SIGMOD 2020 Tutorial

Beyond Analytics

The Evolution of Stream Processing Systems

Load management & Elasticity

Paris Carbone, Marios Fragkoulis, Vasiliki Kalavri, Asterios Katsifodimos

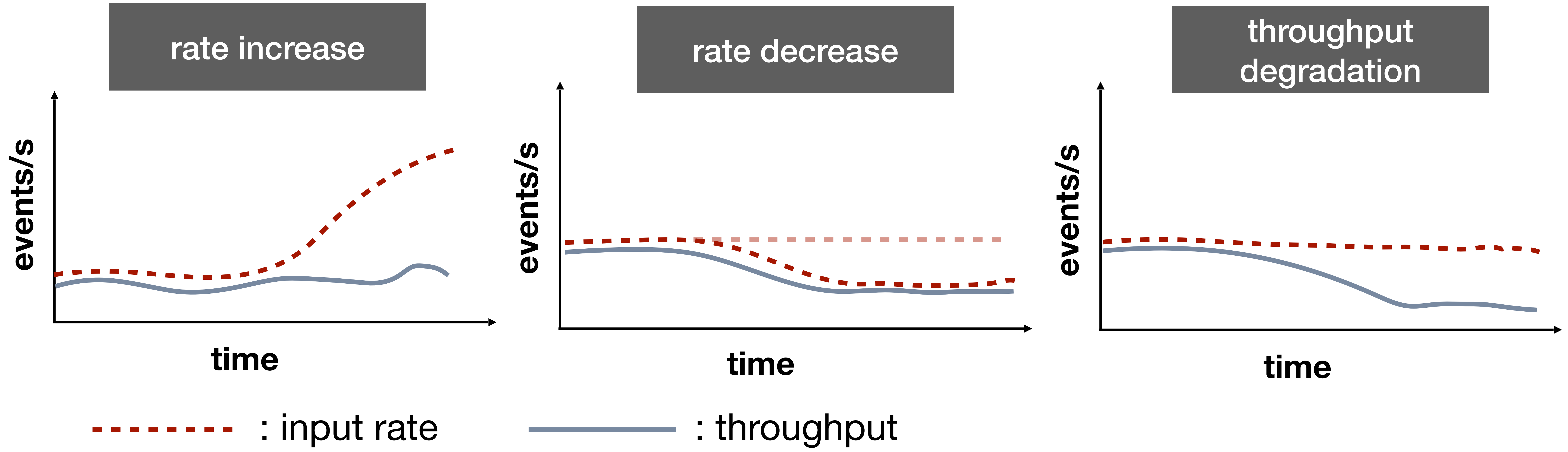
Slides: **streaming-research.github.io/Tutorial-SIGMOD-2020**

Tutorial overview

- Part I: Introduction & Fundamentals (Vasia)
- Part II: Time, Order, & Progress (Marios)
- Part III: State Management (Paris)
- Part IV: Fault Recovery & High Availability (Marios)
- Part V: Load Management & Elasticity (Vasia)
- Part VI: Prospects (All)

Streaming applications are long-running

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



Load management & reconfiguration

Agenda

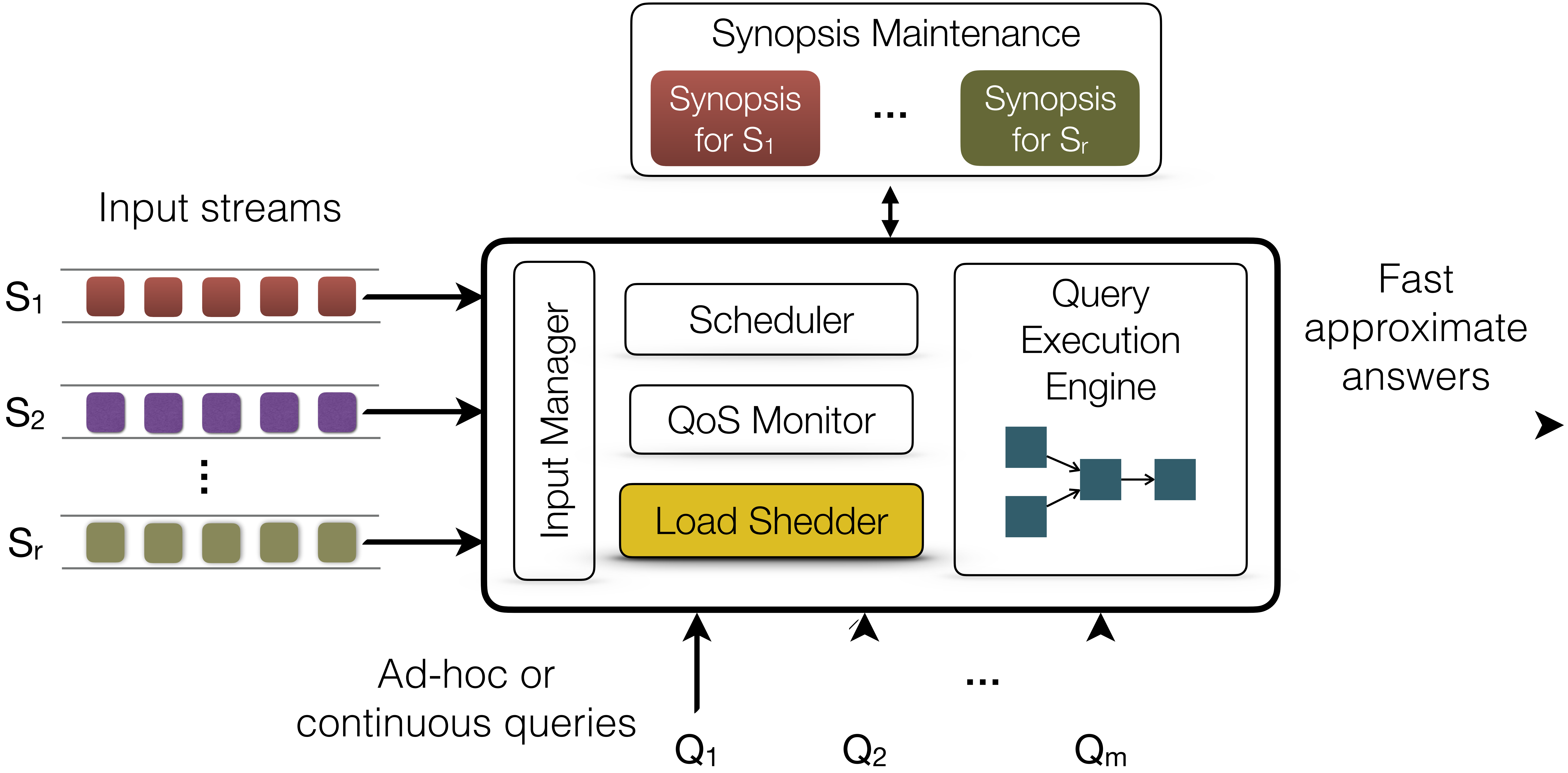
1. **Load shedding** - Selectively drop tuples
2. **Load-aware scheduling** - Adapt resource allocation
3. **Back-pressure** - Slow down the data flow
4. **Elasticity** - Scale the number of resources
5. Conclusion, vintage vs. modern

Selectively drop data

Temporarily trade-off accuracy for sustainable performance

- Load shedding is the process of **discarding data** when input rates increase beyond system capacity.
- The system detects an overload situation during runtime and selectively drops tuples according to a QoS specification.
- Similar to **congestion control** or video streaming in a lower quality.

DSMS with load shedder



Load shedding decisions

- **When** to shed load?
  - detect overload quickly to avoid latency increase
  - monitor input rates
- **Where** in the query plan?
  - dropping at the sources vs. dropping at bottleneck operators
- **How much** load to shed?
  - enough for the system to keep-up
- **Which** tuples to drop?
  - improve latency to an acceptable level
  - cause only minimal results quality degradation

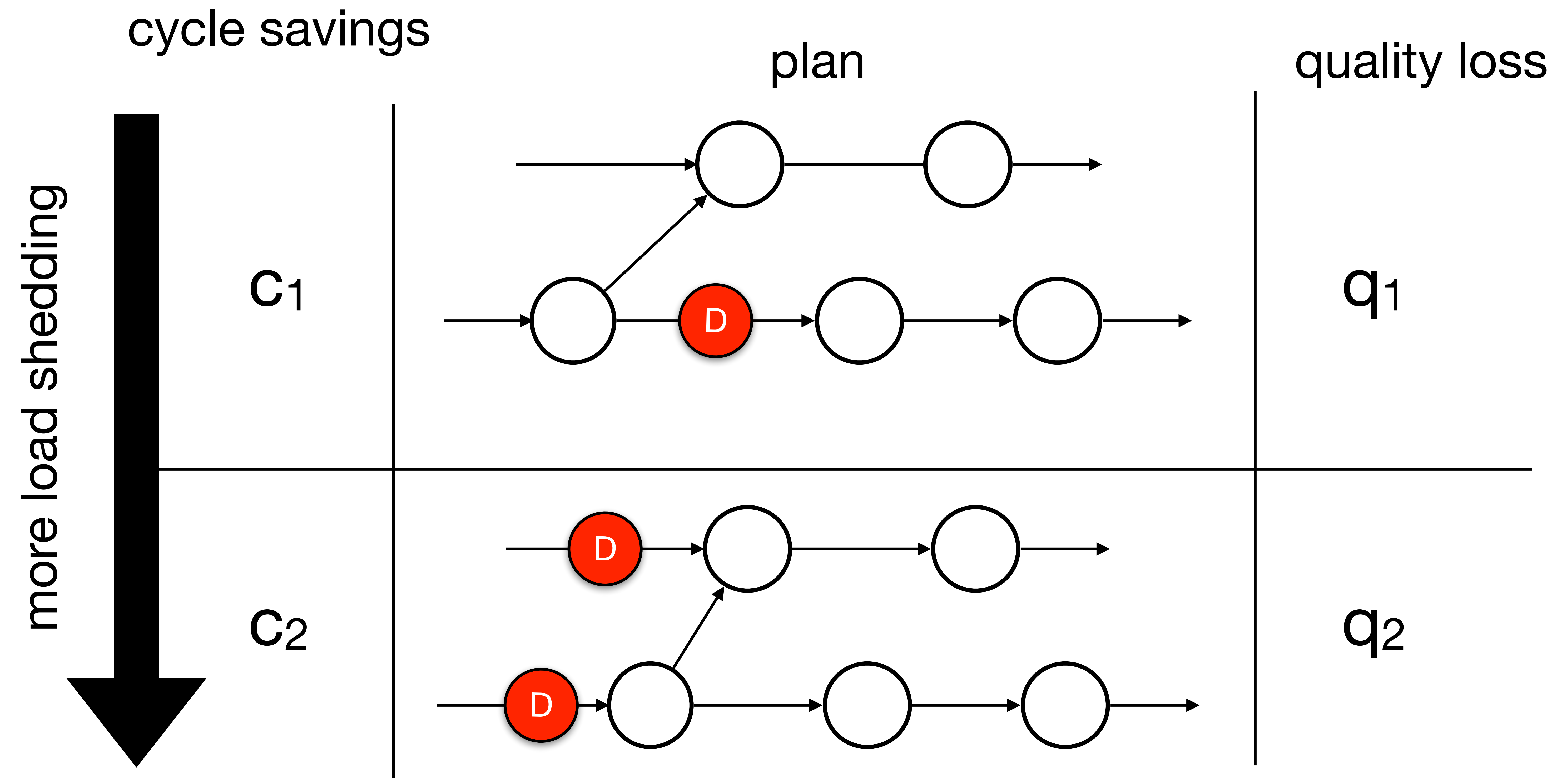
Detecting overload

When to shed load?

- An incorrectly triggered shedding action can cause unnecessary result degradation
- Load shedding components rely on **statistics** gathered during execution:
  - Monitor processing and input rates, estimate operator costs and selectivities.
- Almost perfect knowledge about an operator's behavior is required
  - Techniques are restricted to a predefined set of operators (filter, union, join, specific window types).
- Feedback control loop approaches have been proposed for operator-independent overload detection.

Load Shedding Road Map (LSRM)

Materialized ordered load shedding plans



- Load shedding can be implemented by placing special **drop operators** in the query plan.
- Dropping near the source avoids wasting work but it might affect results of multiple queries.

Which tuples to drop?

N. Tatbul (VLDB’06-’07), N. Katsipoulakis (IEEE Big Data’18)

Random: drop tuples at random

- Approximate query processing techniques can be used for known aggregations, where accuracy is measured in terms of relative error in the computed query answers.

Window-aware: drop entire windows instead of individual tuples

- Window integrity is preserved so that results under shedding are not approximations but a subset of the exact answers.

Concept-driven: drop by measuring tuple utility

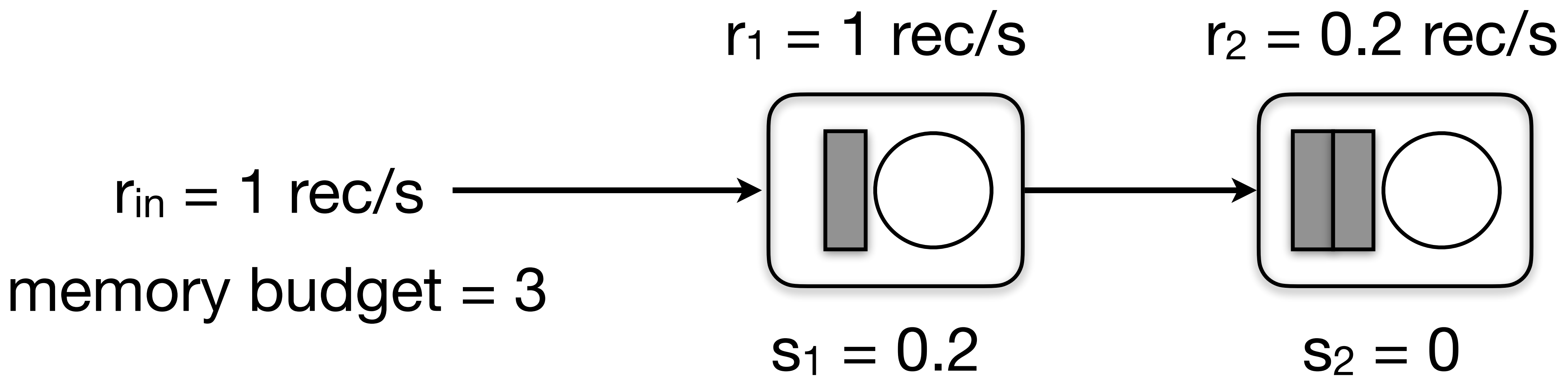
- Tuples are discarded by relying on the notion of a window-based concept drift which is computed as a similarity metric across consecutive windows.

How many tuples to drop?

- The amount of tuples to discard depends on the decisions of where and which tuples to shed.
- If input rates and processing capacity are known or easy to measure, estimates can be computed in a straight-forward manner.
- Estimations based on static operator selectivities and heuristics are unsuitable for frequent load fluctuations.
- Naive approaches can lead to system instability or unnecessary load shedding.

Load-aware scheduling

Minimize the backlog during bursts

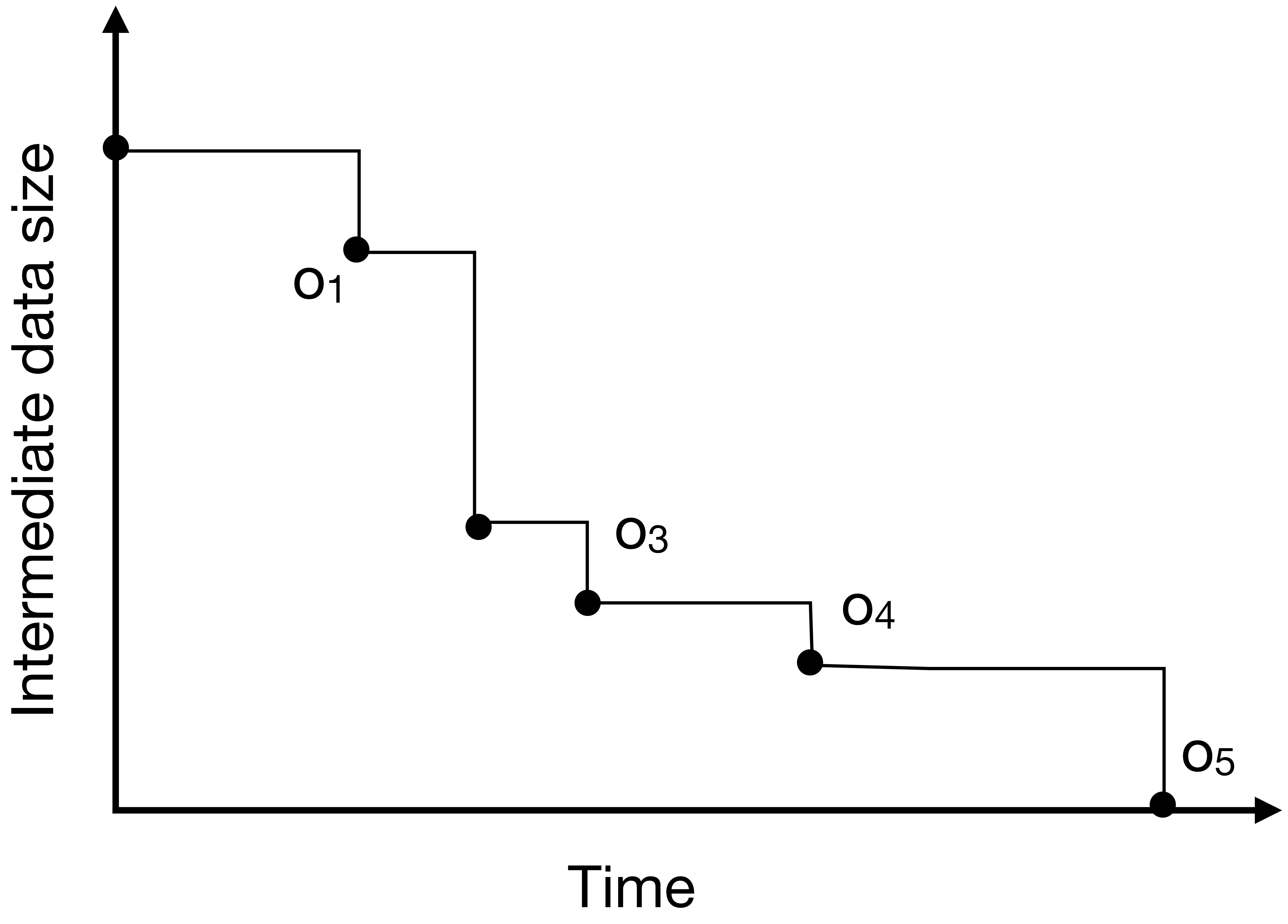


Naive FIFO scheduling will exceed the memory budget after 6s.

- Load-aware scheduling decides the operator order execution with the objective to minimize memory requirements during temporary load bursts.
- We assume that the arrival rate is within computational limits: the system will eventually be able to process the backlog.

Schedule highly selective operators first

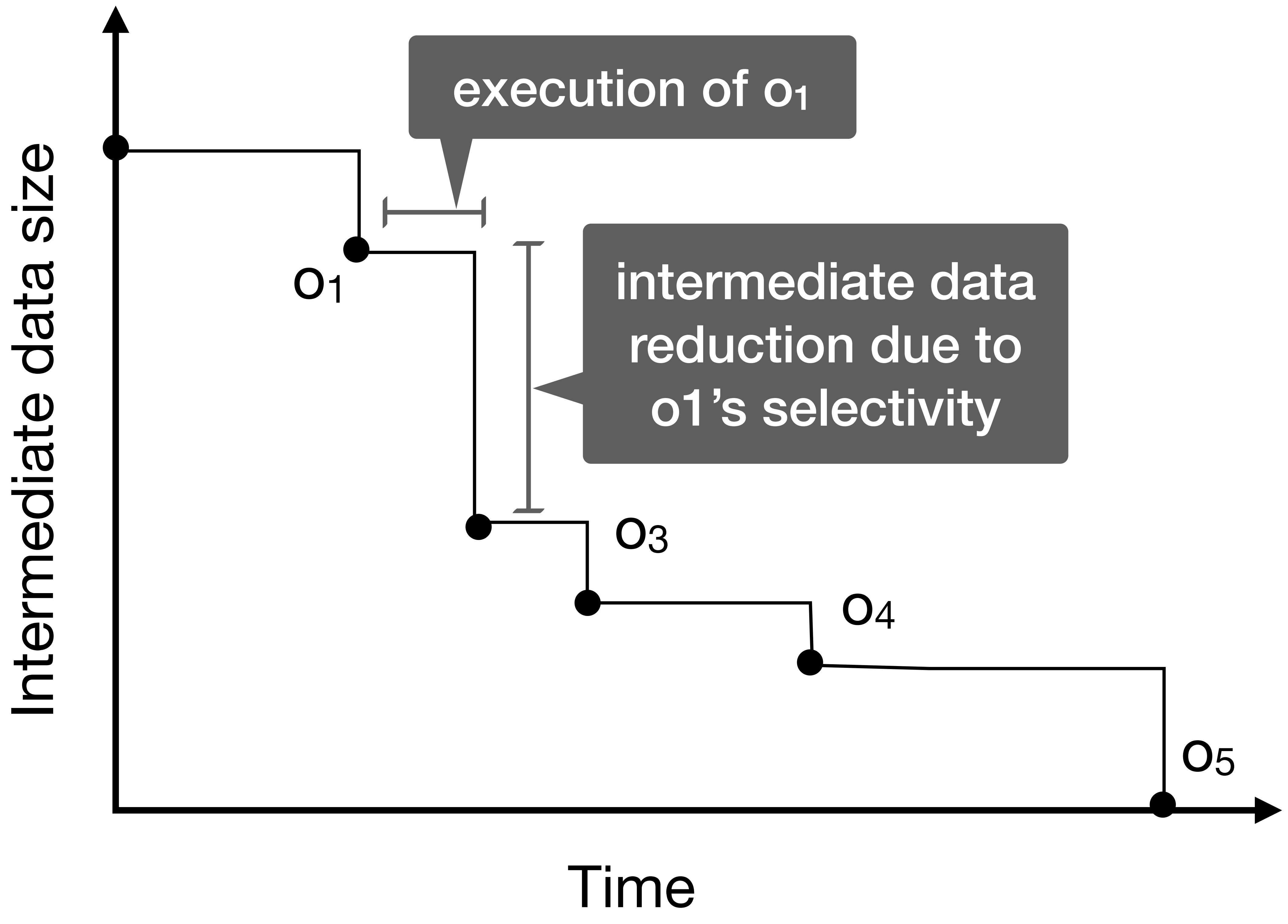
Chain scheduling (SIGMOD’03)



- The system simulates the **progress** of tuples along the dataflow.
- It assign **priorities** to operator chains based on the fraction of tuples they eliminate per unit of time and their position in the operator path.

Schedule highly selective operators first

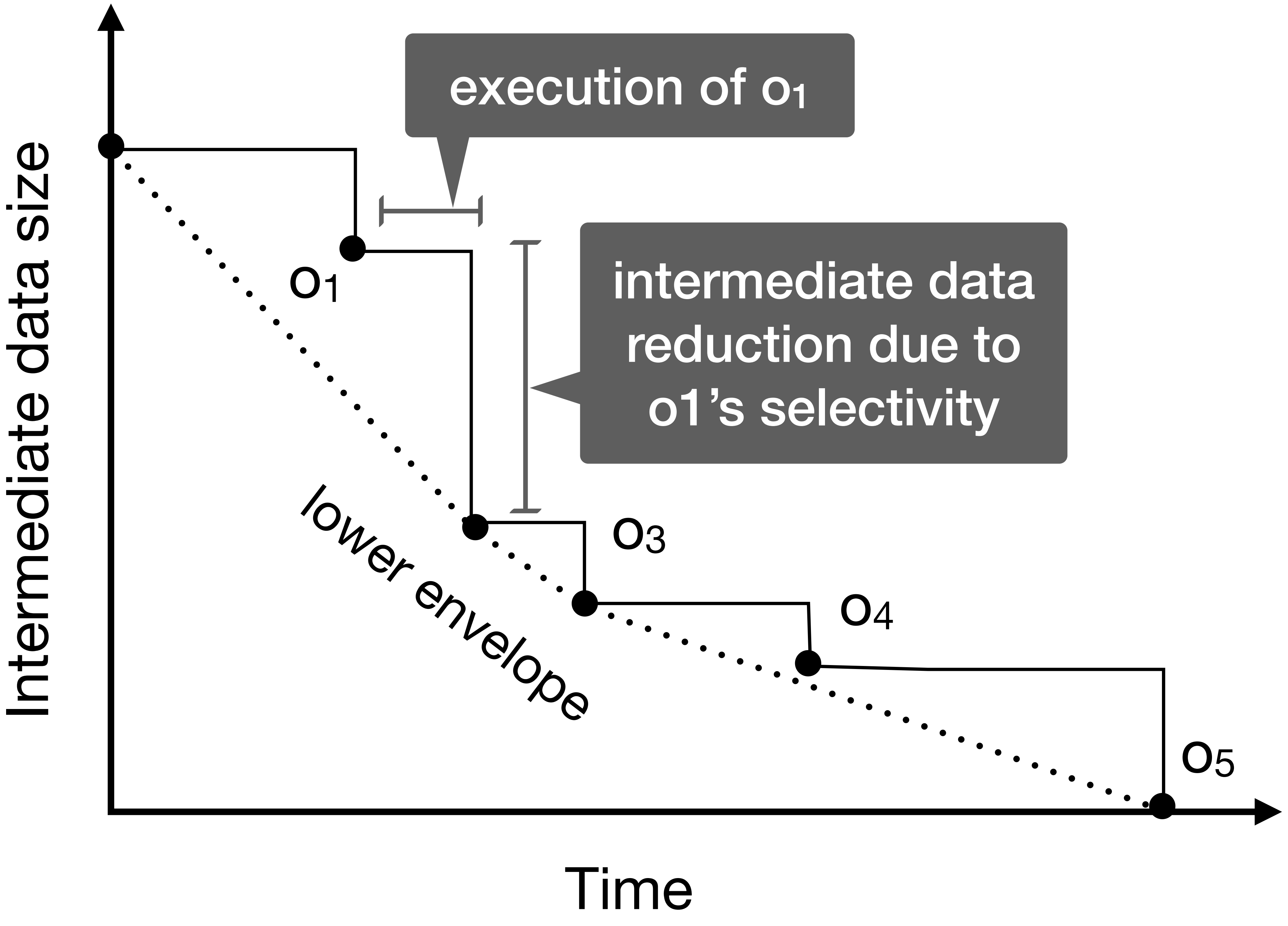
Chain scheduling (SIGMOD'03)



- The system simulates the **progress** of tuples along the dataflow.
- It assign **priorities** to operator chains based on the fraction of tuples they eliminate per unit of time and their position in the operator path.

Schedule highly selective operators first

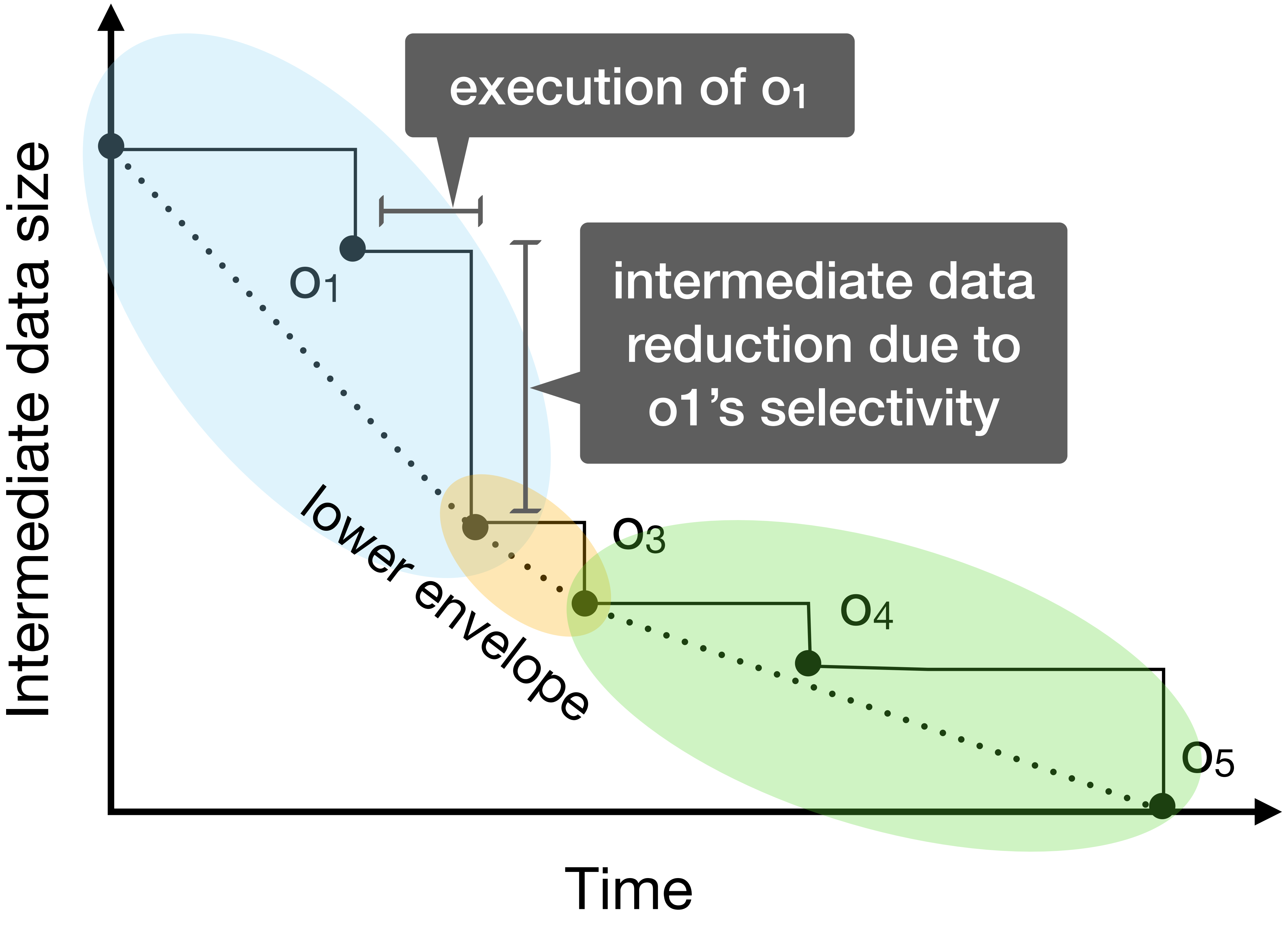
Chain scheduling (SIGMOD’03)



- The system simulates the **progress** of tuples along the dataflow.
- It assign **priorities** to operator chains based on the fraction of tuples they eliminate per unit of time and their position in the operator path.

Schedule highly selective operators first

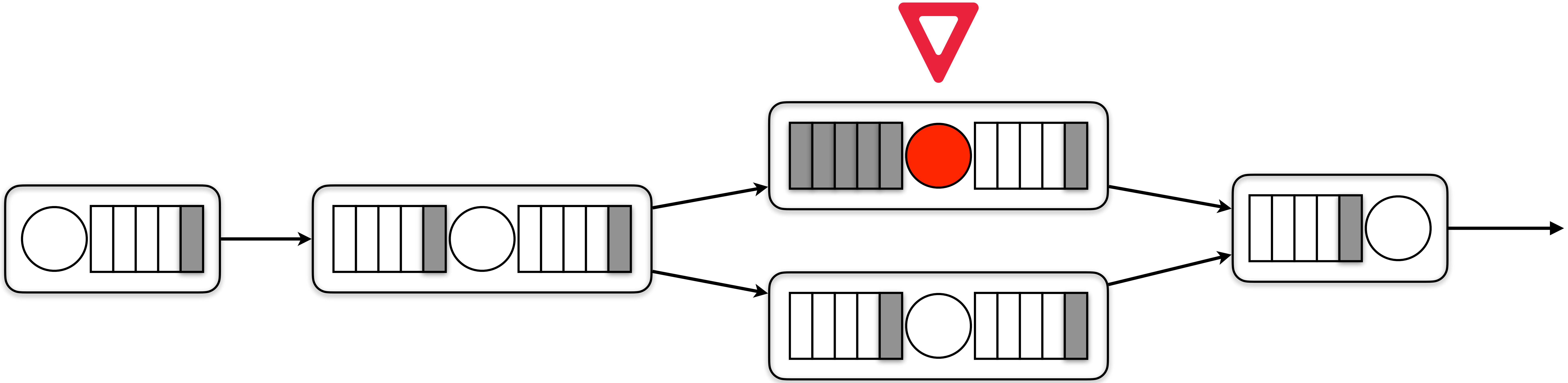
Chain scheduling (SIGMOD'03)



- The system simulates the **progress** of tuples along the dataflow.
- It assign **priorities** to operator chains based on the fraction of tuples they eliminate per unit of time and their position in the operator path.

Control rate through buffer availability

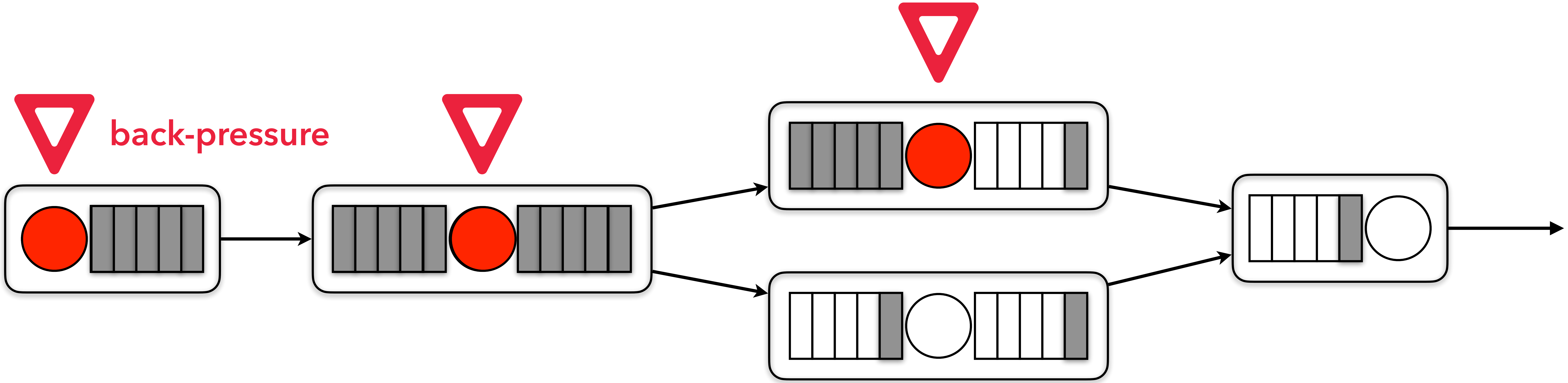
Back-pressure propagates to the sources



- All operators **slow down** to match the processing speed of **the slowest consumer**.
- To ensure no data loss, a **persistent input queue** (e.g. Kafka) and enough storage is required.

Control rate through buffer availability

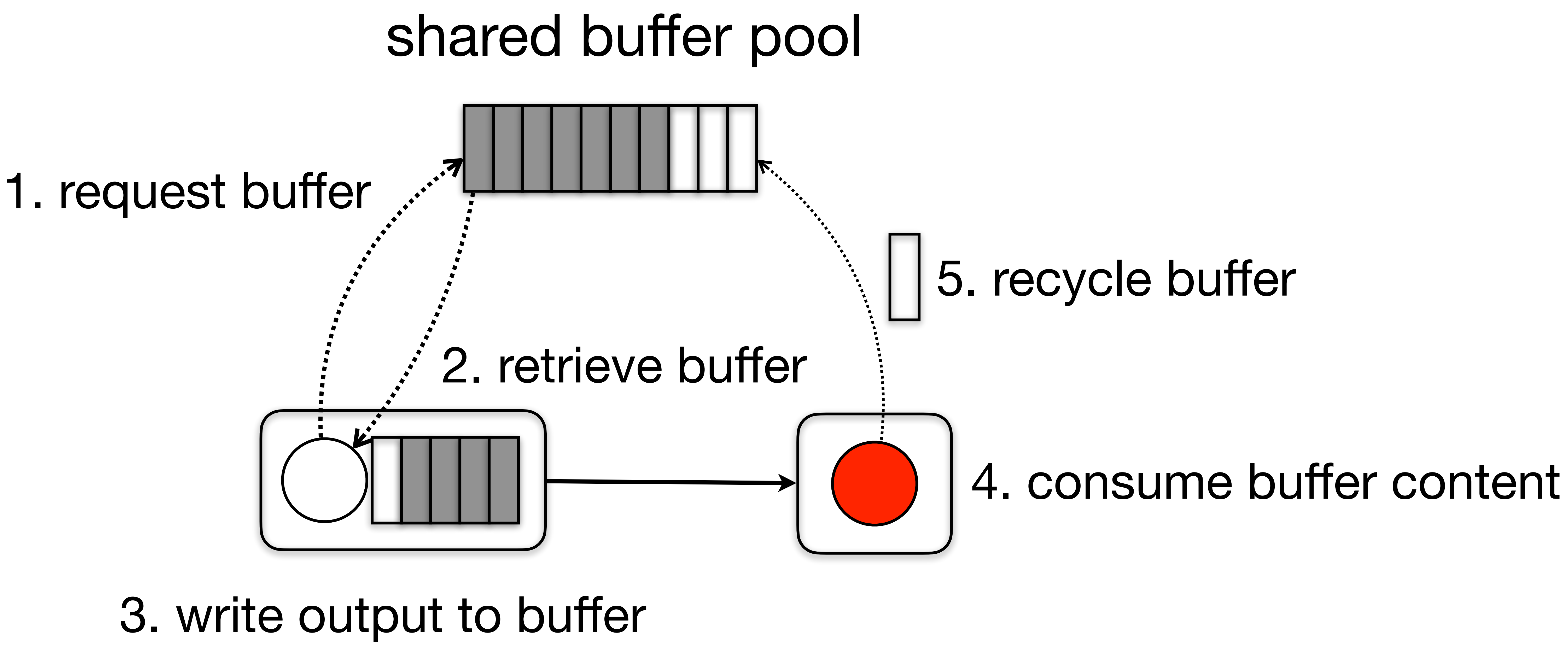
Back-pressure propagates to the sources



- All operators **slow down** to match the processing speed of **the slowest consumer**.
- To ensure no data loss, a **persistent input queue** (e.g. Kafka) and enough storage is required.

Local exchange

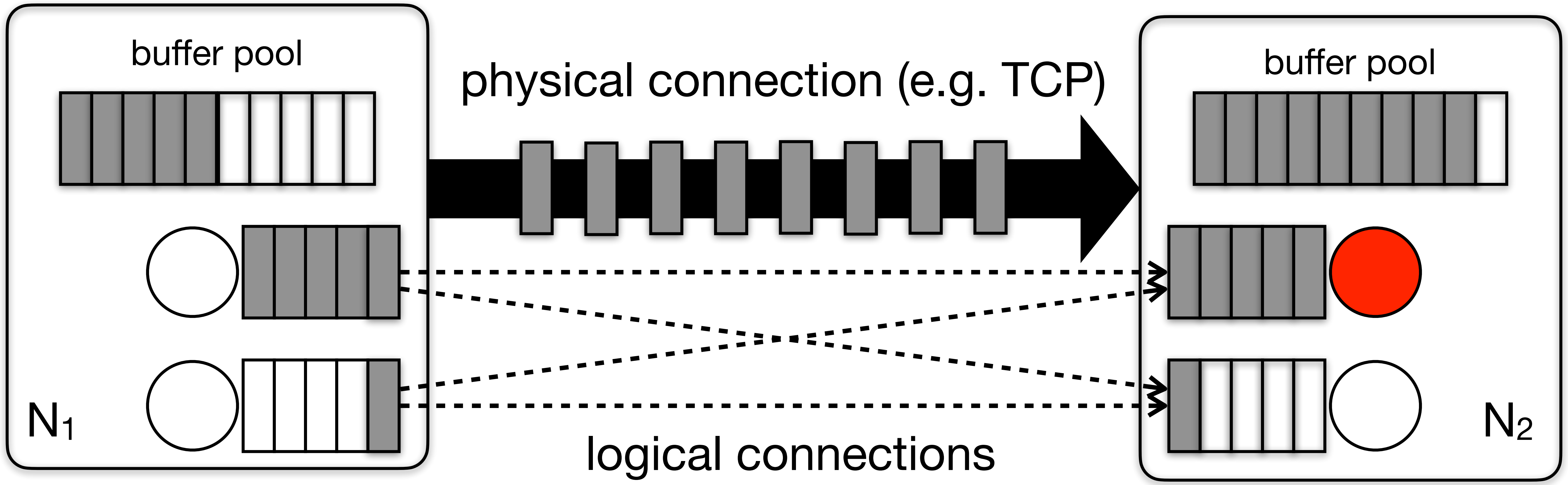
The producer and consumer run on the same machine



The producer slows down according to the rate the consumer recycles buffers.

Remote exchange

The producer and consumer run on different machines

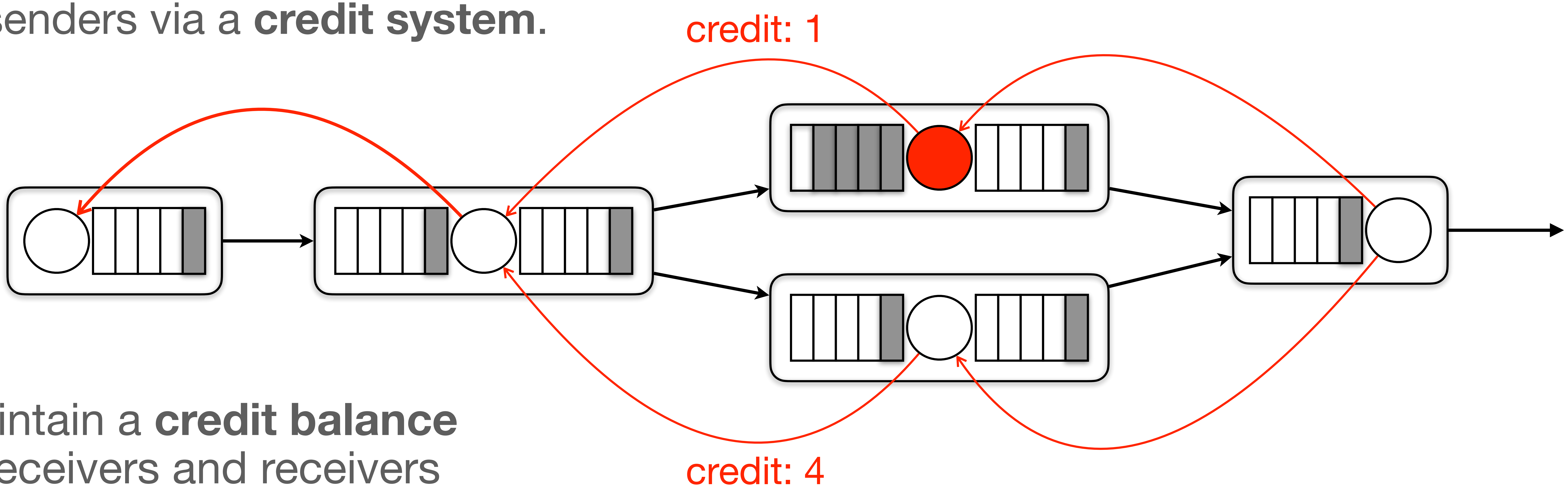


- If there is no buffer on the consumer side, reading from the TCP connection is interrupted.
- The producer is slowed down if it cannot put new data on the wire.

Credit-based flow control

Link-by-link congestion control

Buffer space availability is signaled from receivers to senders via a **credit system**.



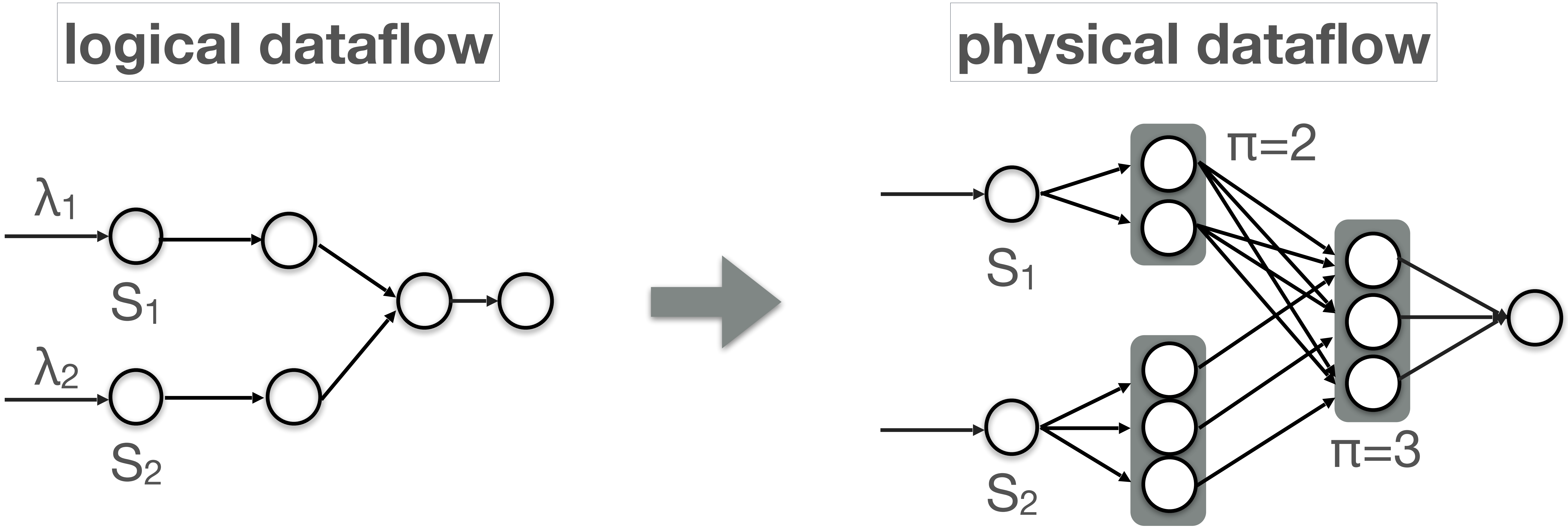
Senders maintain a **credit balance** for all their receivers and receivers regularly send notifications upstream containing their available credits.

Buffer-based vs. CFC

- CFC inflicts back-pressure on pairs of communicating tasks only.
- In the presence of bursty traffic, CFC causes back-pressure to build up fast and propagate along congested VCs to their sources which can be throttled.
- In the presence of skew, CFC avoids blocking the flow of data to downstream operators due to a single overloaded task.
- On the downside, the additional credit announcement messages might increase end-to-end latency.

Elastic streaming systems

Adaptive resource scaling according to the workload



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

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

Scaling Controller

Detect symptoms	*Decide whether to scale*	*Decide how much to scale*
metrics	policy	scaling action
Service time, waiting time, CPU utilization, congestion, back-pressure, throughput	**Predictive**: Queuing theory, control theory, analytical dataflow-based models **Heuristic**: Rule-based models, e.g. if CPU utilization > 70% => scale out	**Dataflow-wide**: at-once for all operators **Speculative**: small changes at one operator at a time

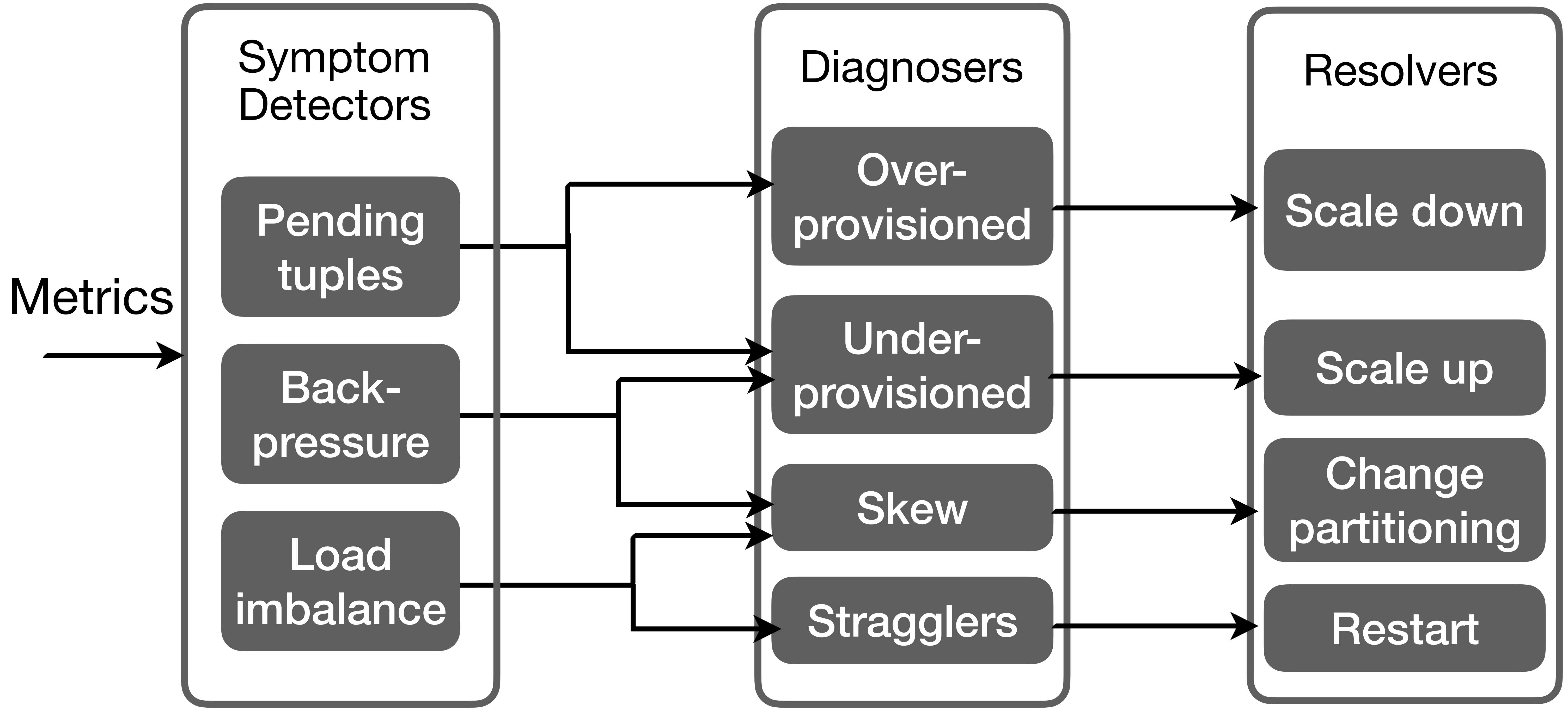
Accuracy: no over/under-provisioning

Stability: no oscillations

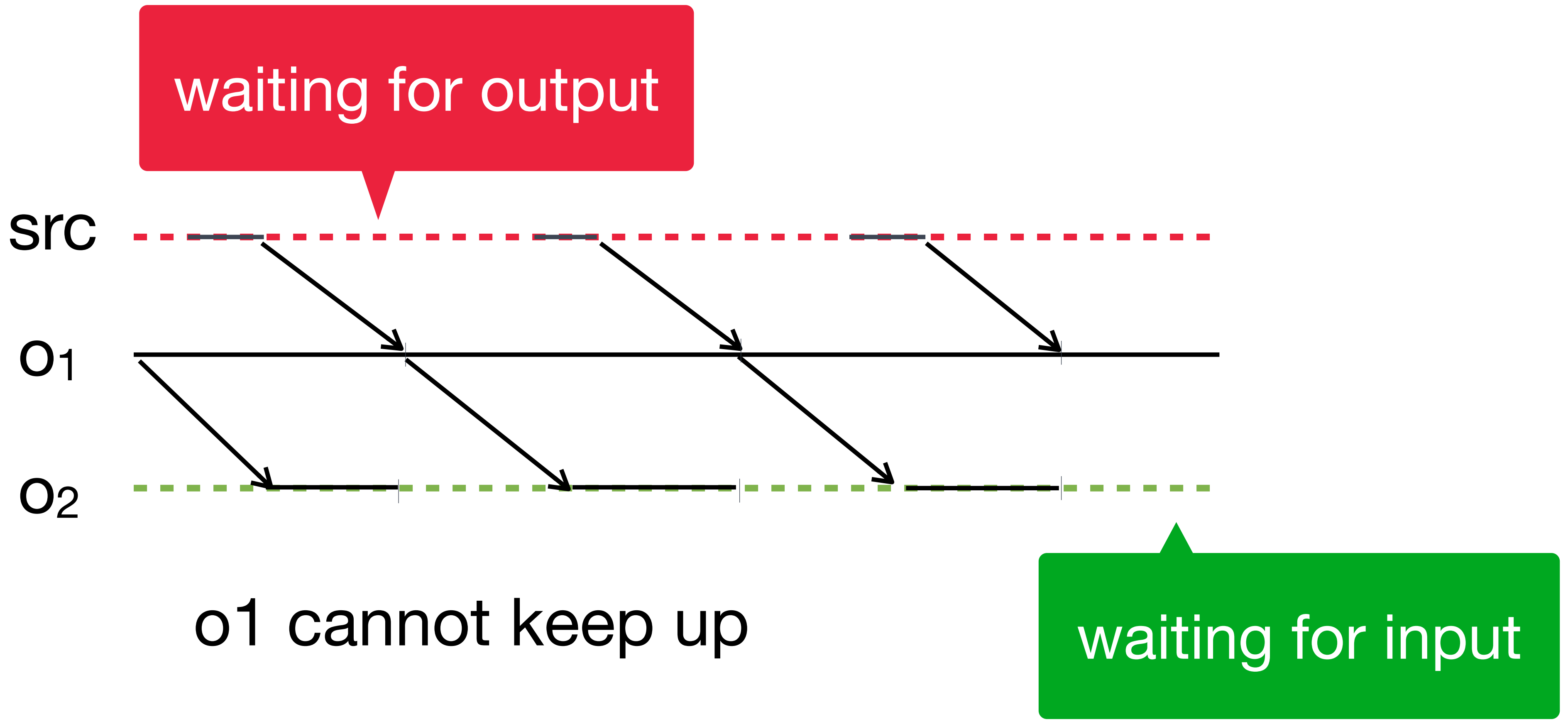
Performance: fast convergence

Heuristic Policy

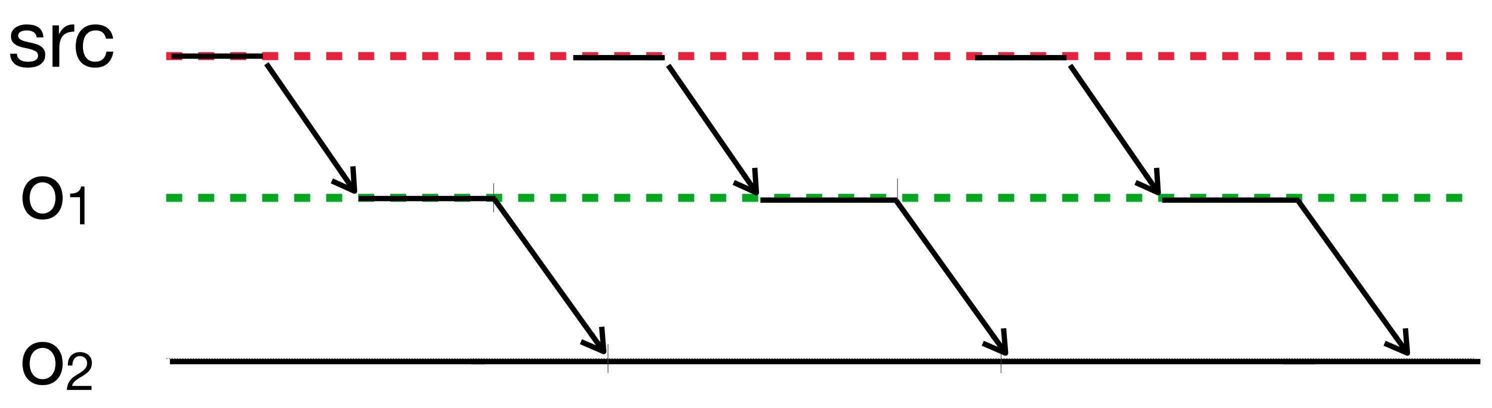
Dhalion (VLDB'17)



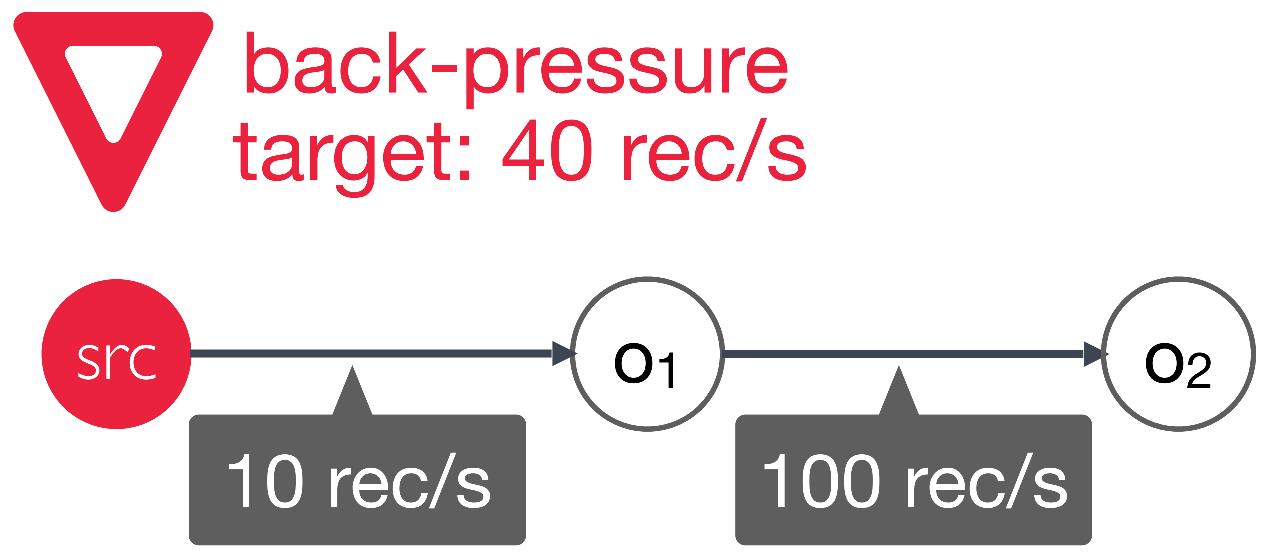
- An **action log** records policies and associated diagnoses
- A **blocklist** records actions that did not produce the expected outcome



o1 cannot keep up



o2 cannot keep up



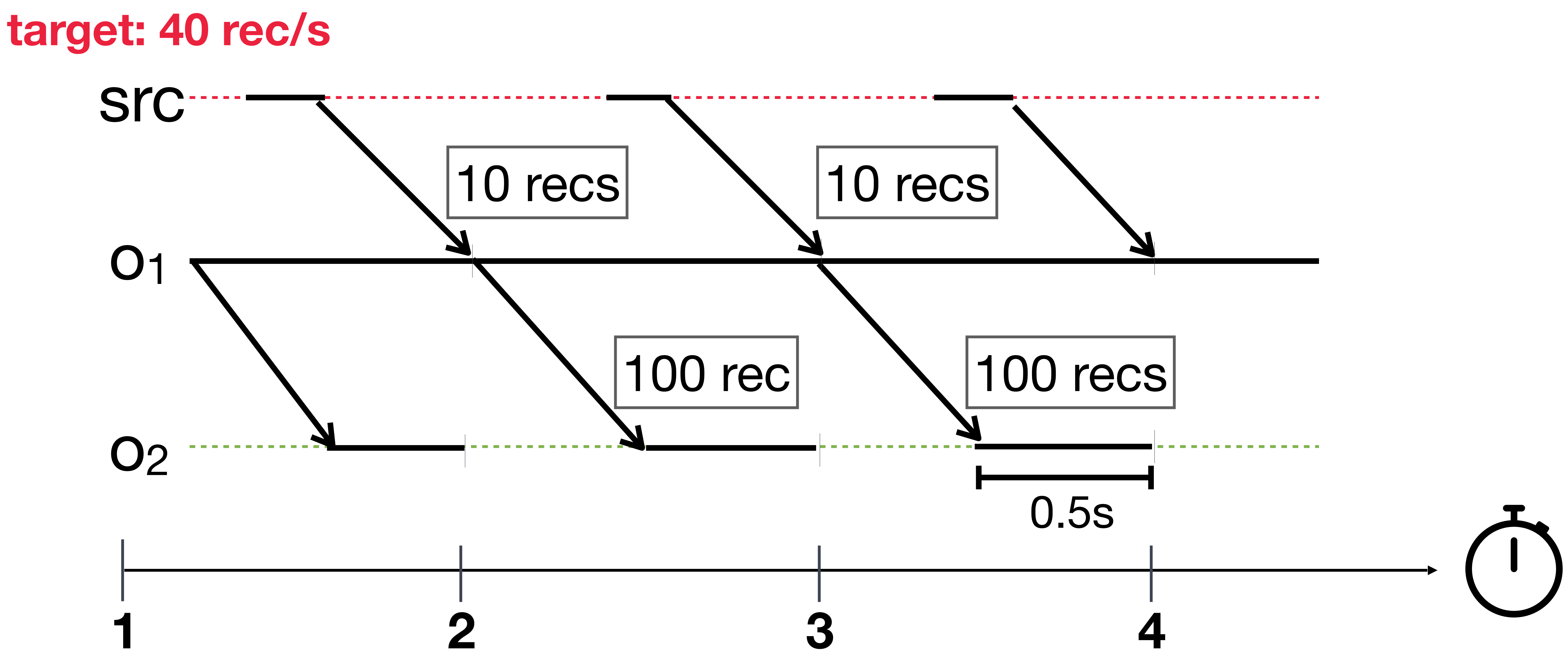
Which operator is the bottleneck?

What if we scale o_1 x 4?

How much to scale o_2?

Predictive Policy

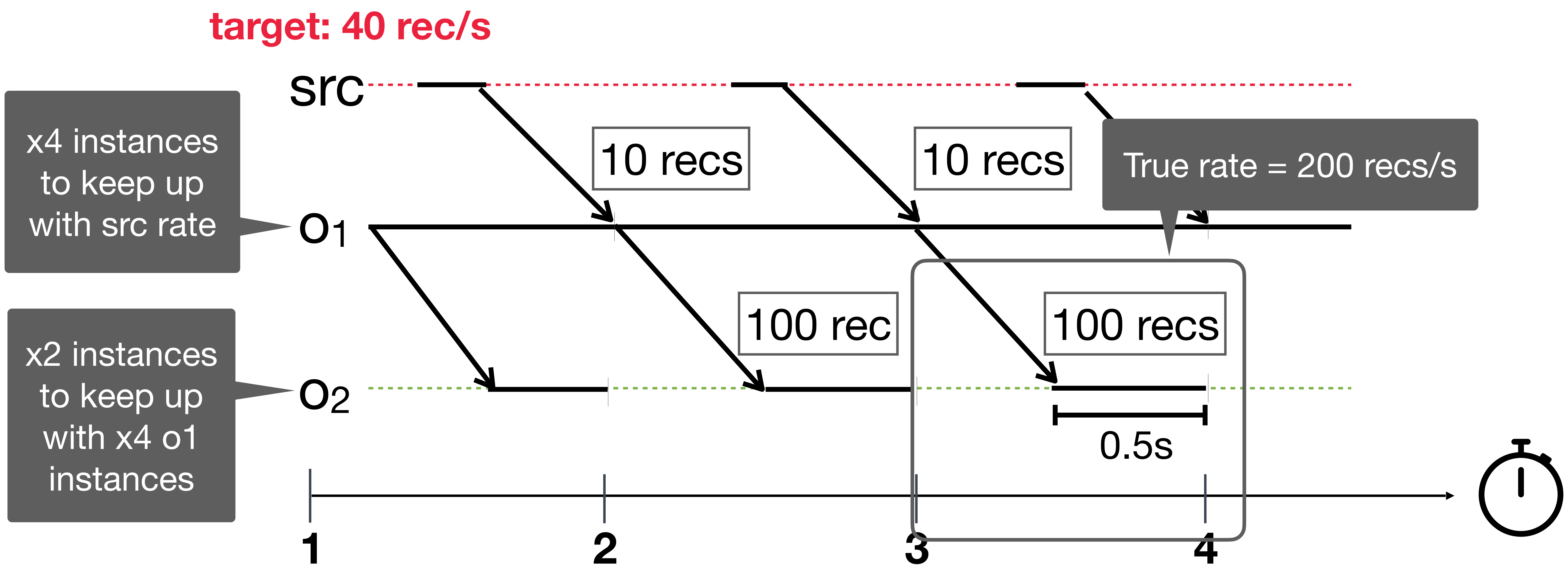
Three steps is all you need - DS2 (OSDI'18)



- 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.

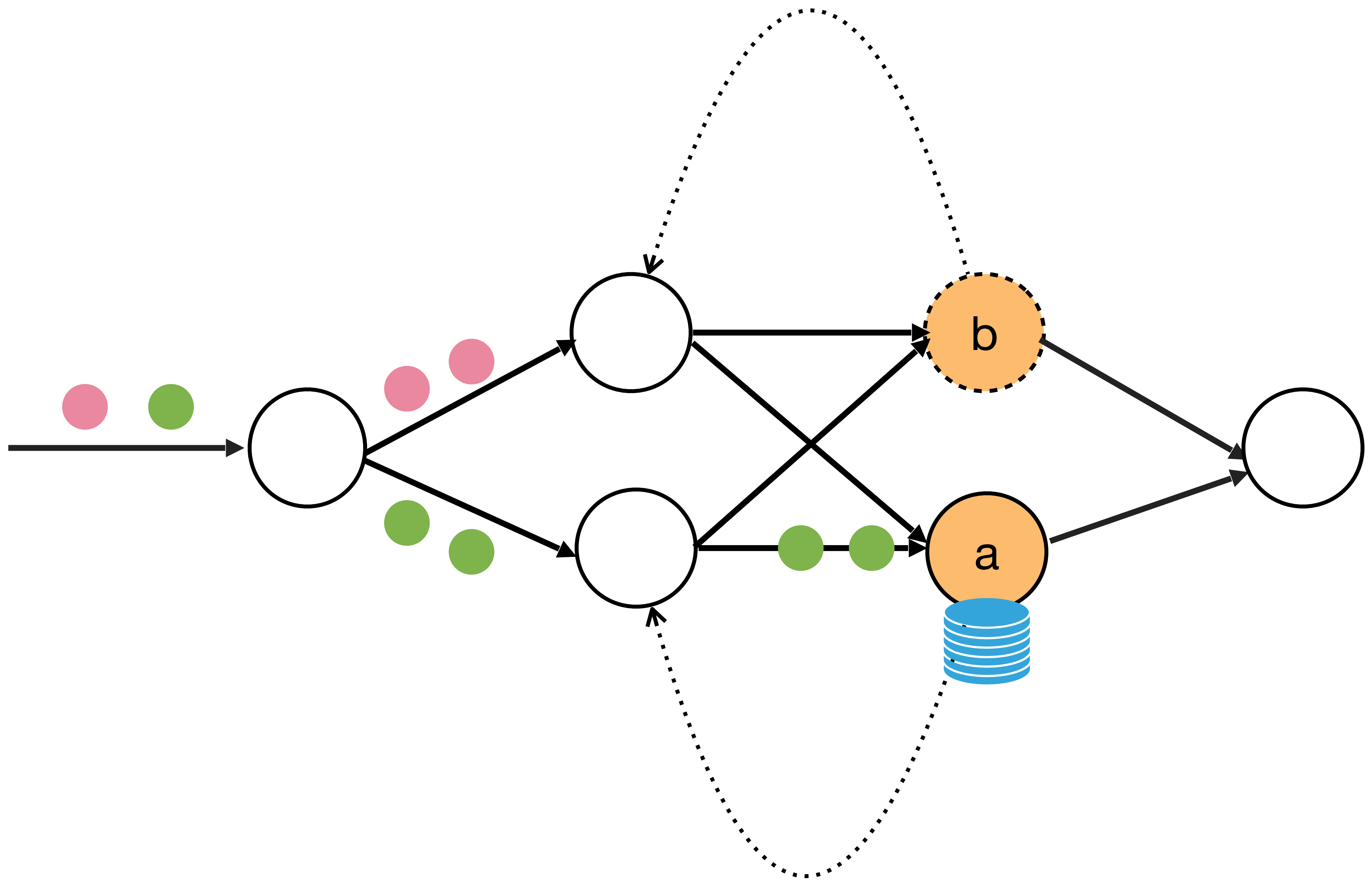
Elasticity mechanisms

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)

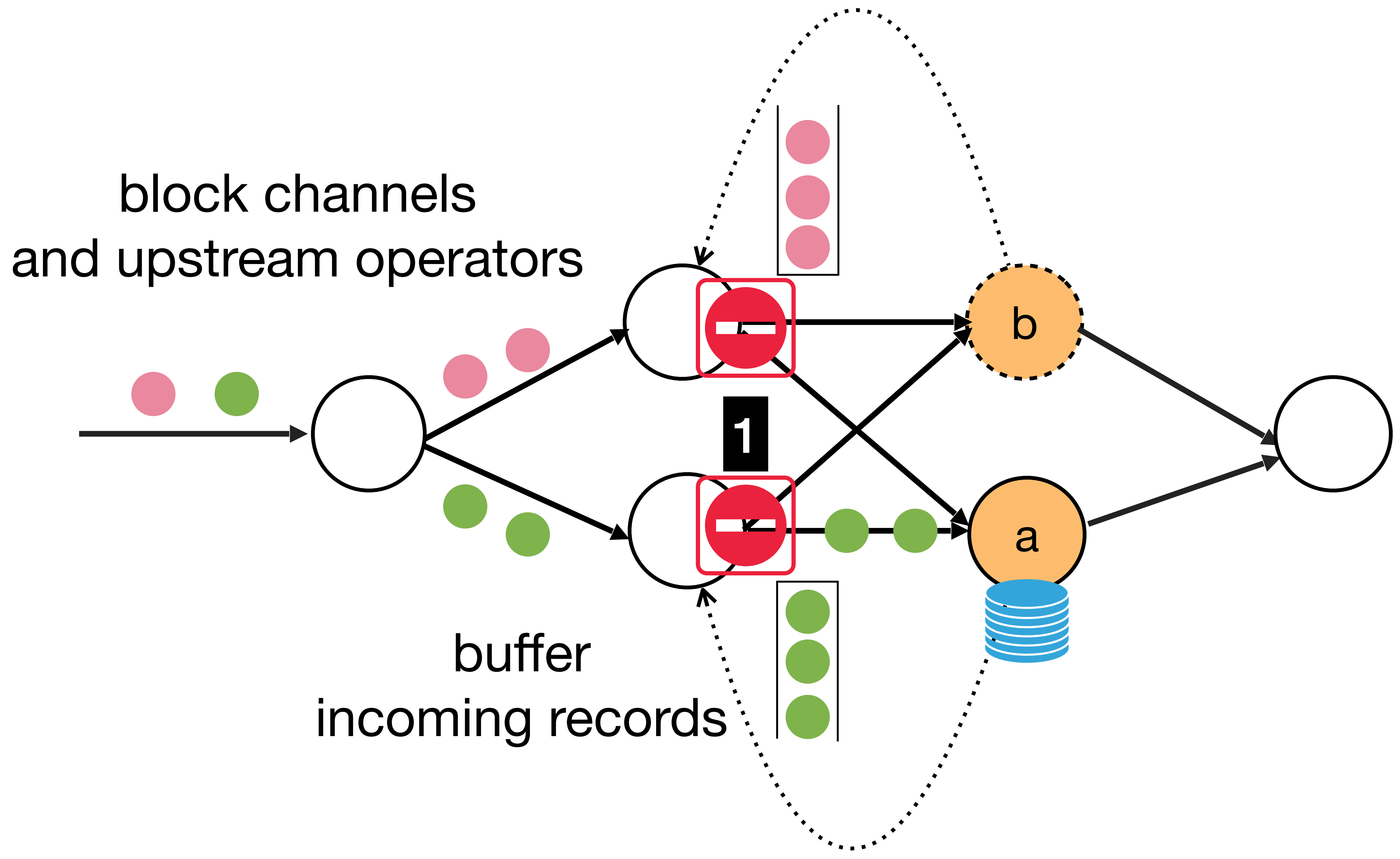


Migrating state from a to b

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)

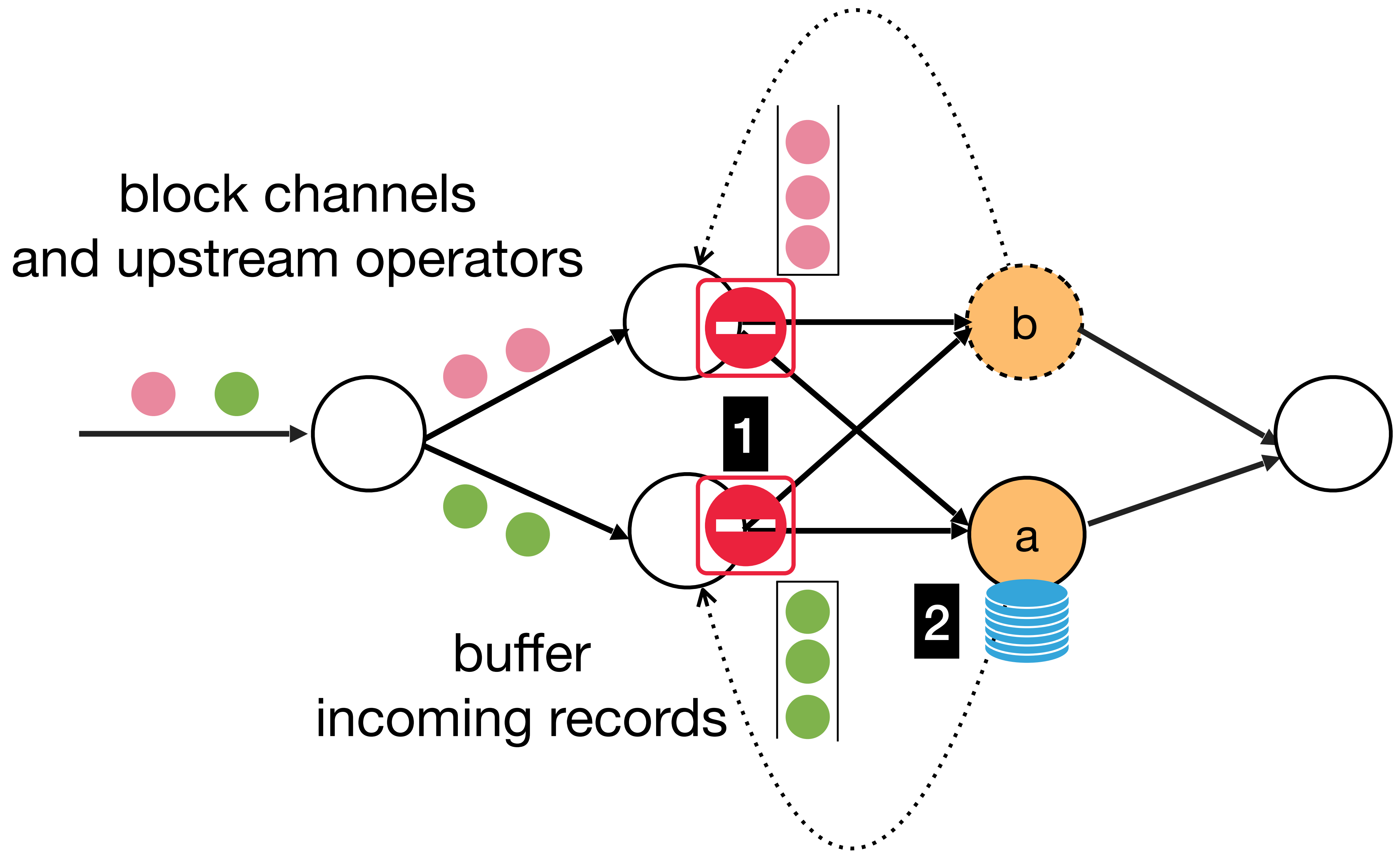


Migrating state from a to b

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)

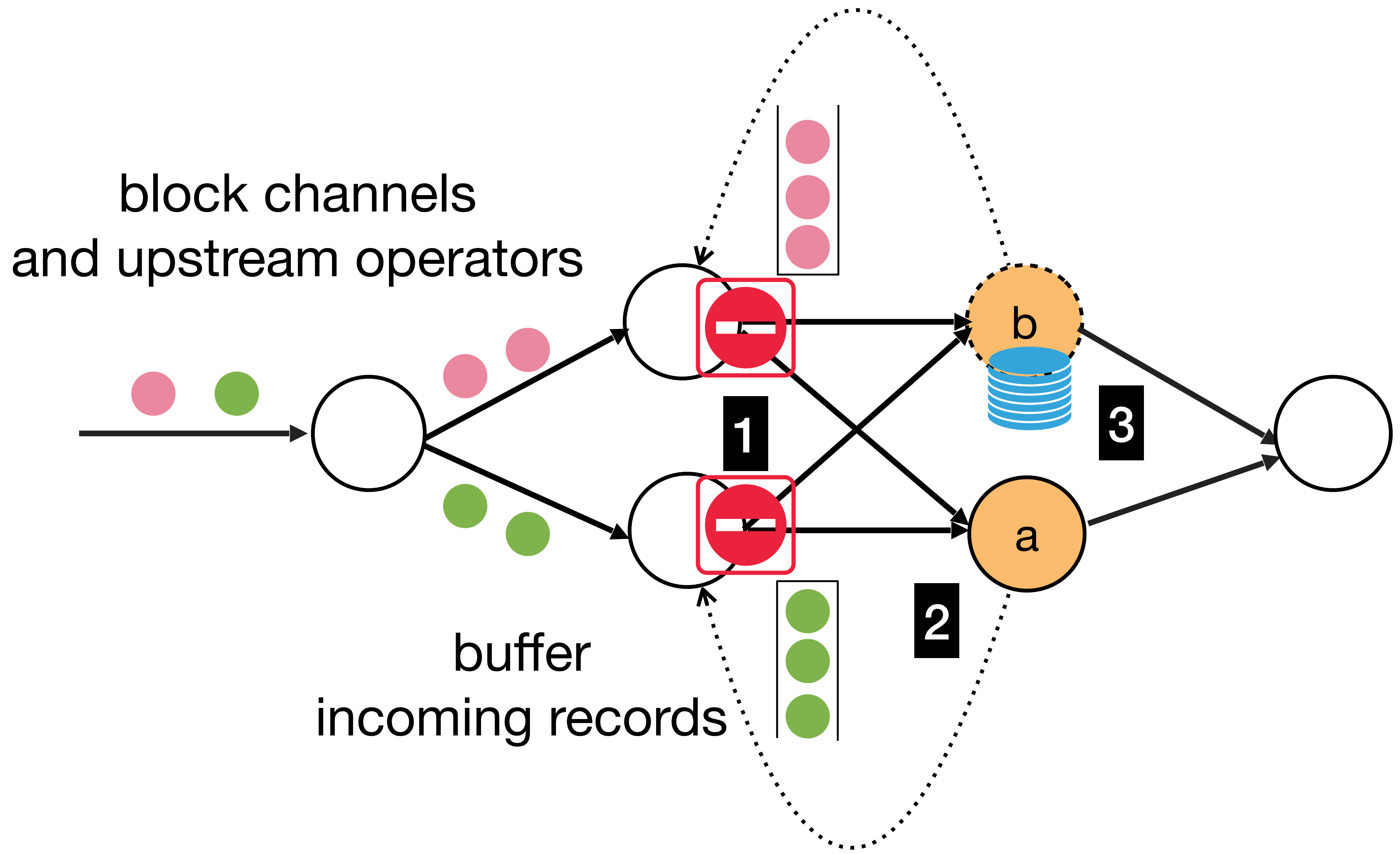


Migrating state from a to b

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)

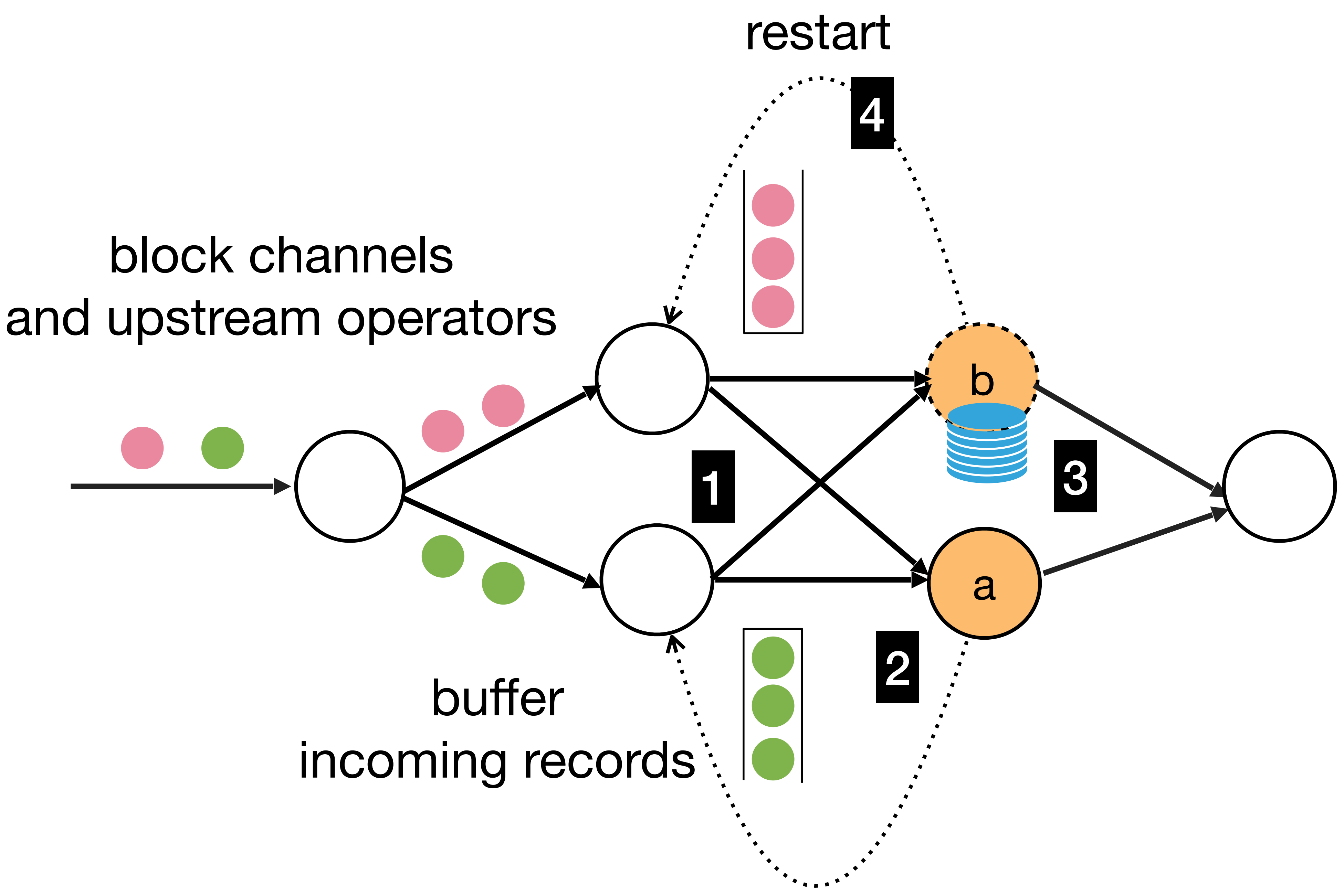


Migrating state from a to b

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)

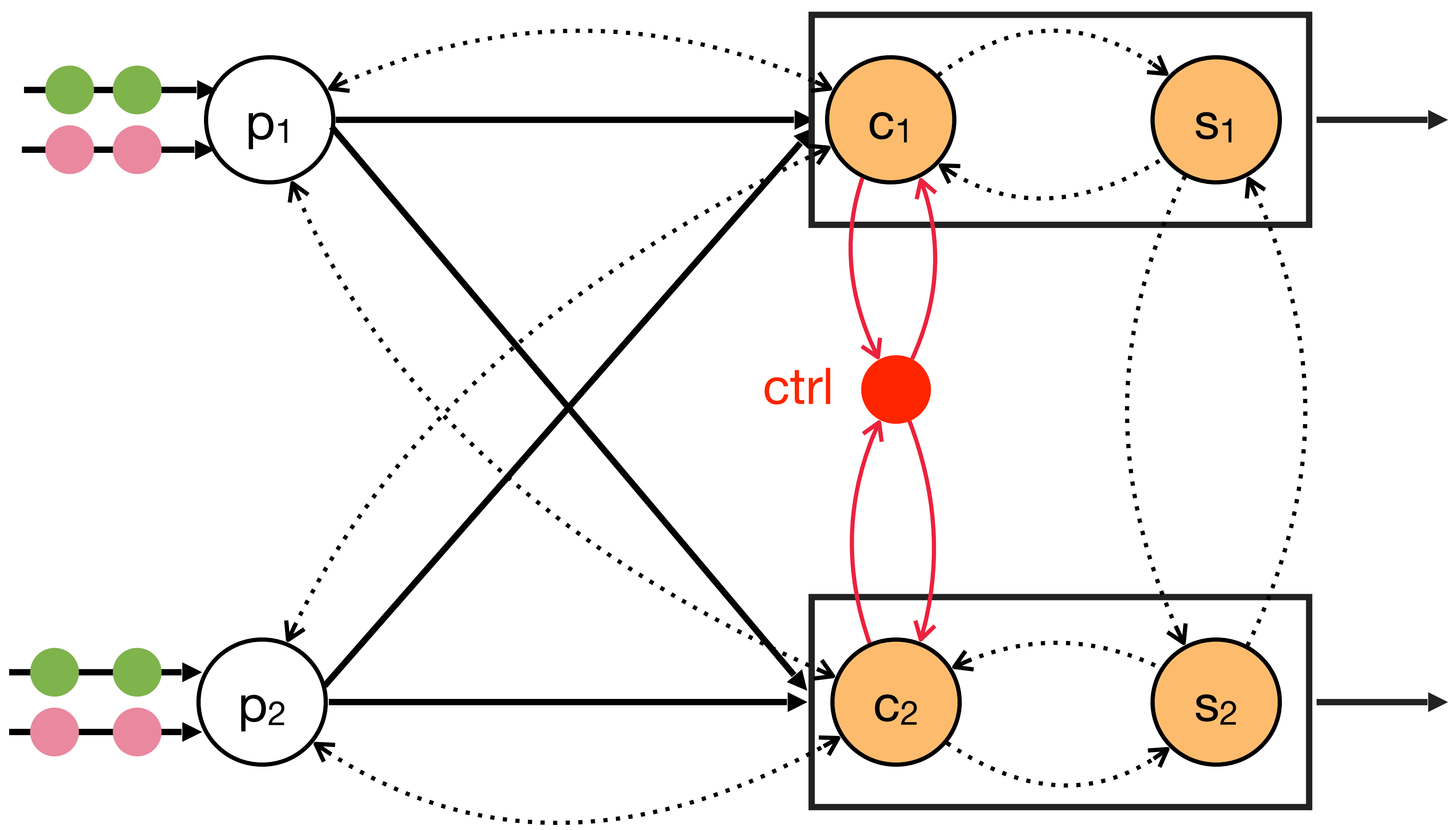


Migrating state from a to b

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)

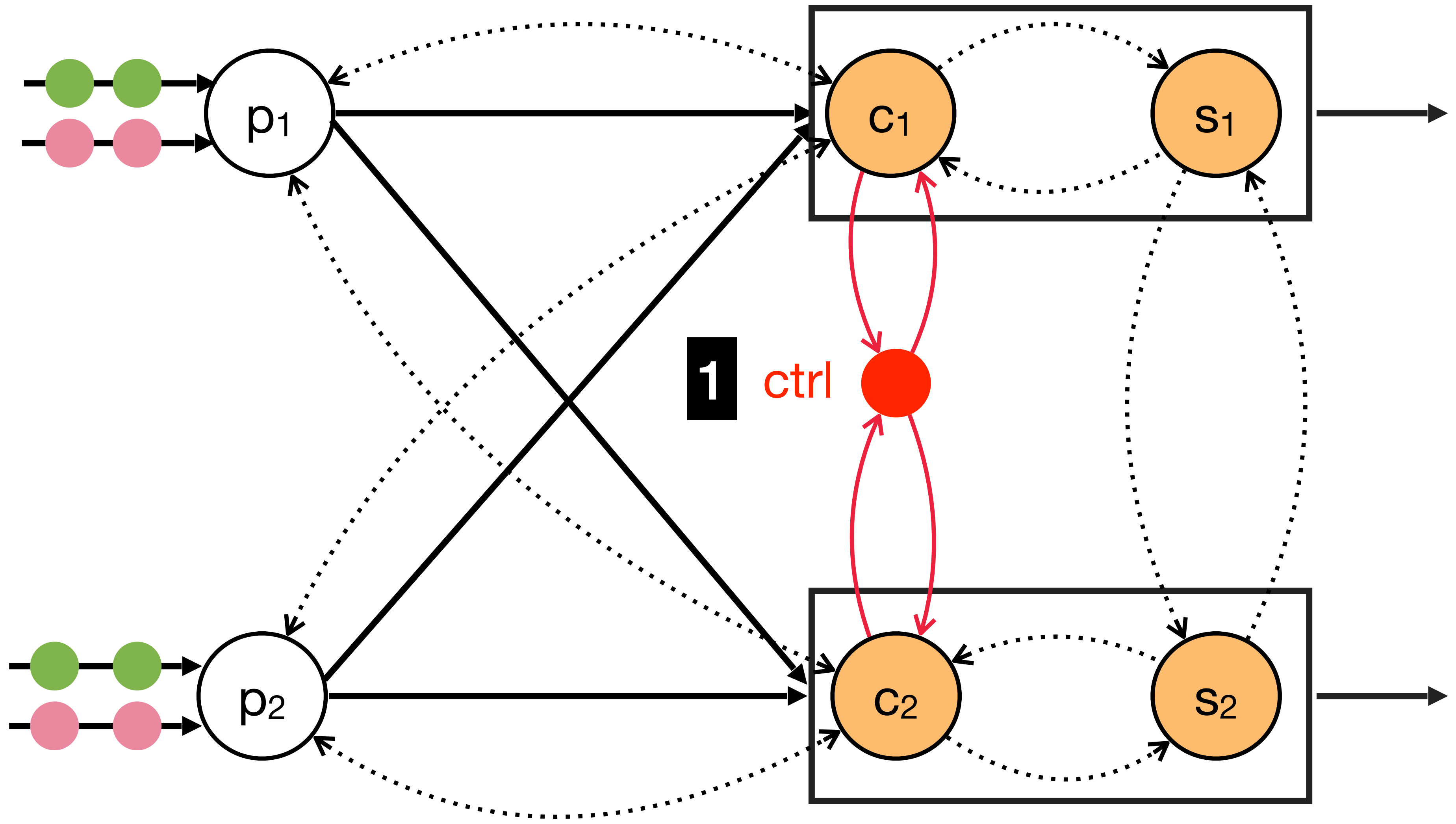


Migrating state from c_1 to c_2

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)

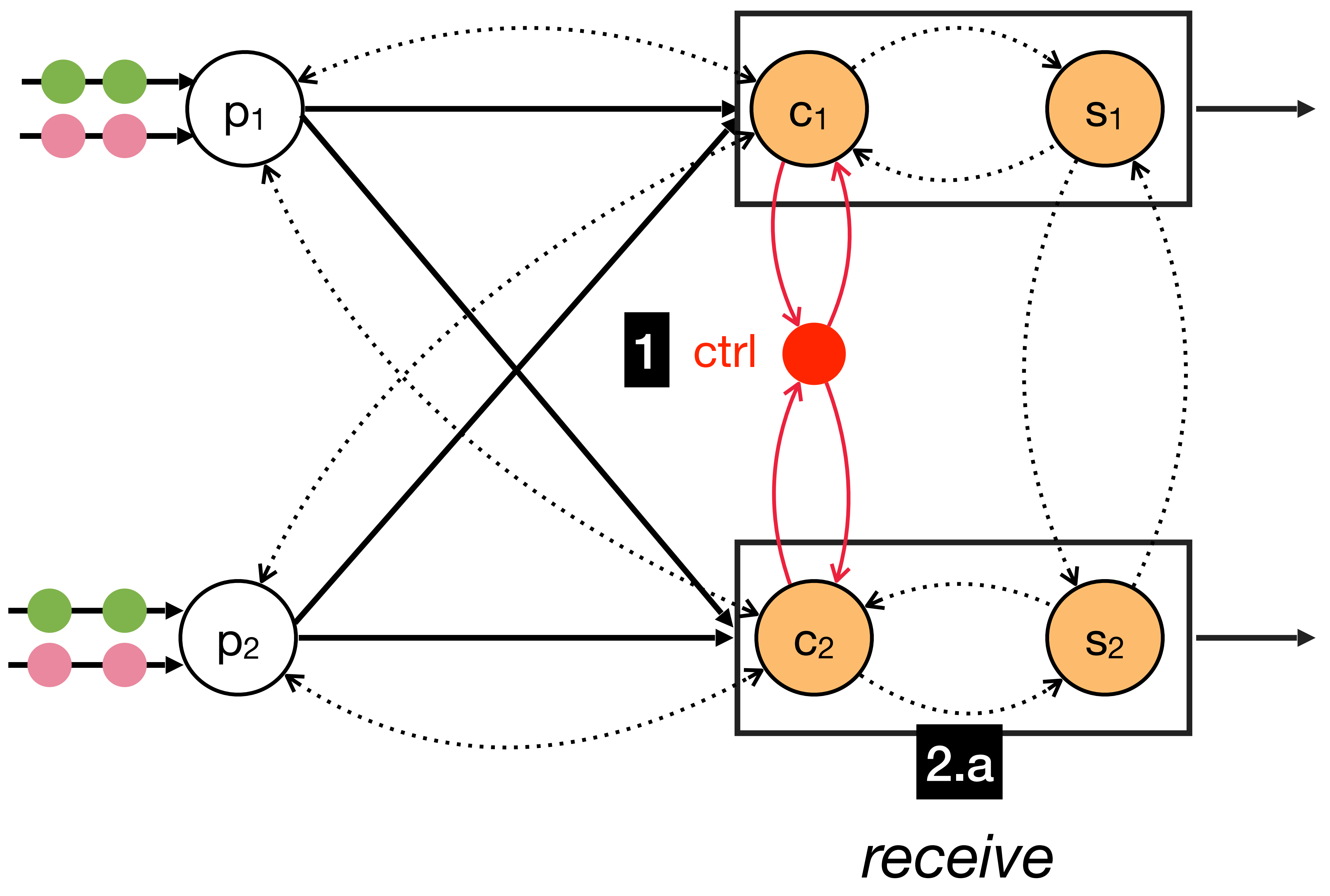


Migrating state from c_1 to c_2

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)

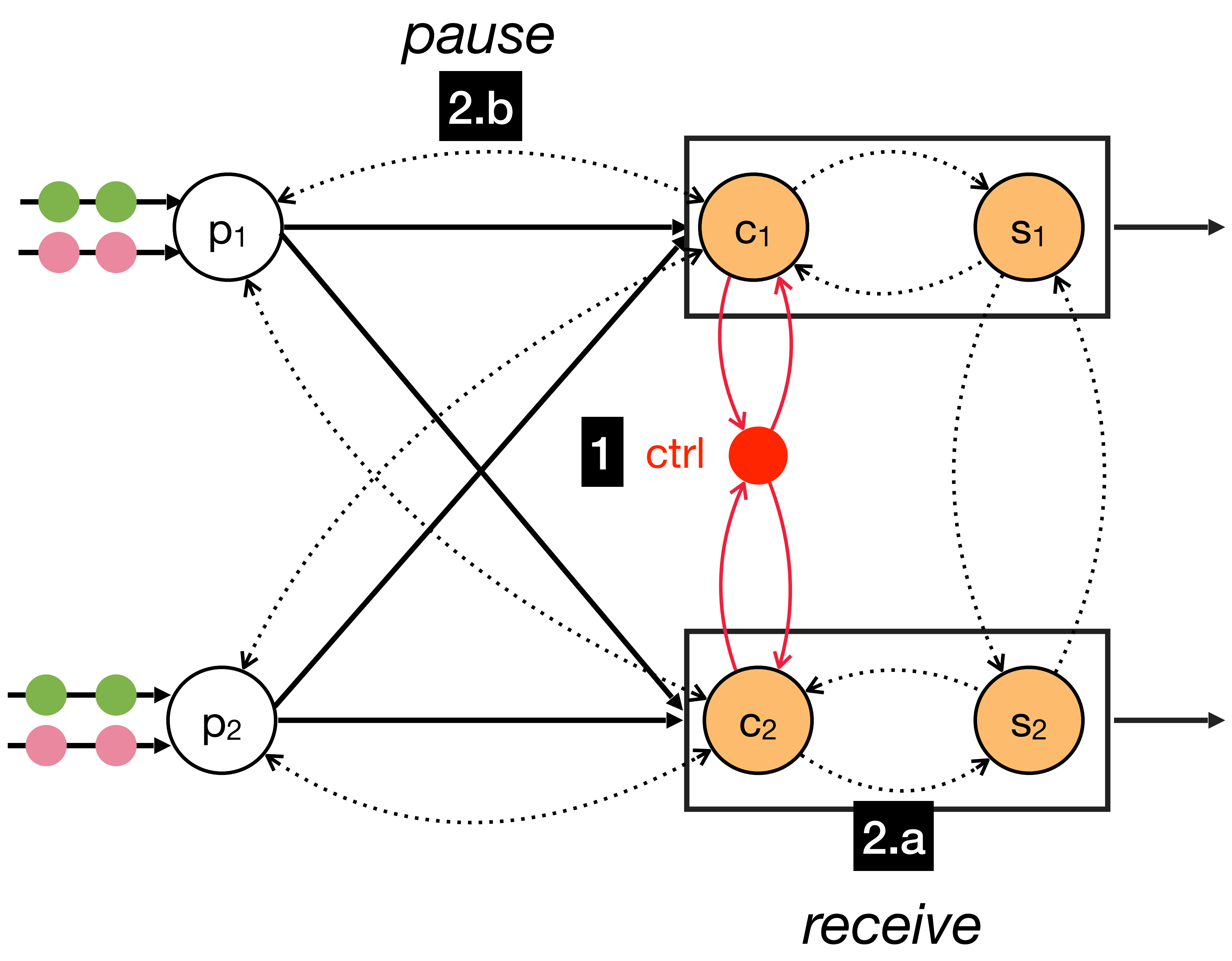


Migrating state from c_1 to c_2

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)

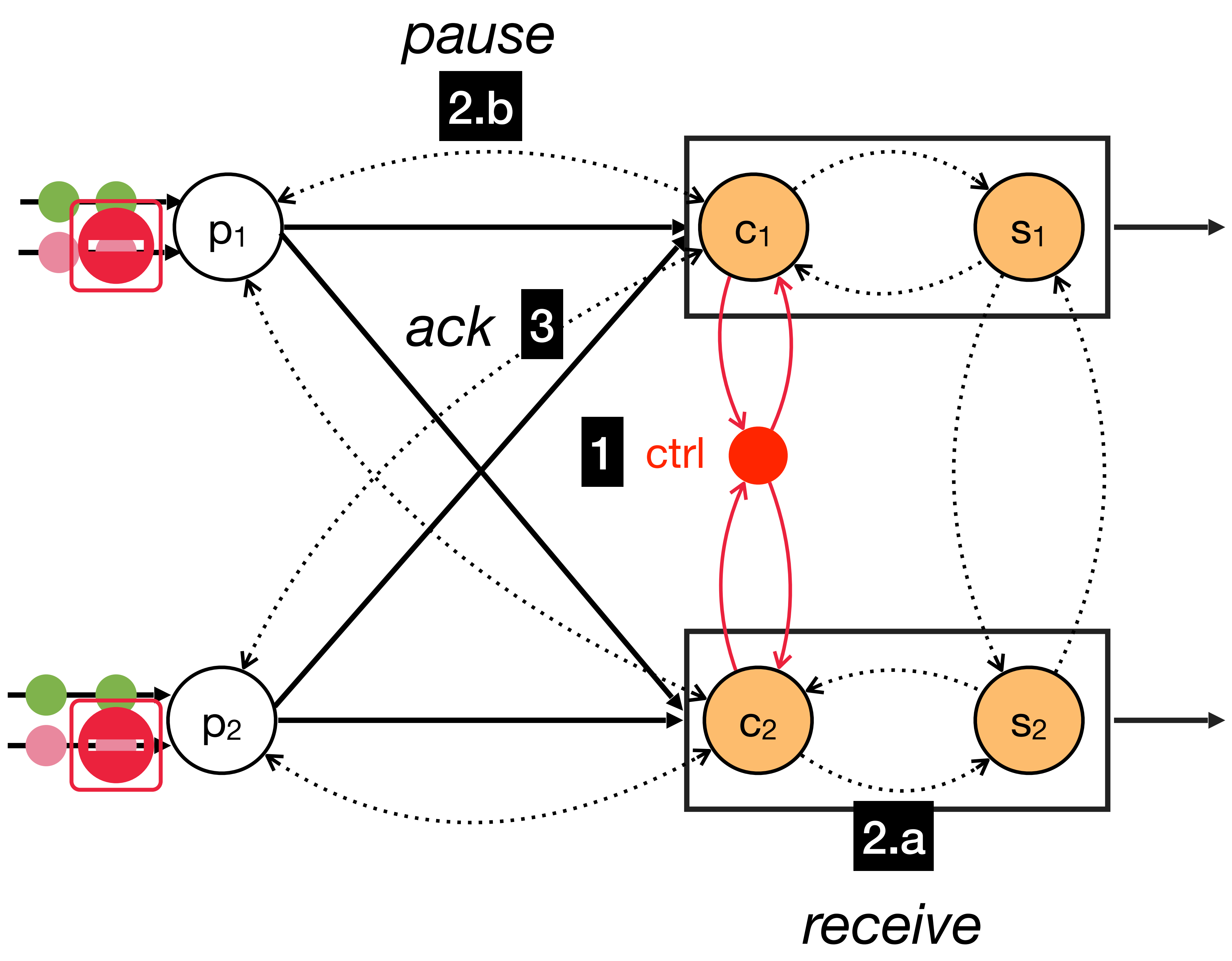


Migrating state from c_1 to c_2

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)

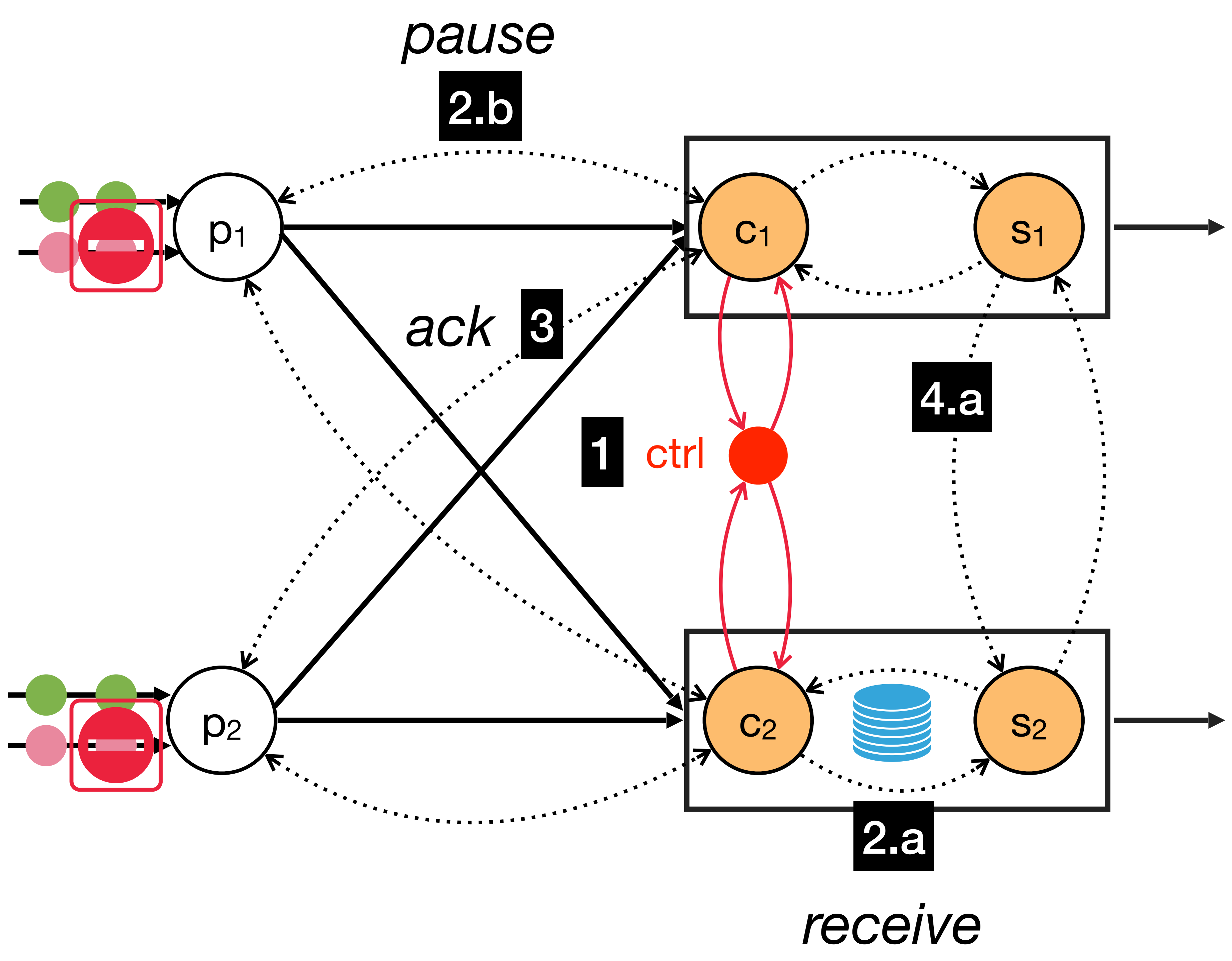


Migrating state from c_1 to c_2

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)

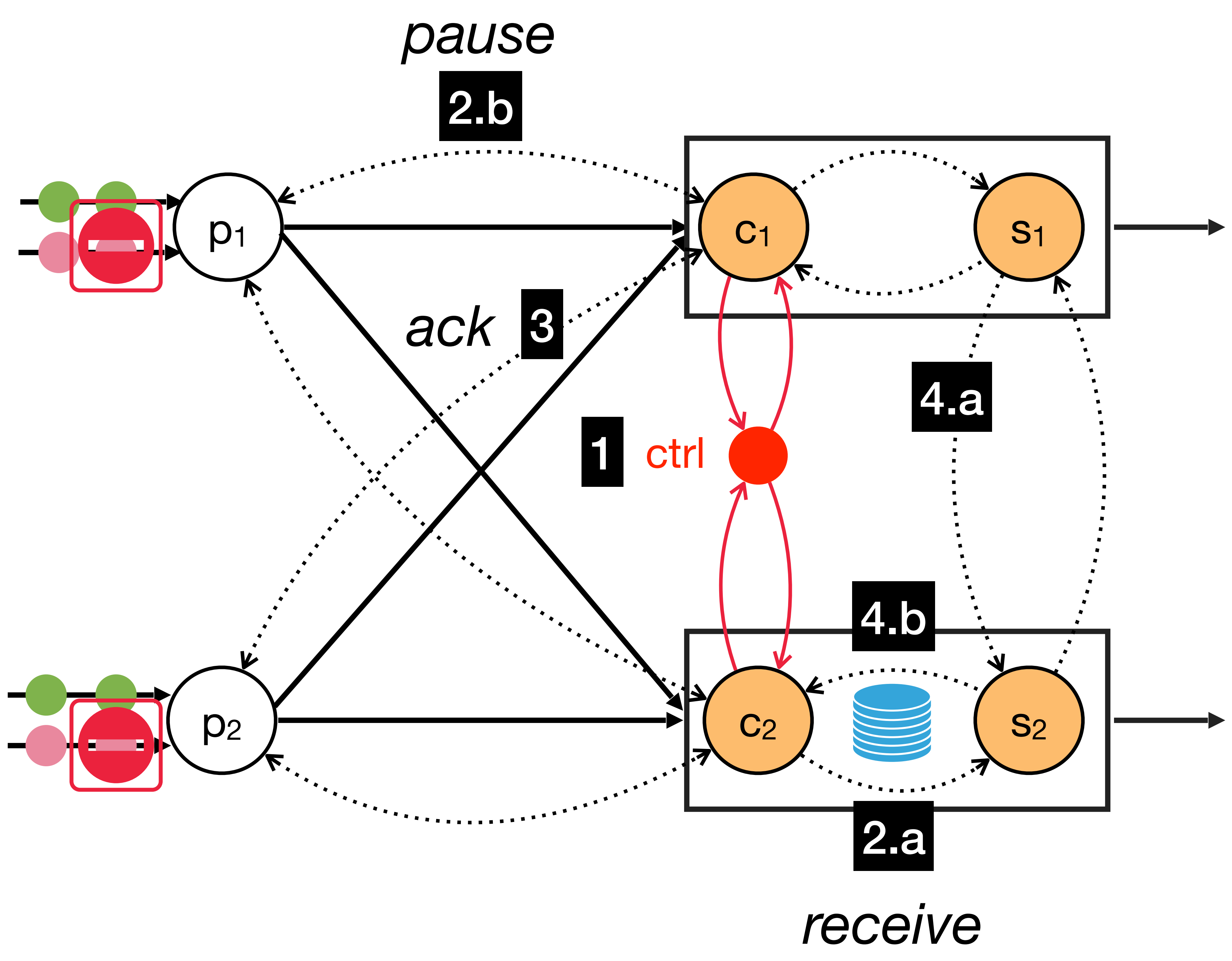


Migrating state from c_1 to c_2

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)

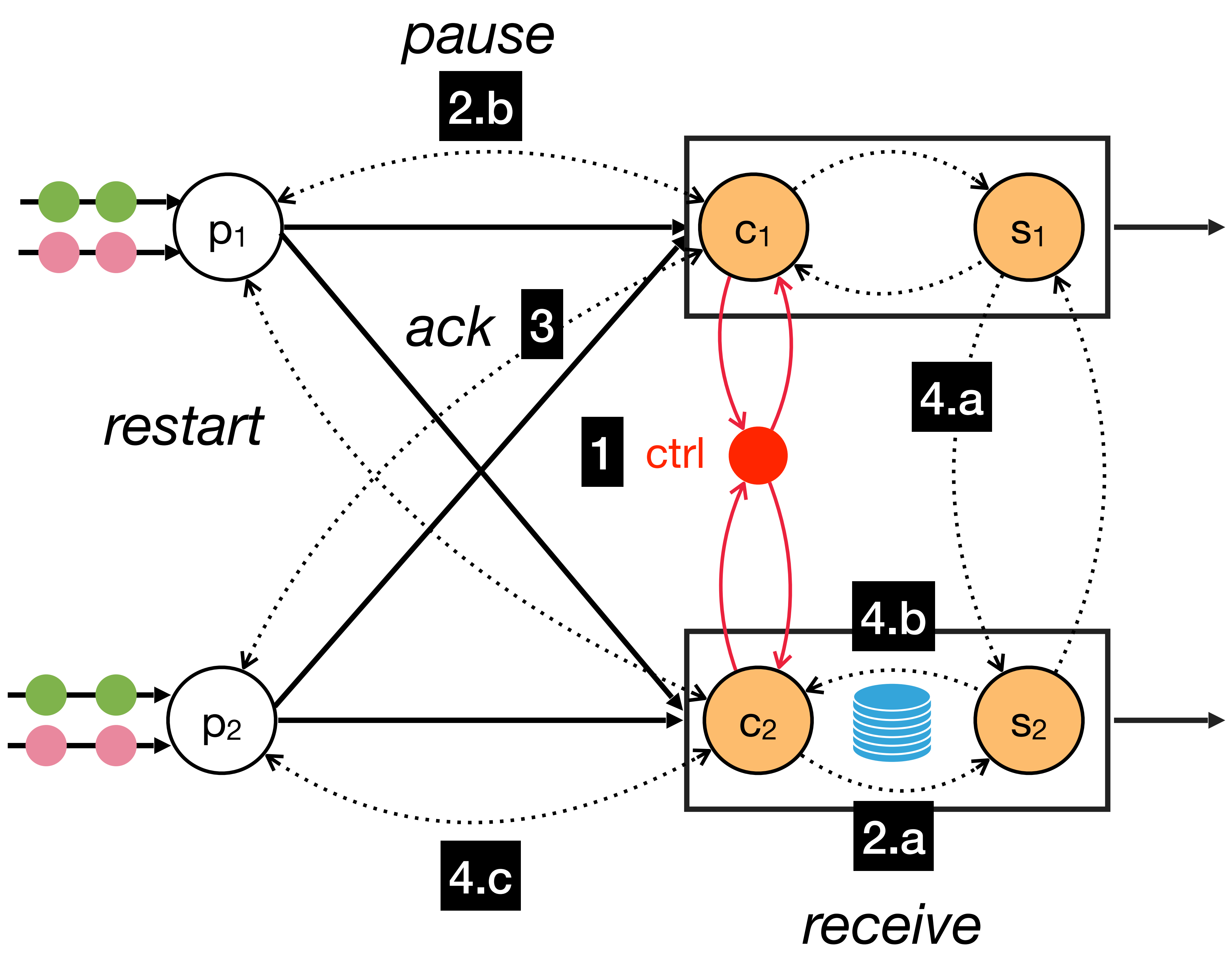


Migrating state from c_1 to c_2

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)

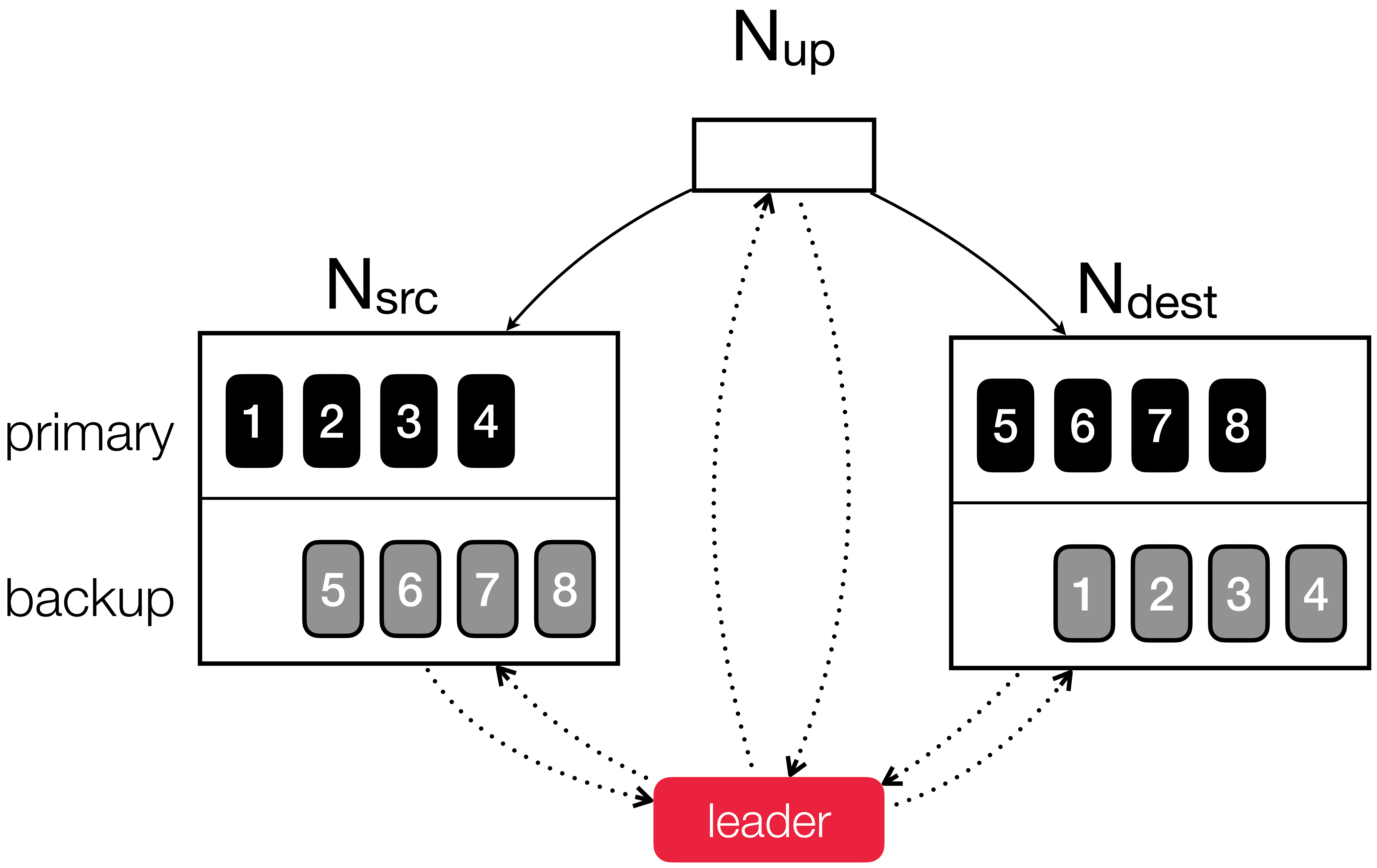


Migrating state from c_1 to c_2

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

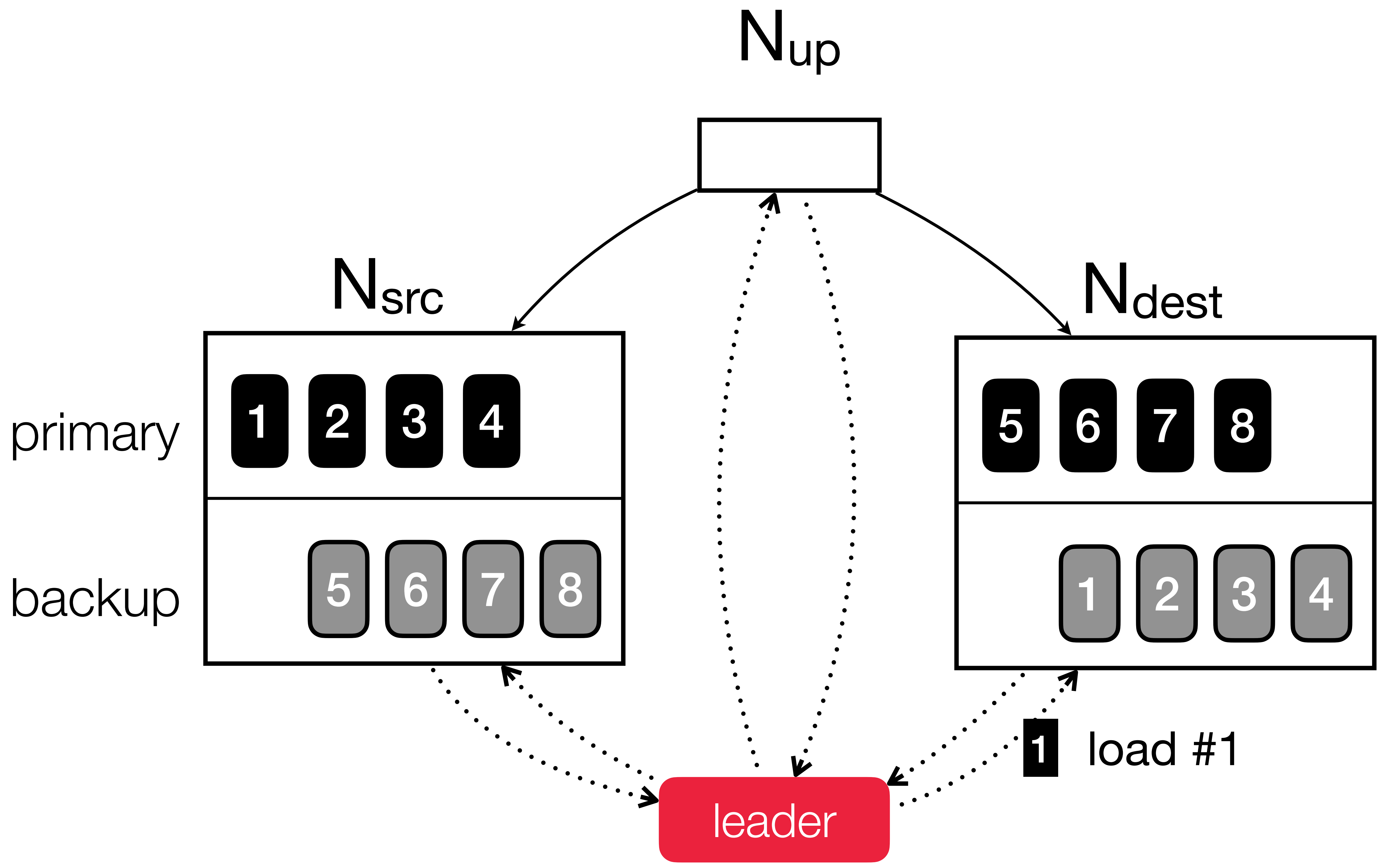
ChronoStream (ICDE'15), Rhino (SIGMOD'20)



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

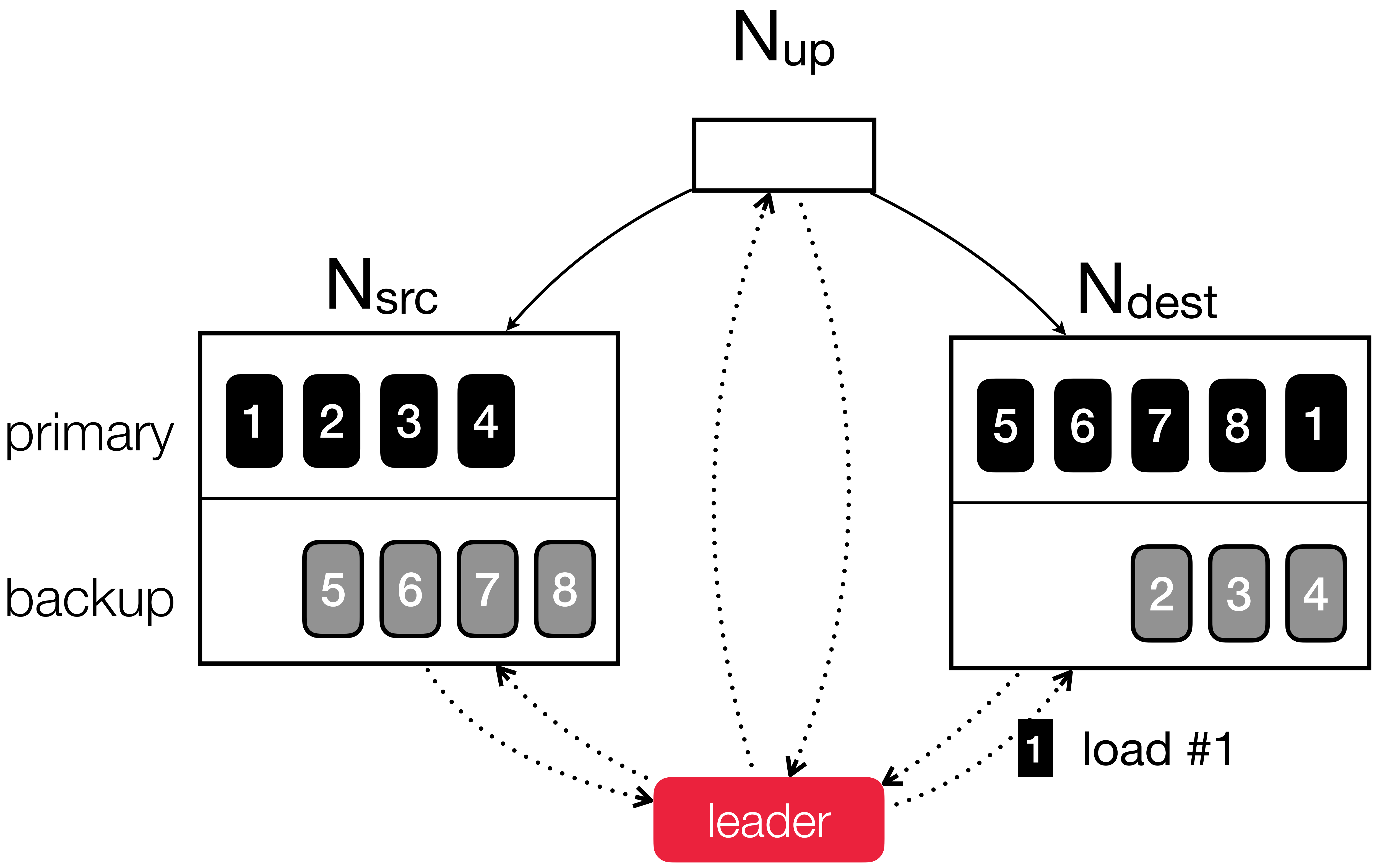
ChronoStream (ICDE'15), Rhino (SIGMOD'20)



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

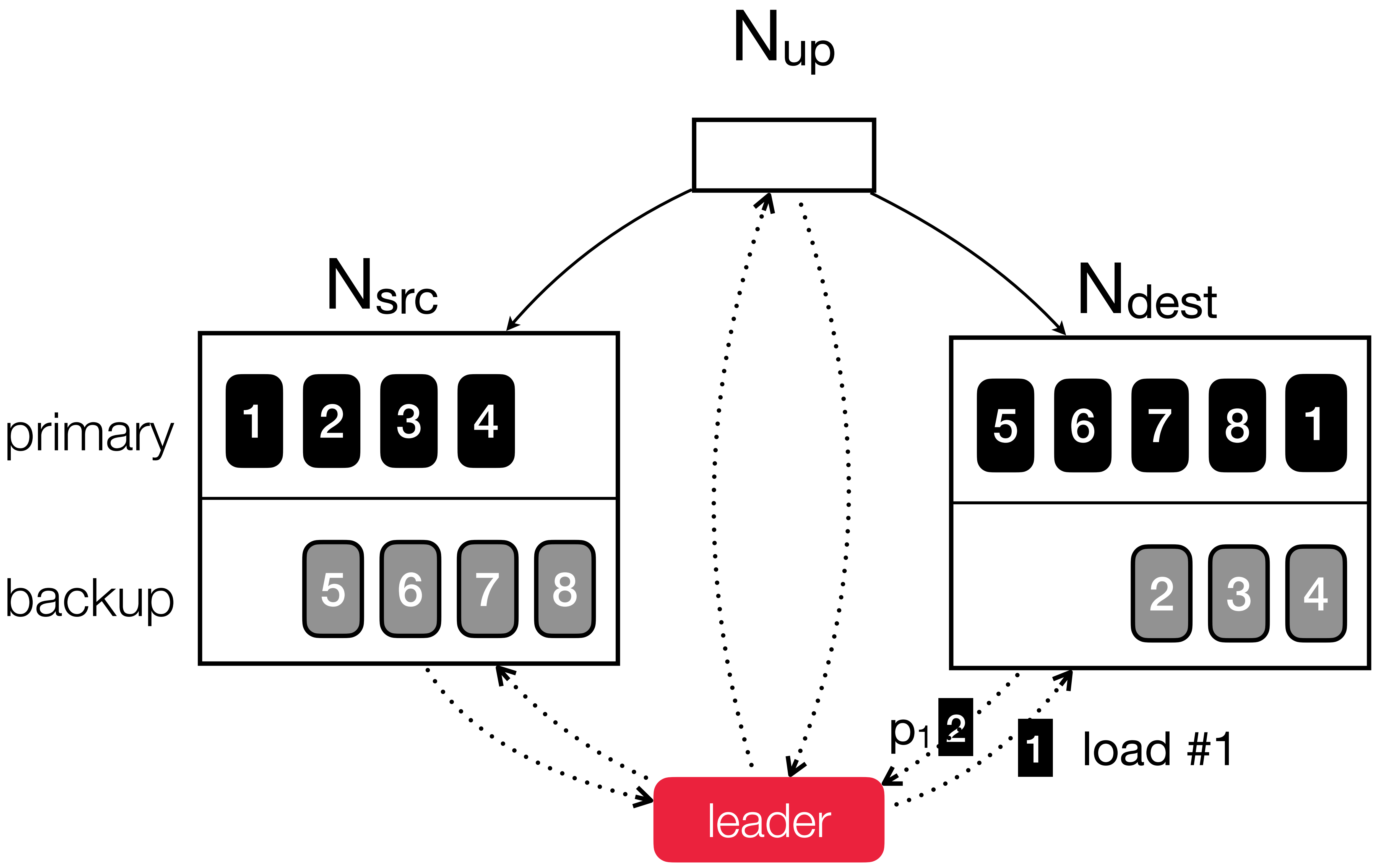
ChronoStream (ICDE'15), Rhino (SIGMOD'20)



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

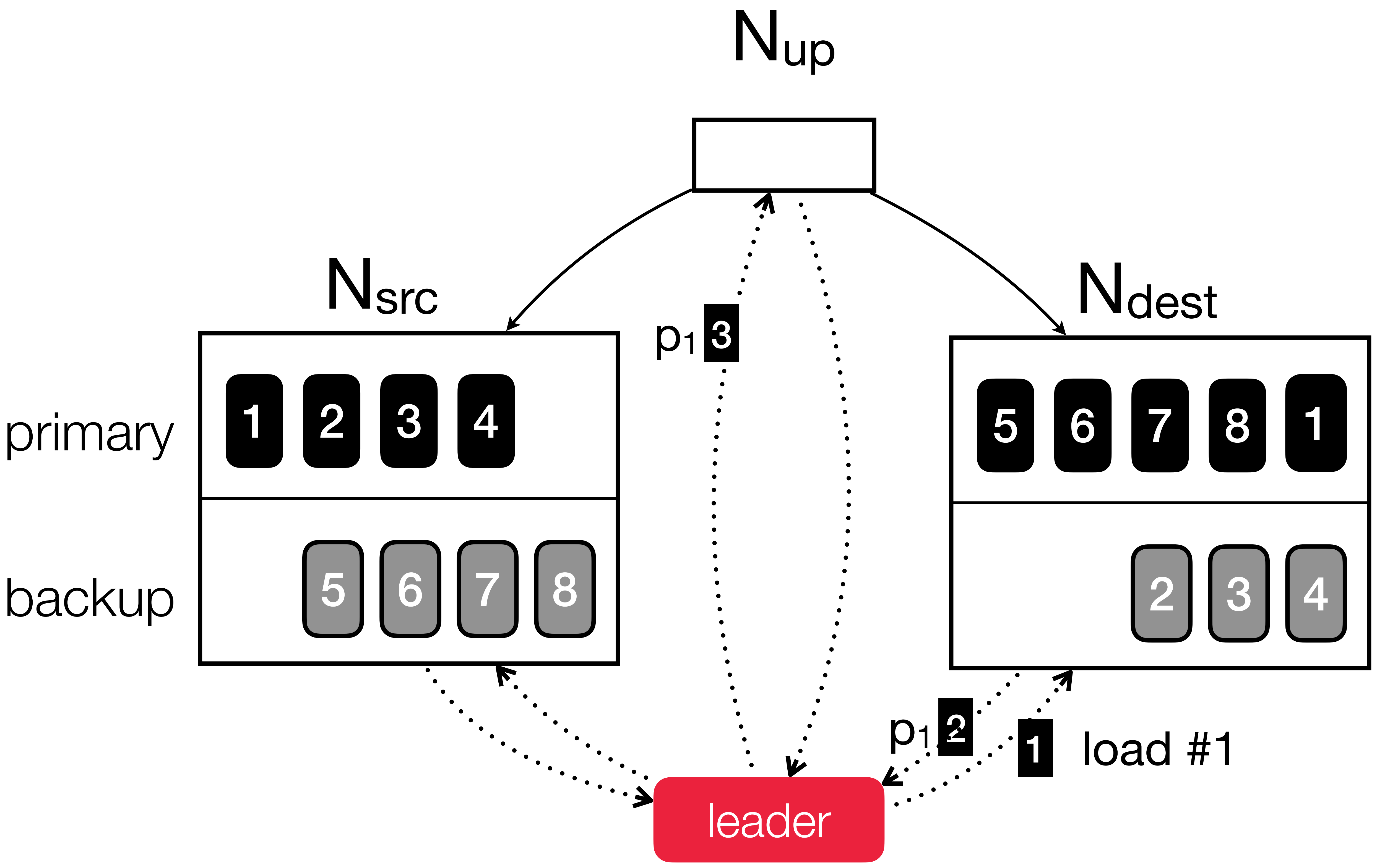
ChronoStream (ICDE’15), Rhino (SIGMOD’20)



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

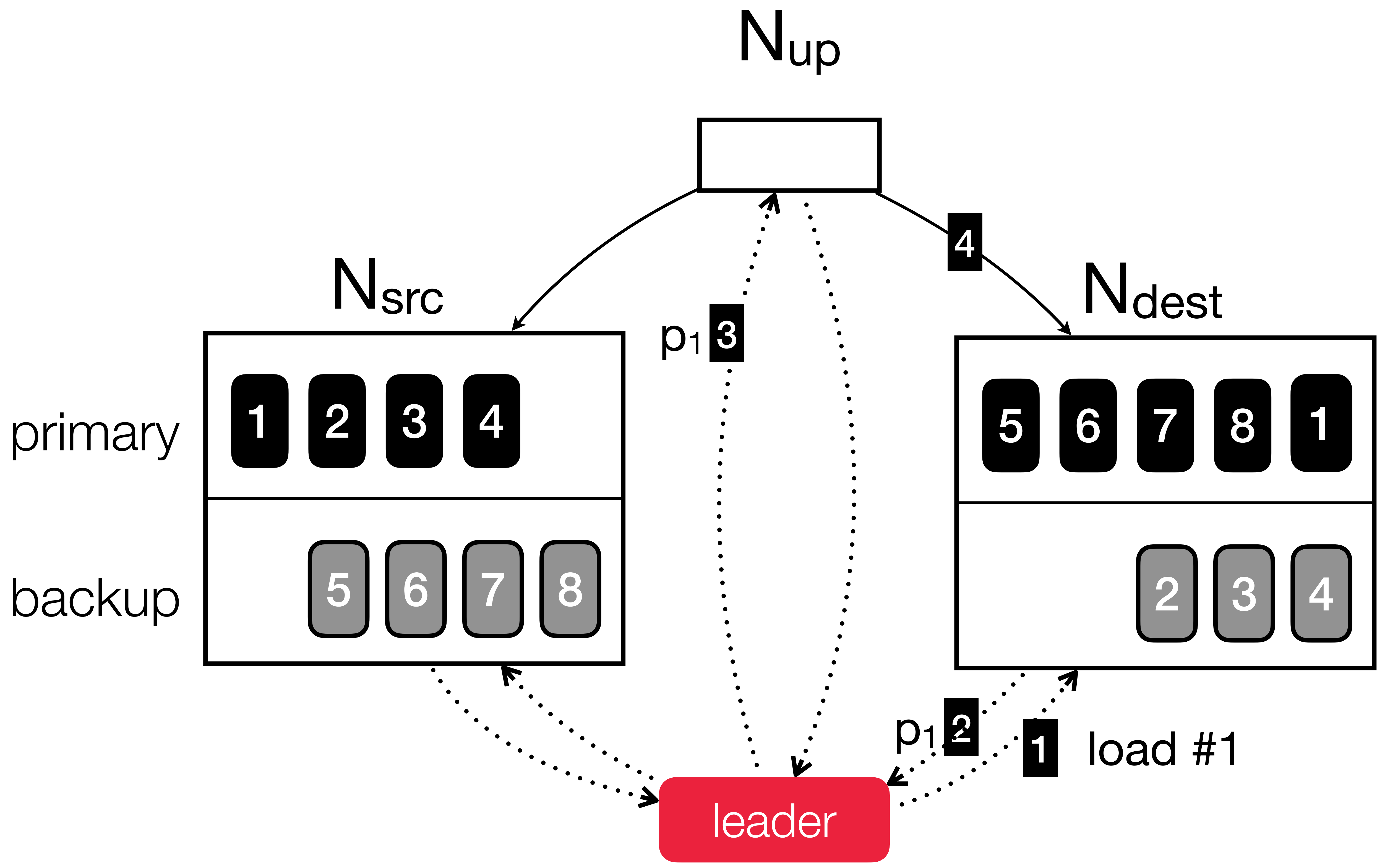
ChronoStream (ICDE’15), Rhino (SIGMOD’20)



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

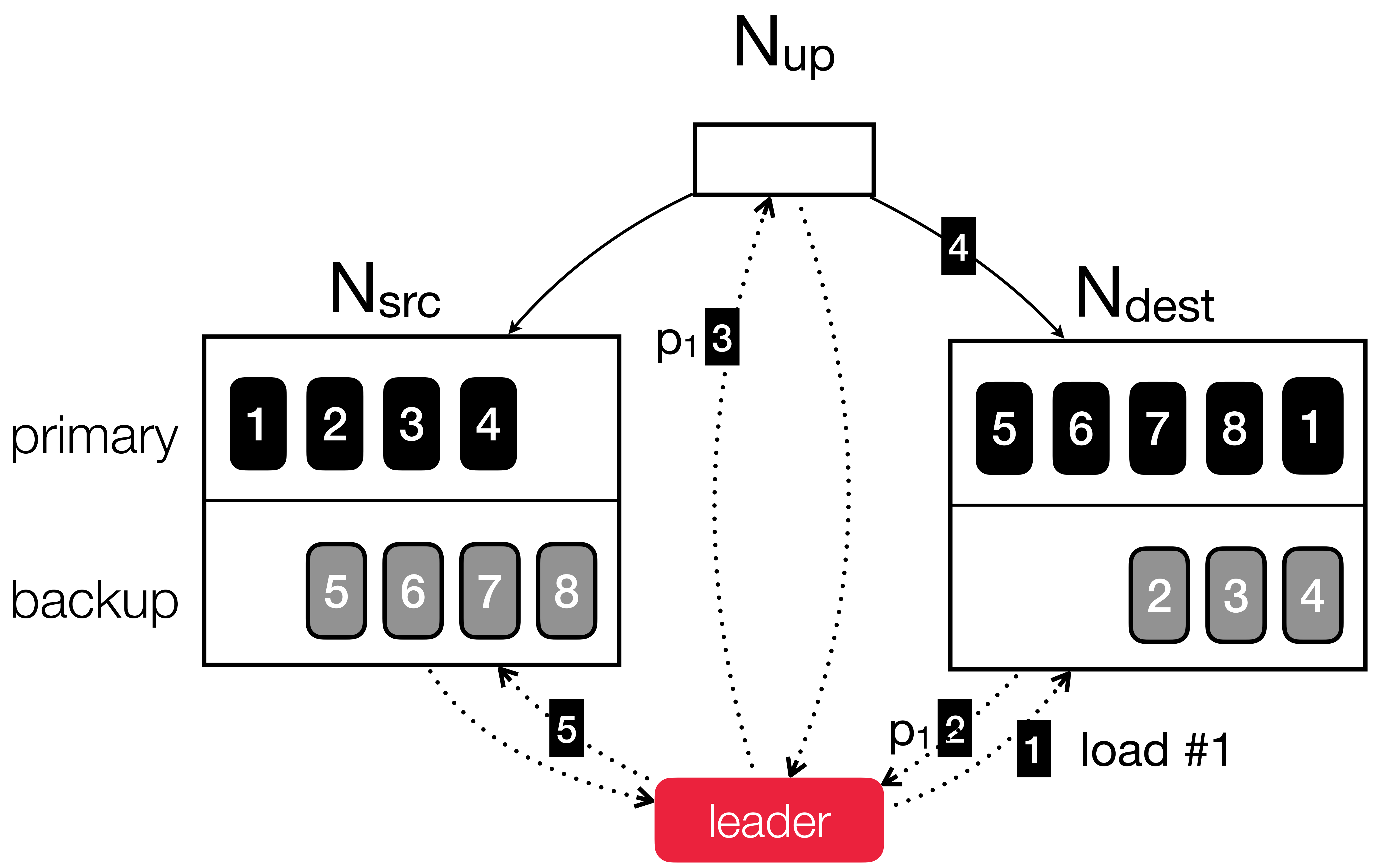
ChronoStream (ICDE'15), Rhino (SIGMOD'20)



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 replay events according to the progress metric provided by N_{dest}.
4. Upstream nodes receive the message and reroute 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

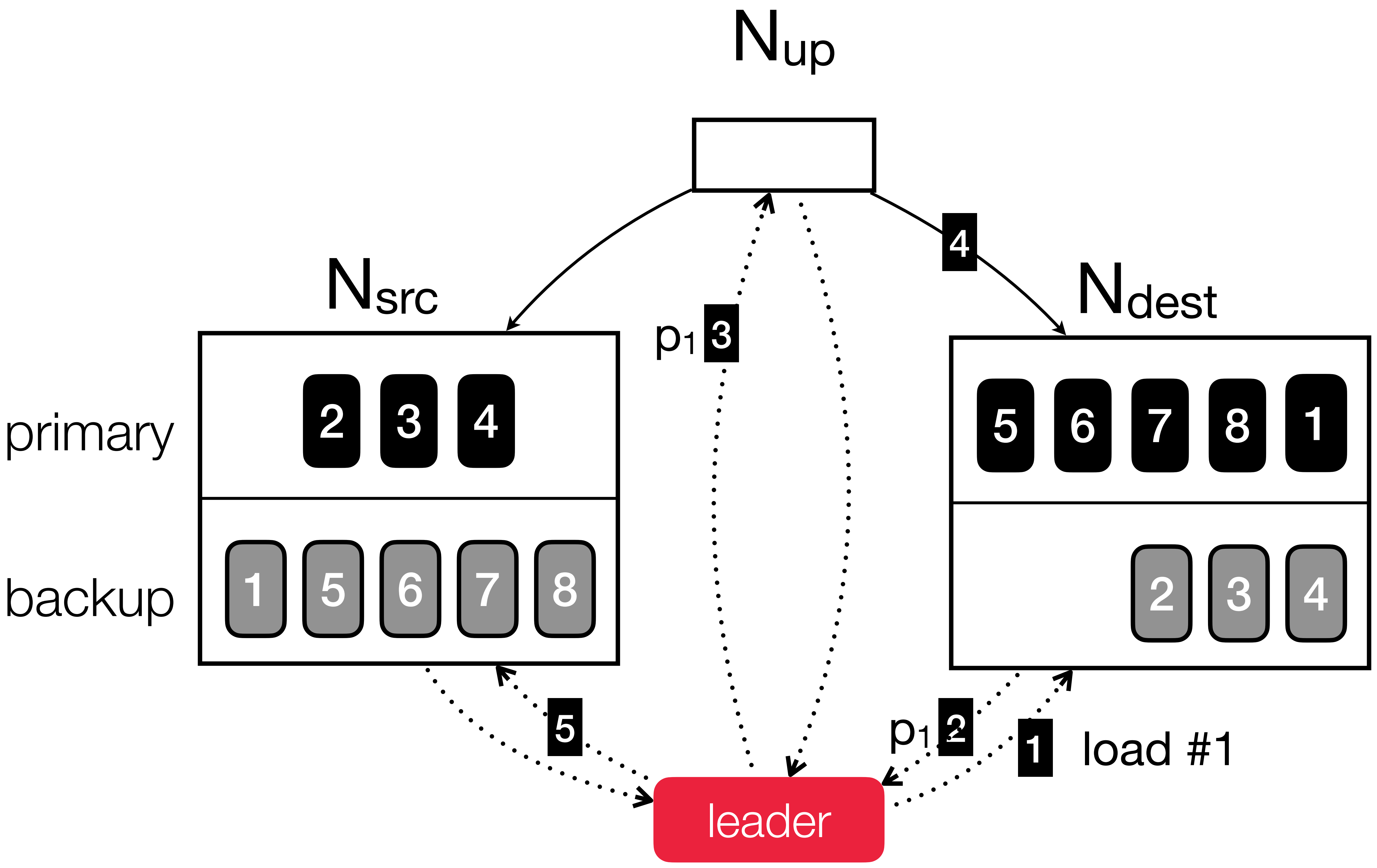
ChronoStream (ICDE’15), Rhino (SIGMOD’20)



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

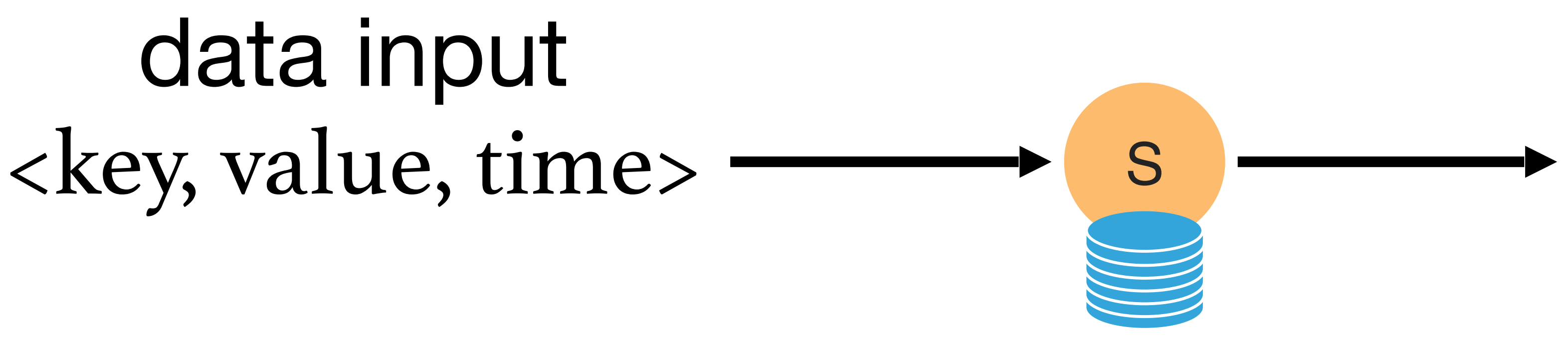
- Move state to be migrated in one operation
- High latency during migration if the state is large

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

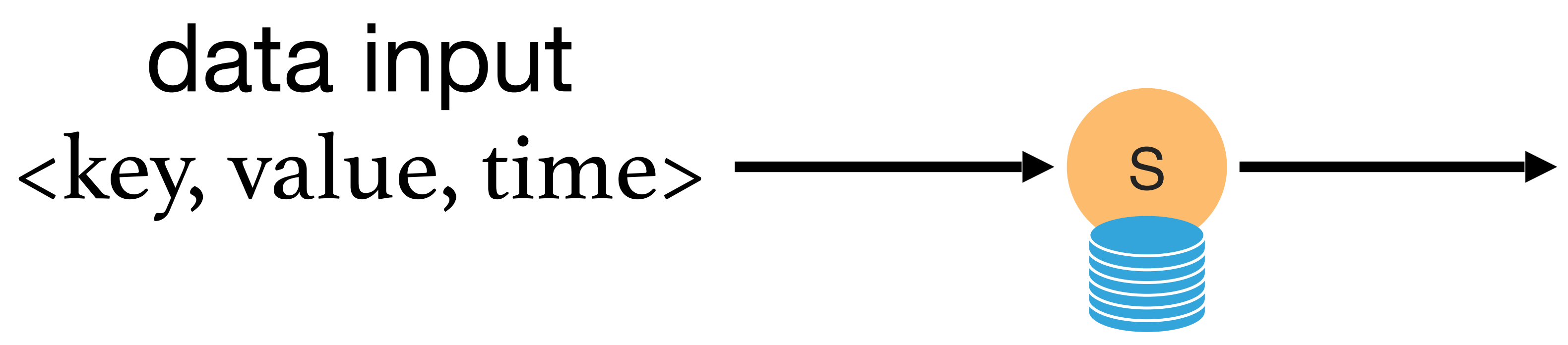
Progressive State migration

Megaphone (VLDB’19)

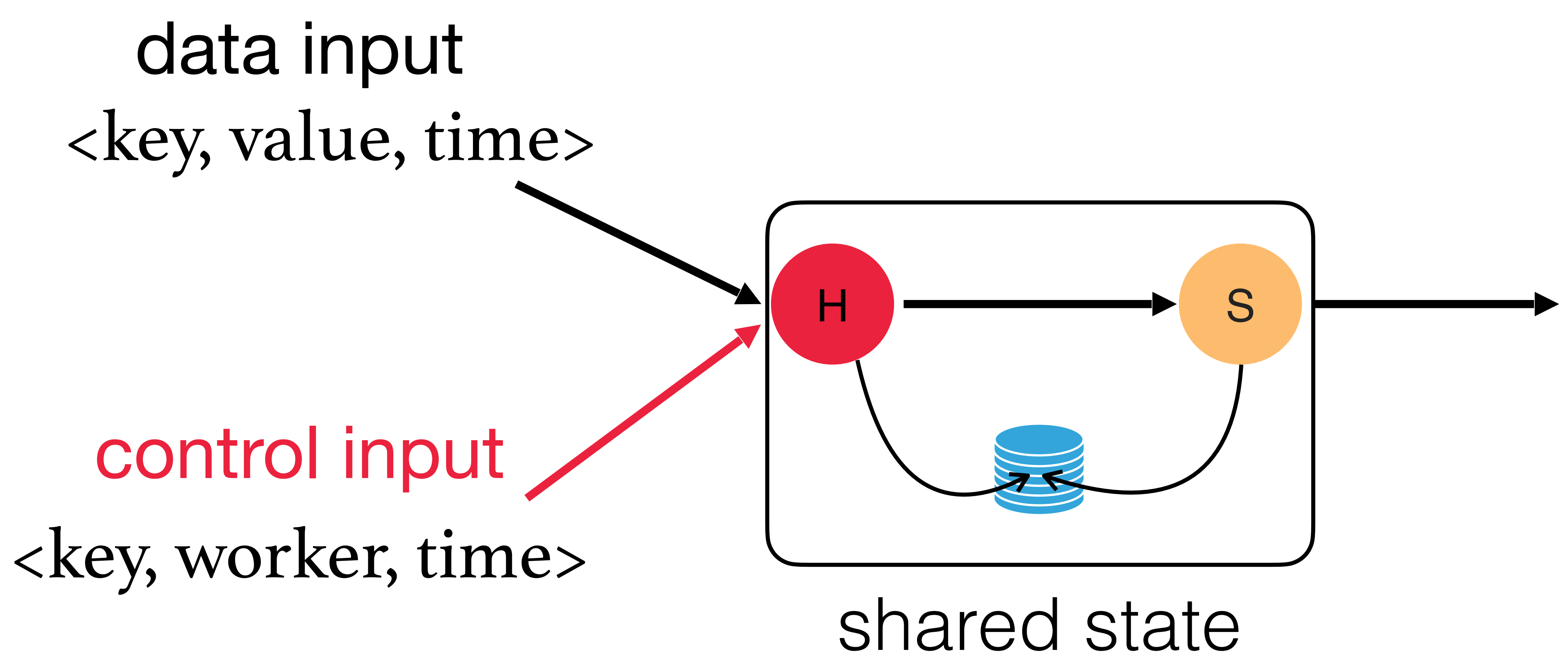


Progressive State migration

Megaphone (VLDB'19)



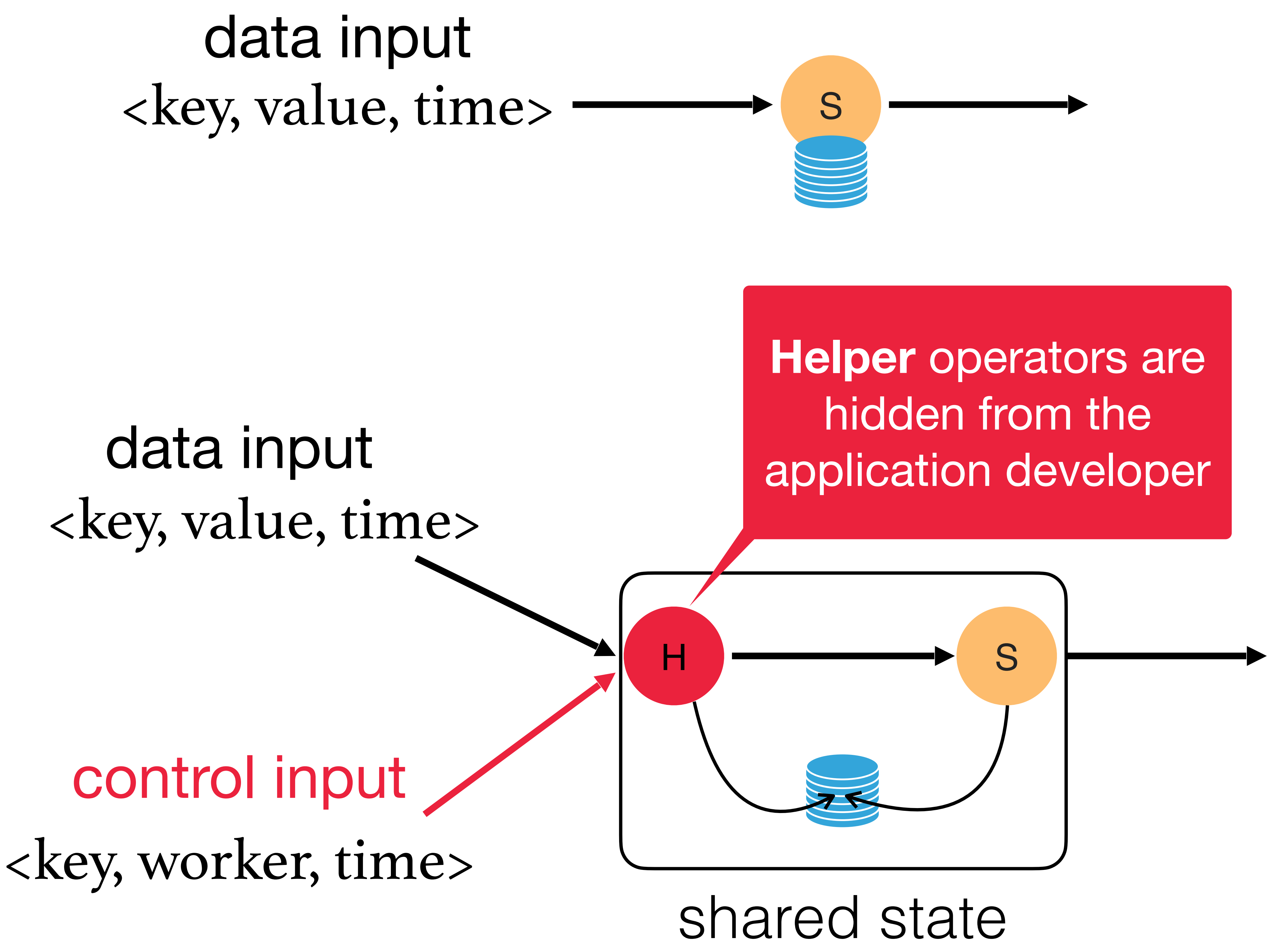
Each stateful operator is augmented with a **helper upstream operator** which accepts a control stream as input



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

Progressive State migration

Megaphone (VLDB’19)

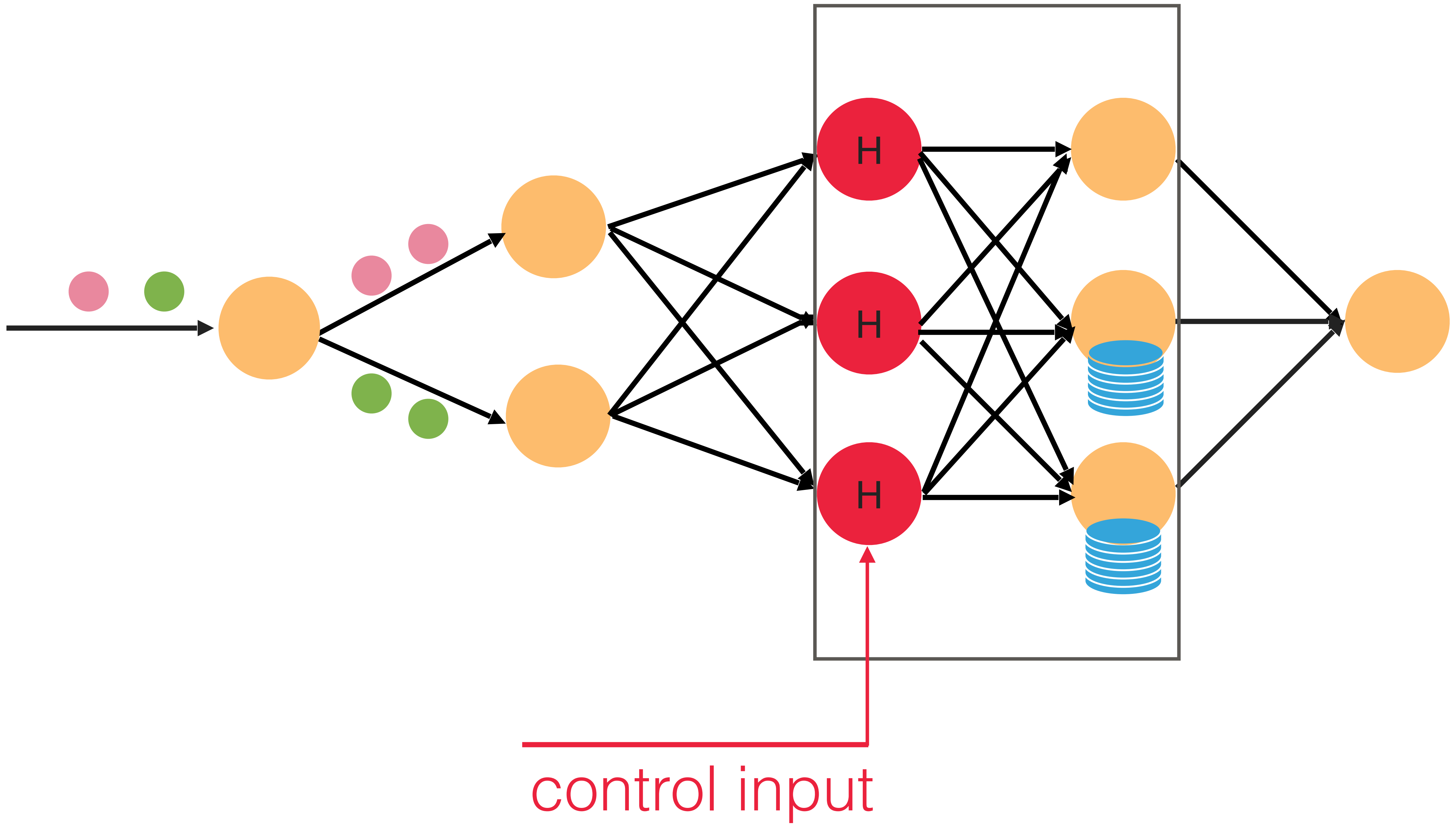


Each stateful operator is augmented with a **helper upstream operator** which accepts a control stream as input

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

Progressive State migration

Megaphone (VLDB'19)

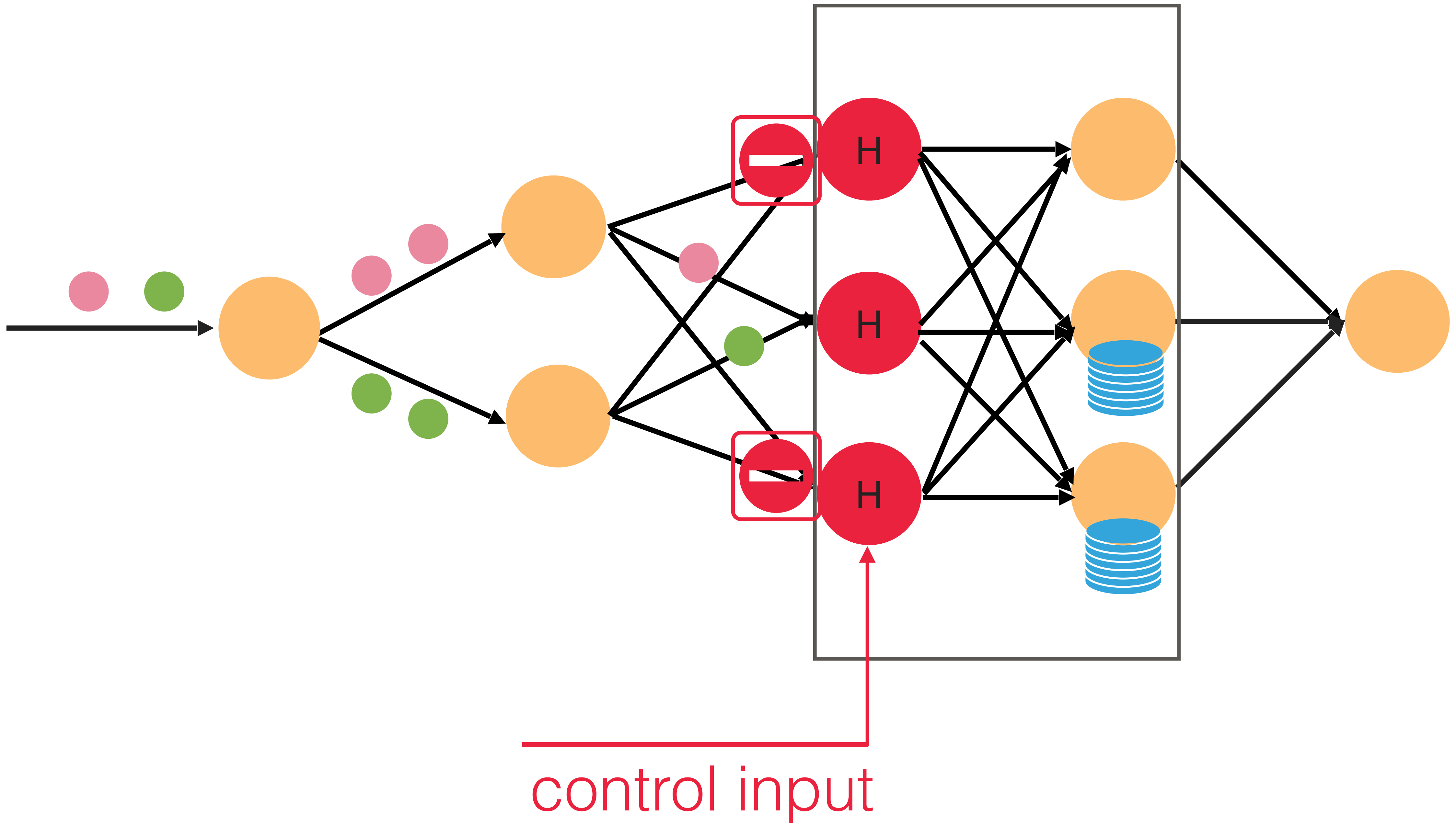


Helper operators:

- **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)

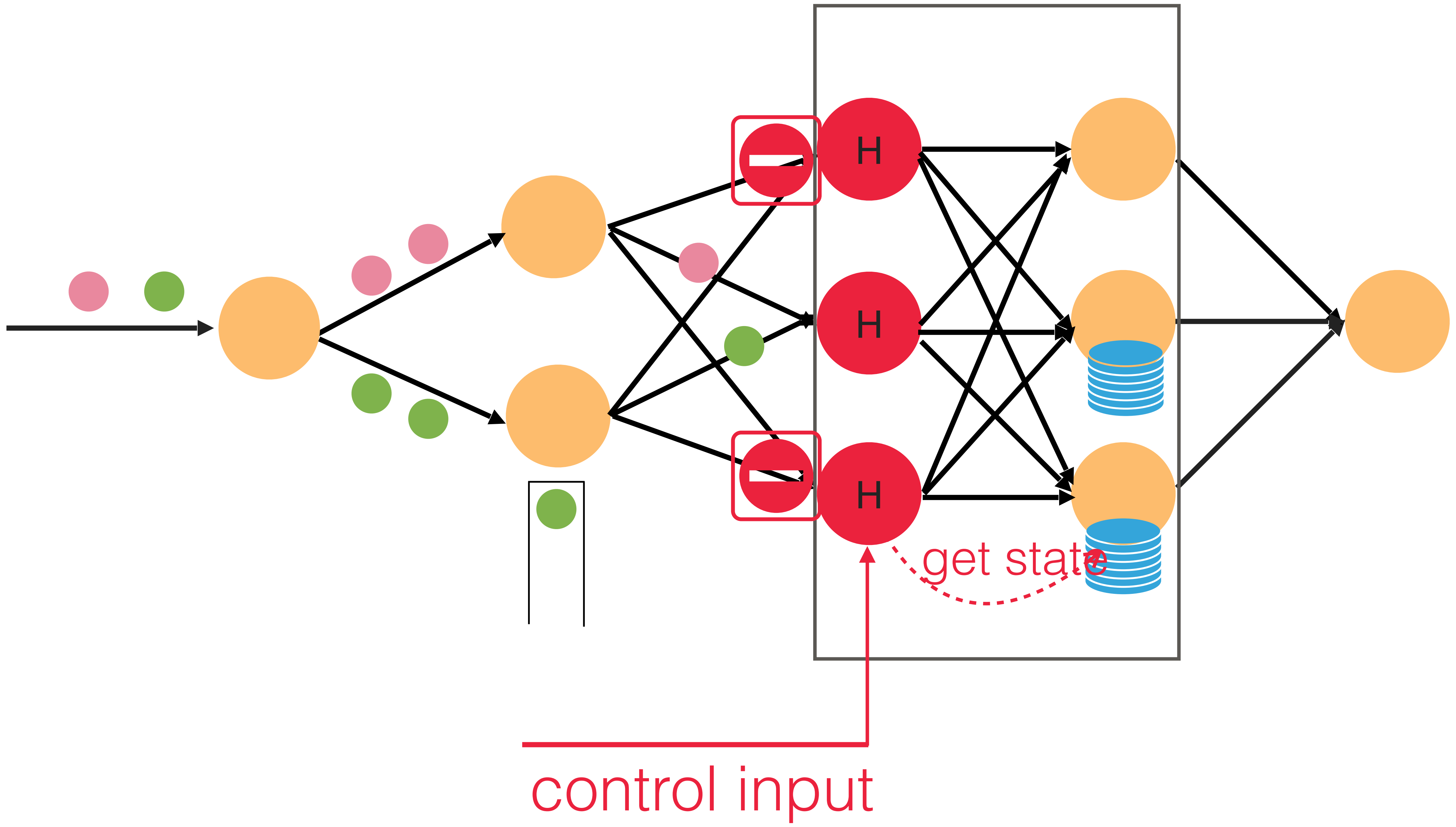


Helper operators:

- **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)

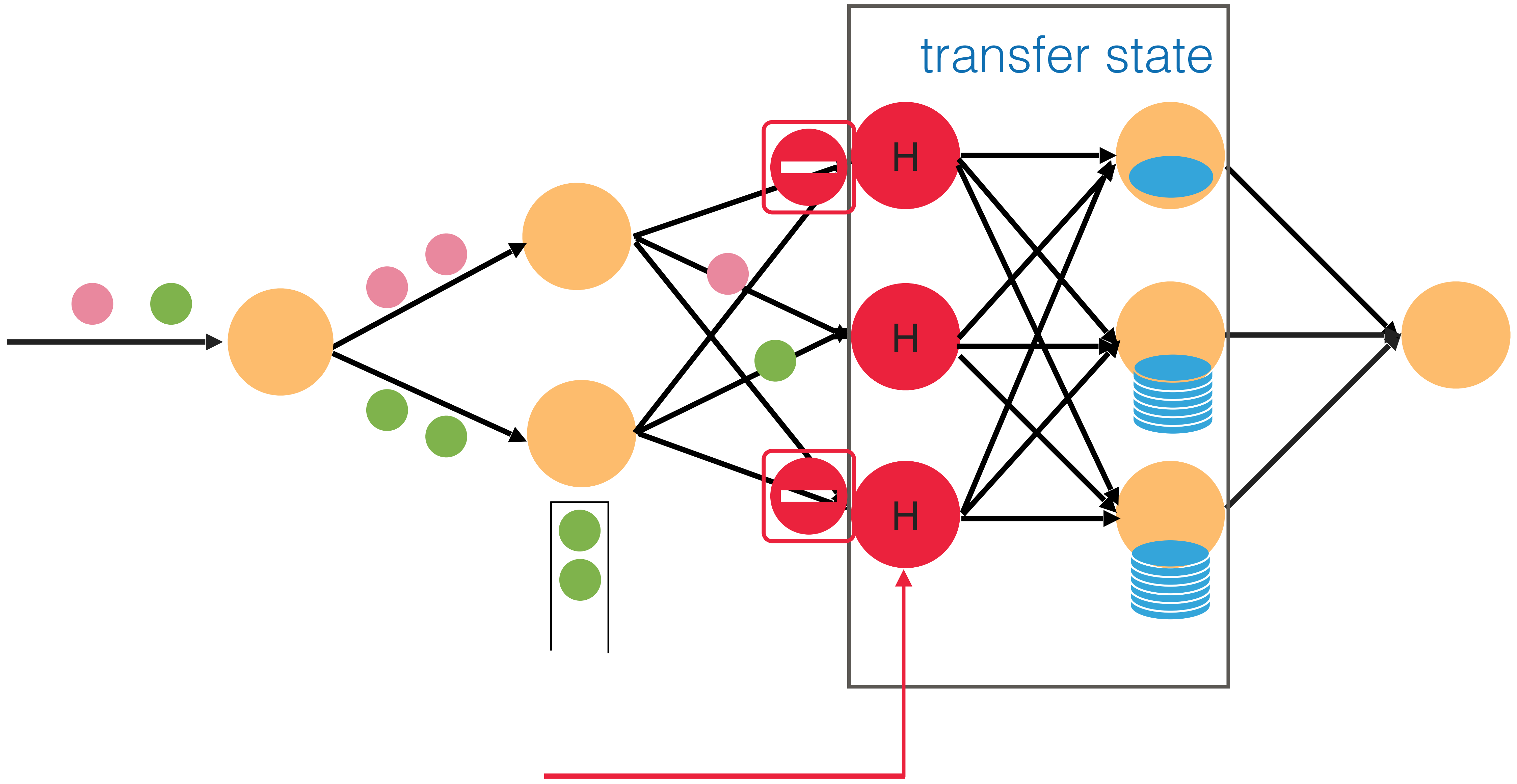


Helper operators:

- **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)

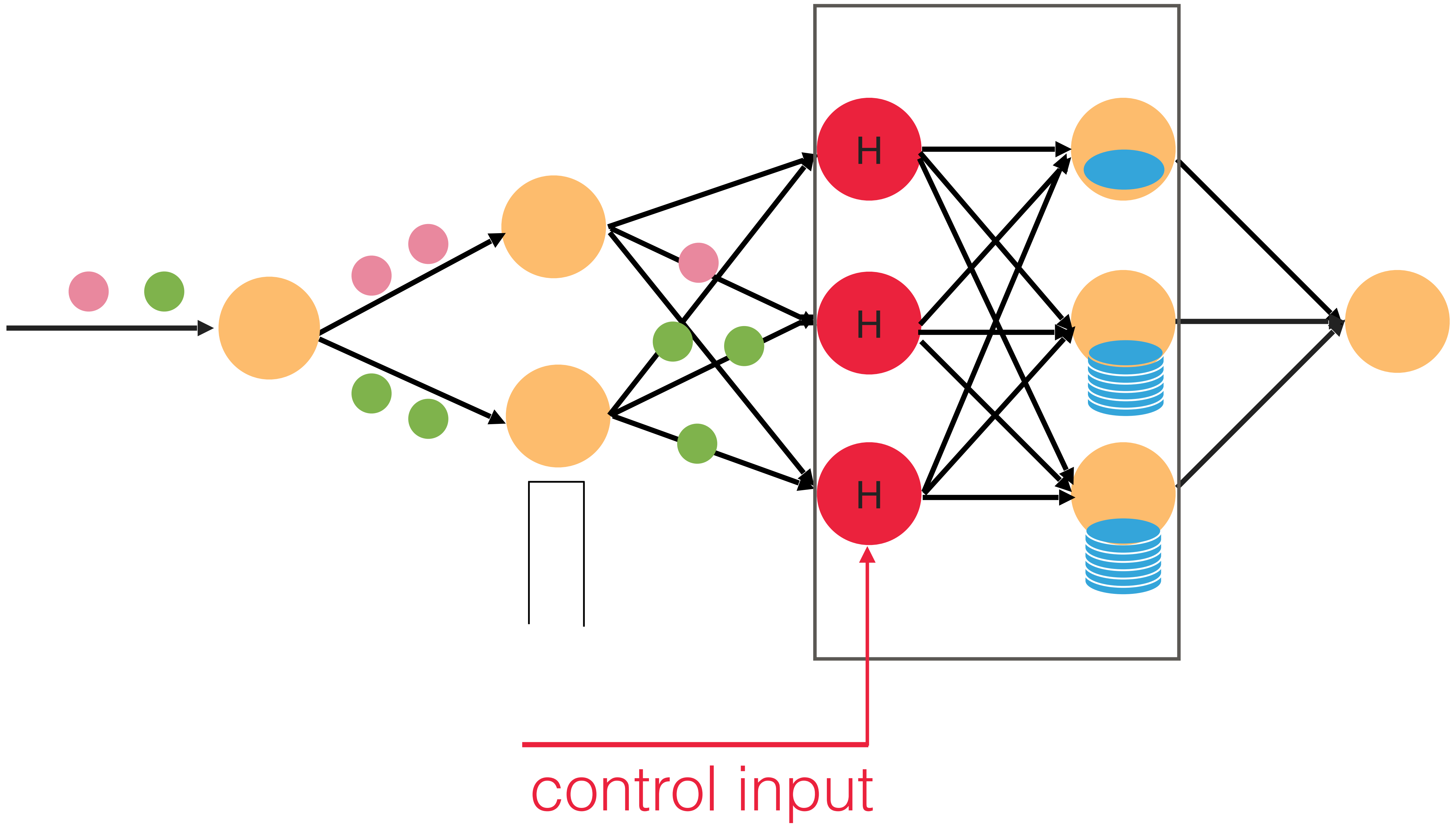


Helper operators:

- **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:

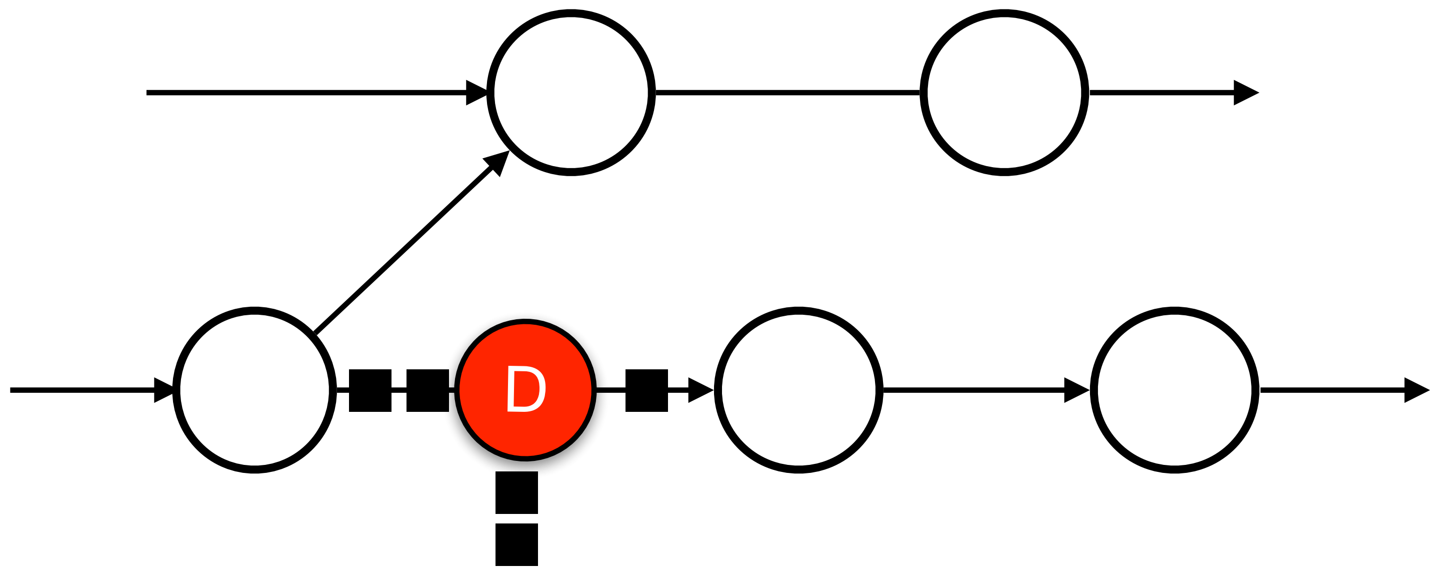
- **buffer** data that cannot be safely routed yet and configuration commands that cannot yet be applied

- check the **frontier (watermark)** at the output of the stateful operator to **ensure only complete state is migrated**

Load management approaches

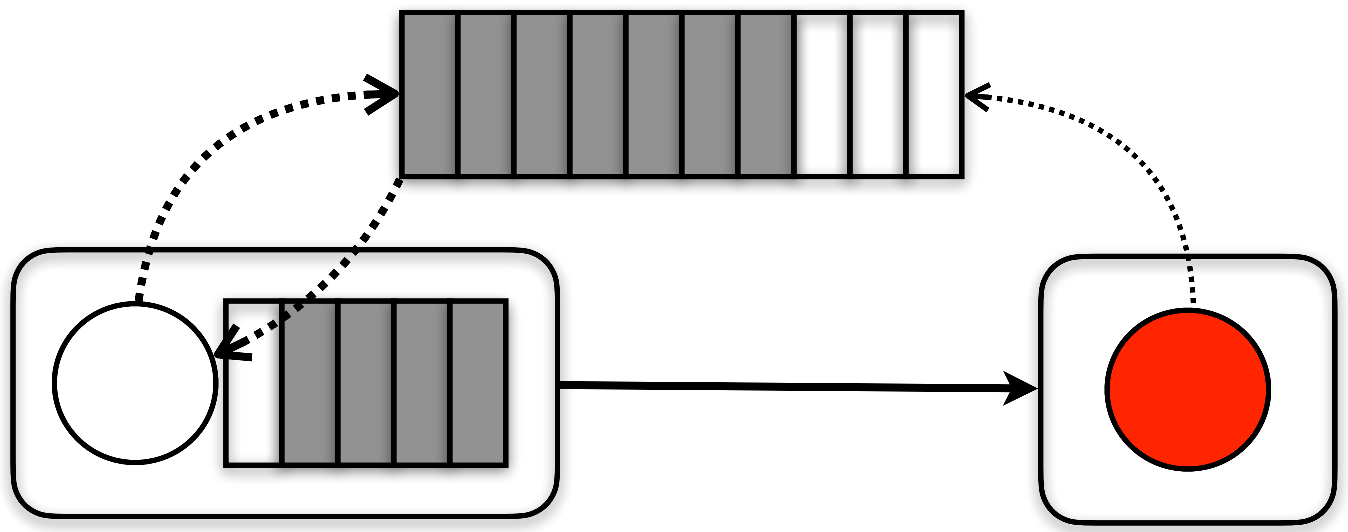
Comparison

Load shedding



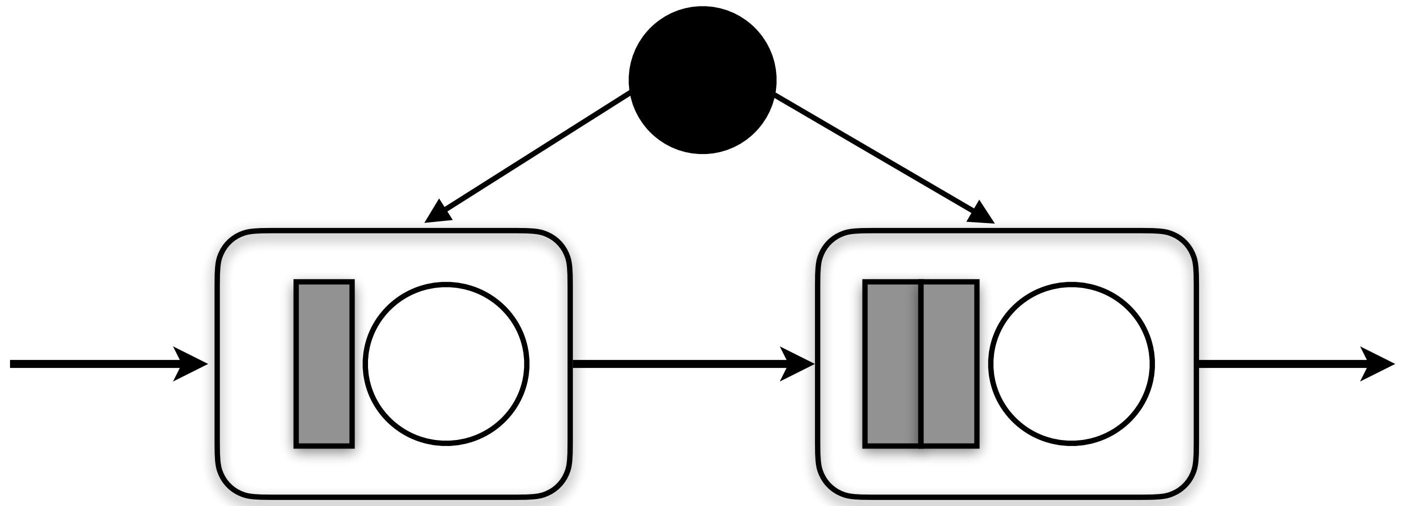
Selectively drop tuples

Back-pressure



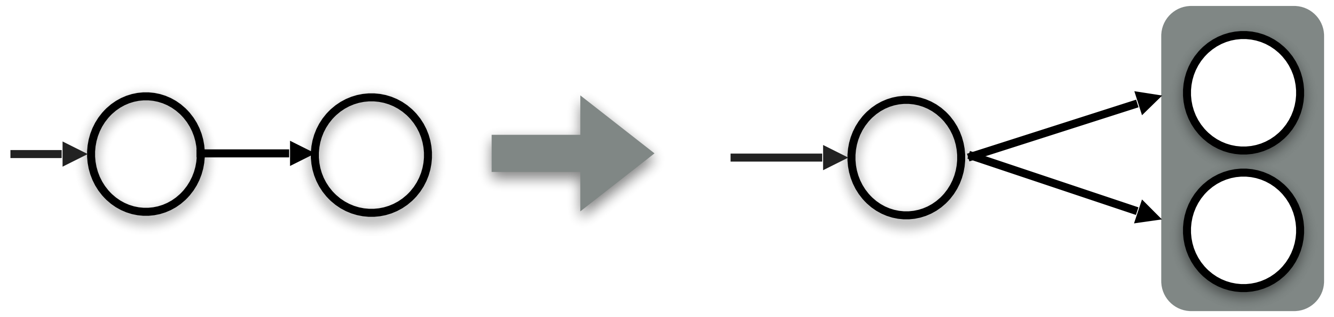
Slow down the flow of data

Load-aware scheduling



Select operator order

Elasticity



Scale resource allocation

Load management approaches

Vintage vs. modern

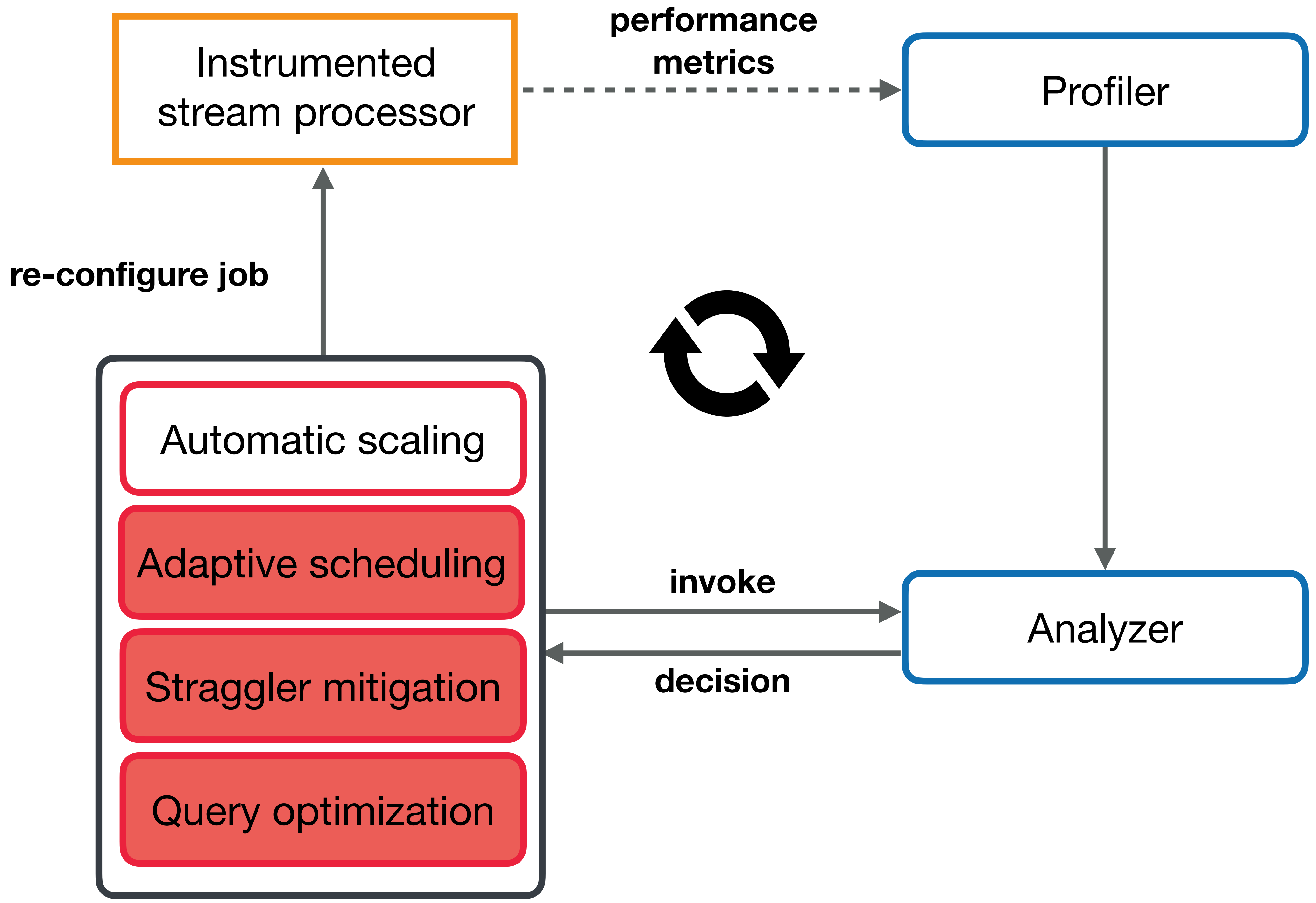
- Temporary latency increase is preferred to degrading results quality in modern systems.
- Flow control mechanisms today are implemented on a per-job basis as opposed to load shedding techniques applied to the set of queries running in the system as a whole.
- Early load management techniques make strong assumptions about the operators and their properties while newer methods are more generally applicable.
- Scaling down was not a matter of concern before cloud deployments.
- Modern systems consider persistent queues and inputs to provide correctness guarantees.
- Reconfiguration in modern systems is often based on the fault-tolerance mechanism.
- Early and new techniques are often combined in today’s systems.

Other reconfiguration cases

- Change parallelism
  - scale out to process increased load
  - scale in to save resources
- Fix bugs or change business logic
- Optimize execution plan
- Change operator placement
  - skew and straggler mitigation
- Migrate to a different cluster or software version

Self-managed and re-configurable stream processing

- Can we collect and analyze meaningful traces efficiently?
- Can we apply query optimization techniques to stream processing?
- Can we build predictive performance models and take actions in real-time?
- How do optimizations interact and what is the cost of deployment?



References

- Nesime Tatbul, Uğur Çetintemel, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. **Load shedding in a data stream manager**. (VLDB 2003)
- Nesime Tatbul and Stan Zdonik. **Window-aware load shedding for aggregation queries over data streams**. (VLDB 2006)
- Nesime Tatbul, Uğur Çetintemel, and Stan Zdonik. **Staying fit: Efficient load shedding techniques for distributed stream processing**. (VLDB 2007)
- N. R. Katsipoulakis, A. Labrinidis, and P. K. Chrysanthis. **Concept-driven load shedding: Reducing size and error of voluminous and variable data streams**. (IEEE Big Data 2018).
- Babcock, Brian, Shivnath Babu, Rajeev Motwani, and Mayur Datar. **Chain: Operator scheduling for memory minimization in data stream systems**. (SIGMOD 2003).
- Carney, Don, Uğur Çetintemel, Alex Rasin, Stan Zdonik, Mitch Cherniack, and Mike Stonebraker. **Operator scheduling in a data stream manager**. (VLDB 2003).
- H. T. Kung, T. Blackwell, and A. Chapman. **Credit-based flow control for atm networks: Credit update protocol, adaptive credit allocation and statistical multiplexing**. (ACM SGCOMM 1994).
- 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 2018).
- Moritz Hoffmann, Andrea Lattuada, Frank McSherry, Vasiliki Kalavri, John Liagouris, Timothy Roscoe. **Megaphone: Latency-conscious state migration for distributed streaming dataflows**. (VLDB 2019).
- Shah, Mehul A., Joseph M. Hellerstein, Sirish Chandrasekaran, and Michael J. Franklin. **Flux: An adaptive partitioning operator for continuous query systems**. (ICDE 2003).
- Castro Fernandez, Raul, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. **Integrating scale out and fault tolerance in stream processing using operator state management**. (SIGMOD 2013).
- Y.-C. Tu, S. Liu, S. Prabhakar, and B. Yao. **Load shedding in stream databases: A control-based approach**. (VLDB 2006).
- Y. Mei, L. Cheng, V. Talwar, M. Y. Levin, G. Jacques-Silva, N. Simha, A. Banerjee, B. Smith, T. Williamson, S. Yilmaz, et al. **Turbine: Facebook's service management platform for stream processing**. (ICDE 2020).
- B. Lohrmann, P. Janacik, and O. Kao. **Elastic stream processing with latency guarantees**. (ICDCS 2015).
- T. Heinze, Z. Jerzak, G. Hackenbroich, and C. Fetzer. **Latency-aware elastic scaling for distributed data stream processing systems**. (DEBS 2014).
- Y. Wu and K.-L. Tan. **ChronoStream: Elastic stateful stream computation in the cloud**. (ICDE 2015).
- Del Monte, Bonaventura, Steffen Zeuch, Tilmann Rabl, and Volker Markl. **Rhino: Efficient Management of Very Large Distributed State for Stream Processing Engines**. (SIGMOD 2020).

SIGMOD 2020 Tutorial

Beyond Analytics

The Evolution of Stream Processing Systems

Load management & Elasticity

Paris Carbone, Marios Fragkoulis, Vasiliki Kalavri, Asterios Katsifodimos

Slides: streaming-research.github.io/Tutorial-SIGMOD-2020

RI. SE

BOSTON UNIVERSITY

TU Delft