


Each stateful operator is augmented with a **helper upstream operator** which accepts a control stream as input

Control inputs have **timestamps** and participate in the progress protocol (e.g. advance and propagate watermarks)

Progressive State migration

Megaphone (VLDB'19)

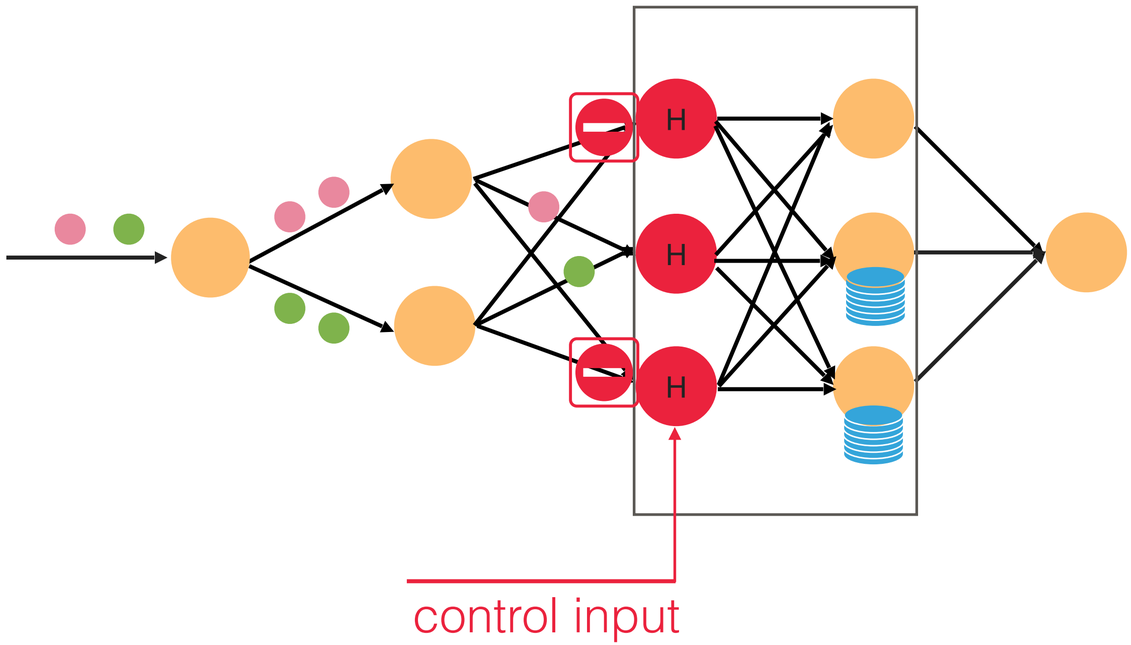


Helper operators:

- **buffer** data that cannot be safely routed yet and configuration commands that cannot yet be applied
- check the **frontier (watermark)** at the output of the stateful operator to **ensure only complete state is migrated**

Progressive State migration

Megaphone (VLDB'19)

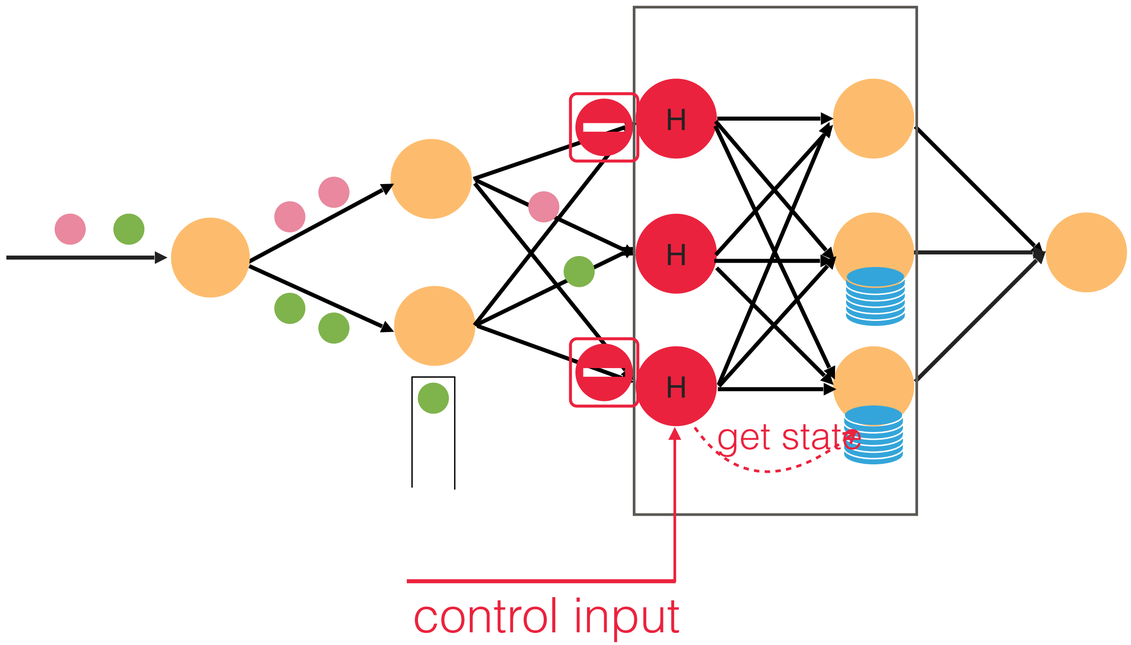


Helper operators:

- **buffer** data that cannot be safely routed yet and configuration commands that cannot yet be applied
- check the **frontier (watermark)** at the output of the stateful operator to **ensure only complete state is migrated**

Progressive State migration

Megaphone (VLDB'19)

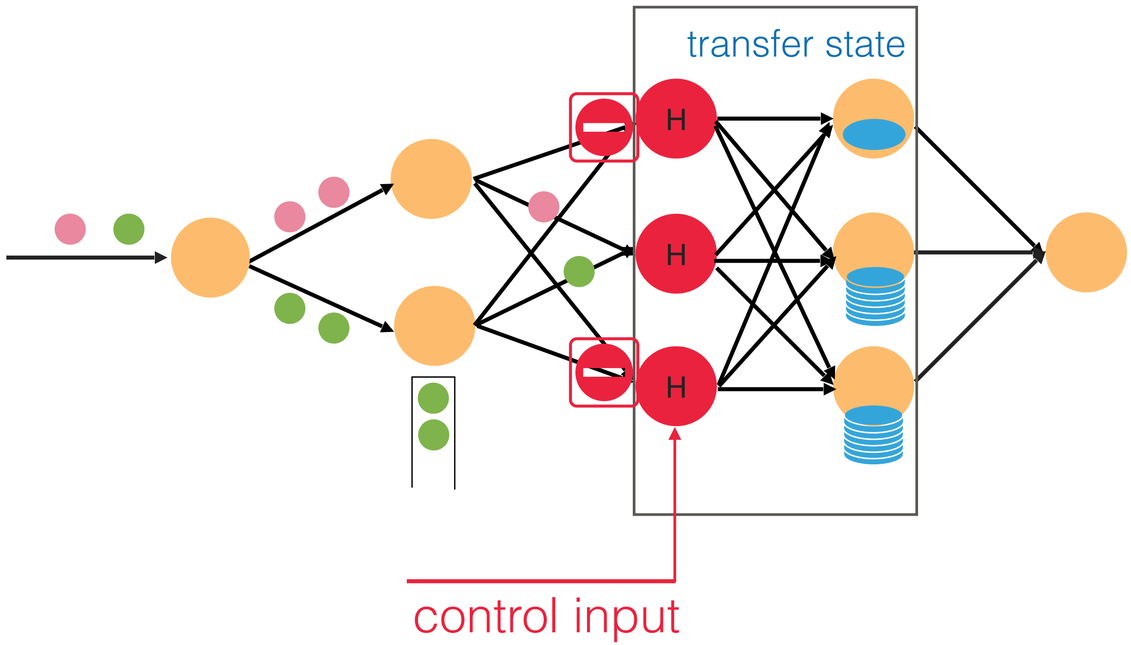


Helper operators:

- **buffer** data that cannot be safely routed yet and configuration commands that cannot yet be applied
- check the **frontier (watermark)** at the output of the stateful operator to **ensure only complete state is migrated**

Progressive State migration

Megaphone (VLDB'19)

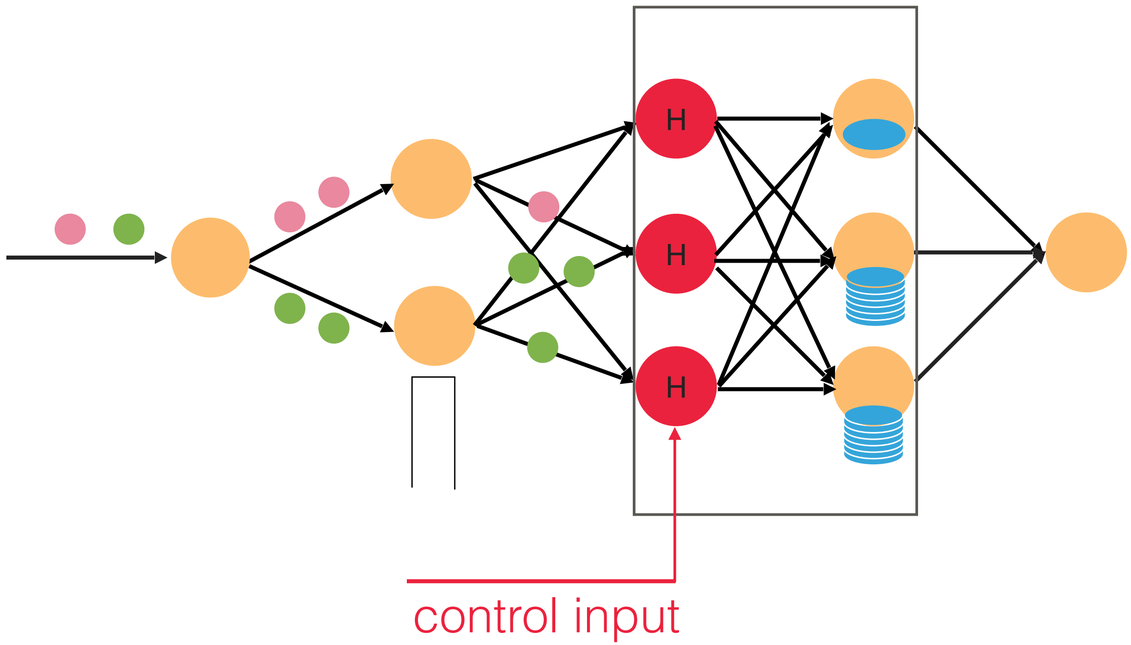


Helper operators:

- **buffer** data that cannot be safely routed yet and configuration commands that cannot yet be applied
- check the **frontier (watermark)** at the output of the stateful operator to **ensure only complete state is migrated**

Progressive State migration

Megaphone (VLDB'19)



Helper operators:

- **buffer** data that cannot be safely routed yet and configuration commands that cannot yet be applied
- check the **frontier (watermark)** at the output of the stateful operator to **ensure only complete state is migrated**

Lecture references

- Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava Dimitrova, Matthew Forshaw, and Timothy Roscoe. **Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows**. (OSDI'18).

- Moritz Hoffmann, Andrea Lattuada, Frank McSherry, Vasiliki Kalavri, John Liagouris, Timothy Roscoe. **Megaphone: Latency-conscious state migration for distributed streaming dataflows**. (VLDB 2019).