CS 591 K1: Data Stream Processing and Analytics

Spring 2020

3/17: High availability, recovery semantics, and guarantees

Vasiliki (Vasia) Kalavri
vkalavri@bu.edu

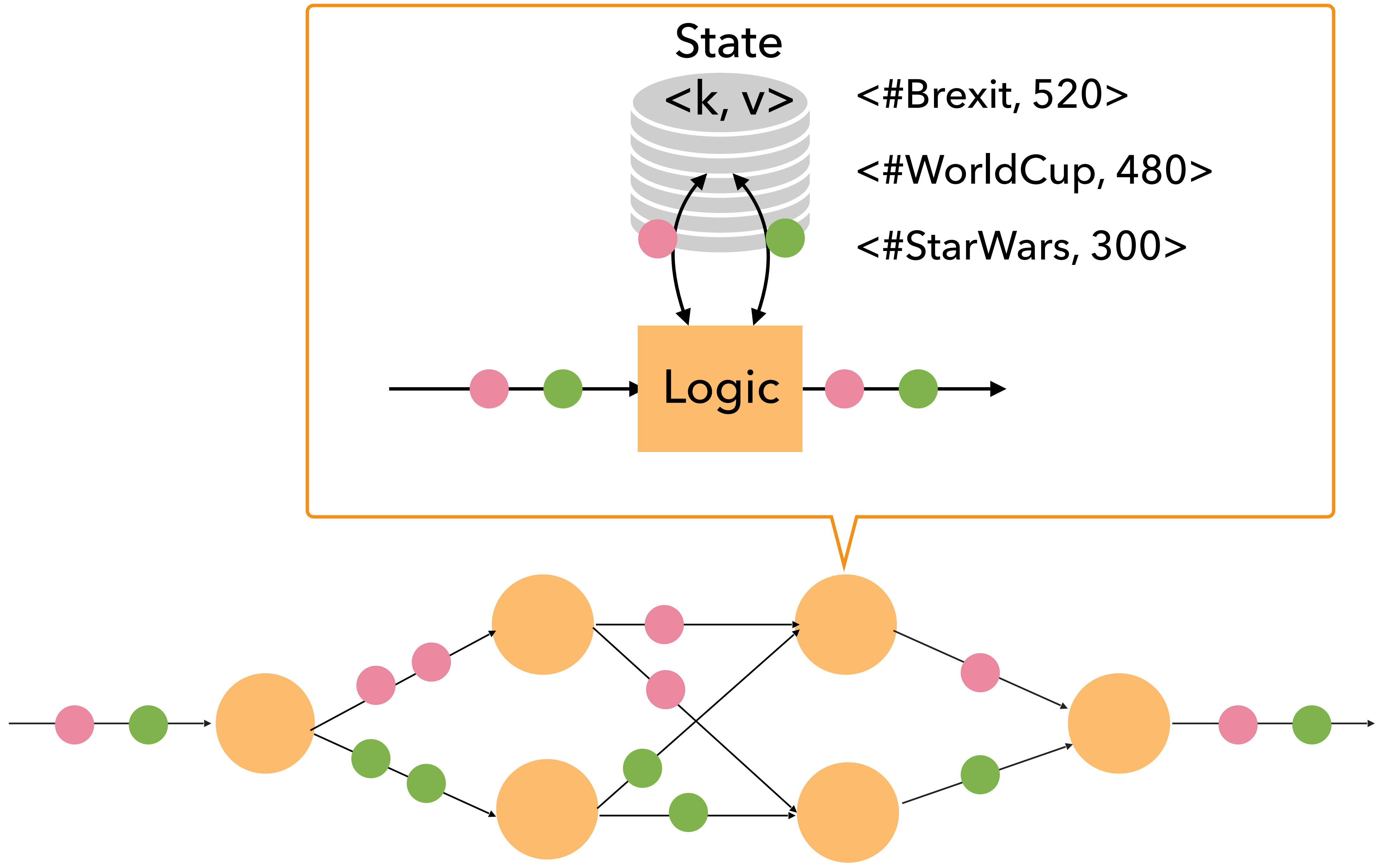
Today's topics

- High-availability and fault-tolerance in distributed stream processing
- Recovery semantics and guarantees
- Exactly-once processing in Apache Beam / Google Cloud Dataflow

State in dataflow computations

Any non-trivial streaming computation maintains **state**:

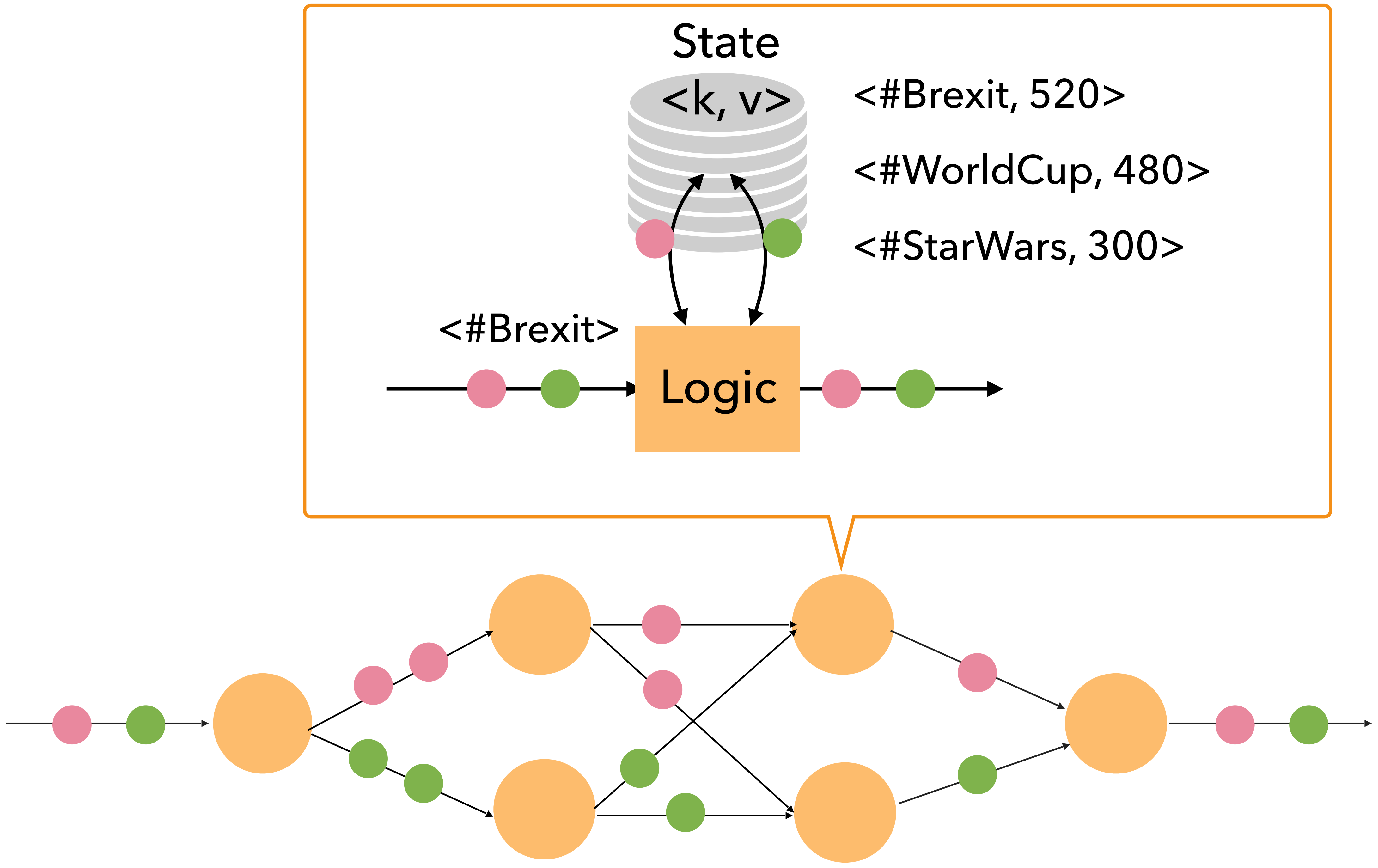
- rolling aggregations
- window contents
- input offsets
- machine learning models



State in dataflow computations

Any non-trivial streaming computation maintains **state**:

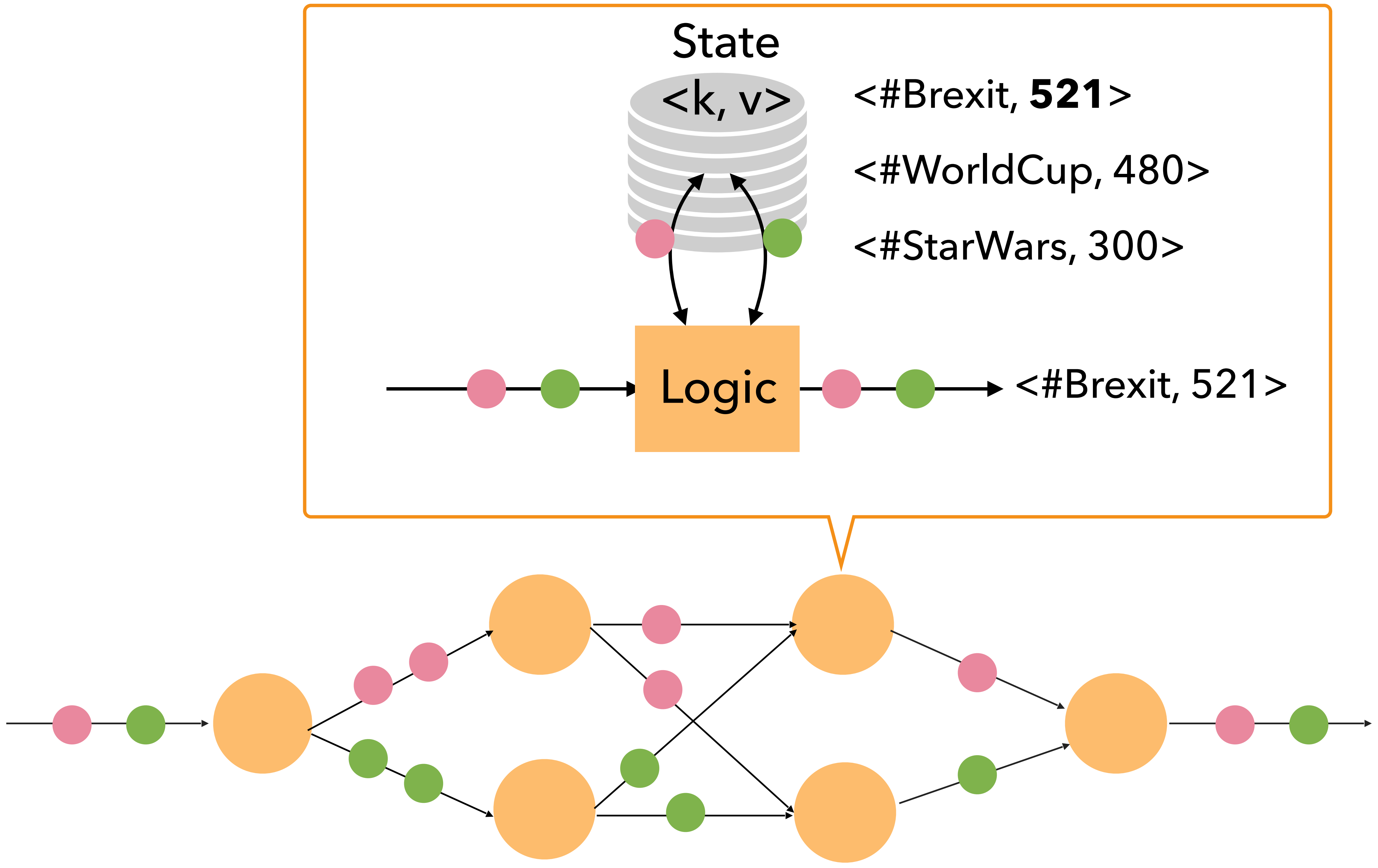
- rolling aggregations
- window contents
- input offsets
- machine learning models



State in dataflow computations

Any non-trivial streaming computation maintains **state**:

- rolling aggregations
- window contents
- input offsets
- machine learning models



https://www.ververica.com/blog/blink-flink-alibaba-search

Alibaba Group

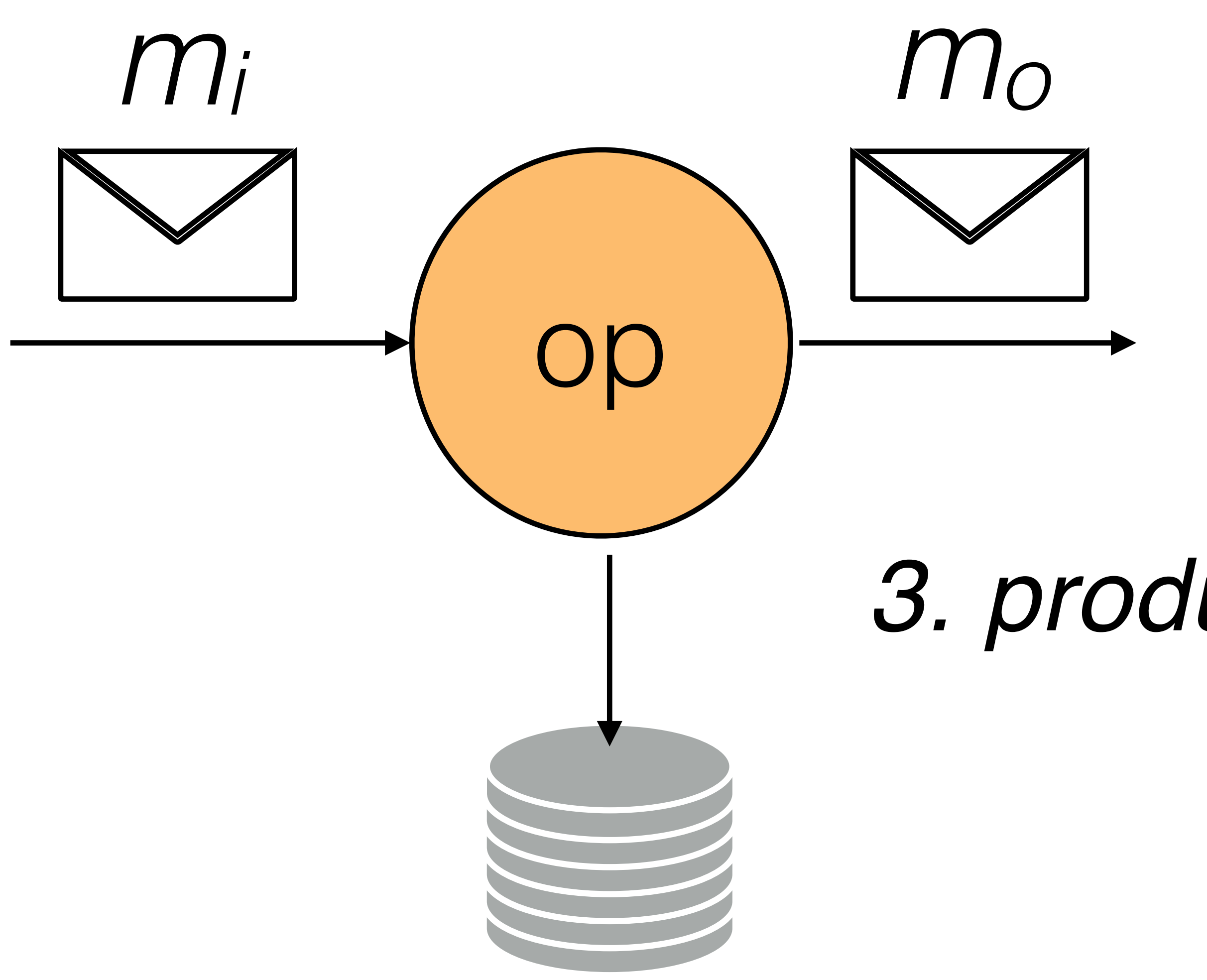
Blink is a forked version of Flink that we have been maintaining to fit some of the unique requirements we have at Alibaba. At this point, Blink is running on a few different clusters, and each cluster has about 1000 machines, so large-scale performance is very important to us.

Distributed streaming systems *will* fail

- how can we guard state against failures and guarantee correct results after recovery?
- how can we ensure minimal downtime and fast recovery?
- how can we hide recovery side-effects from downstream applications?

What is a failure?

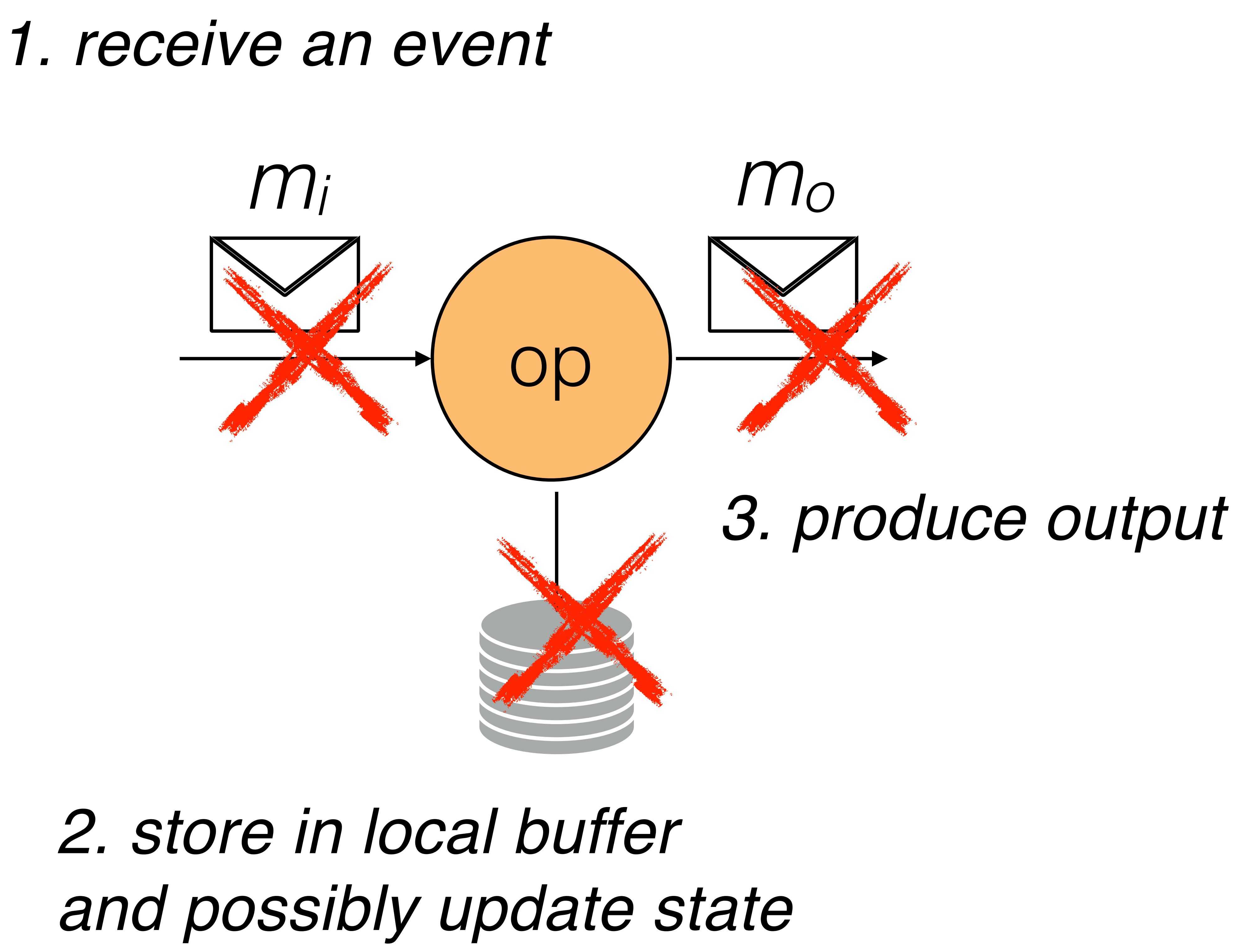
1. receive an event



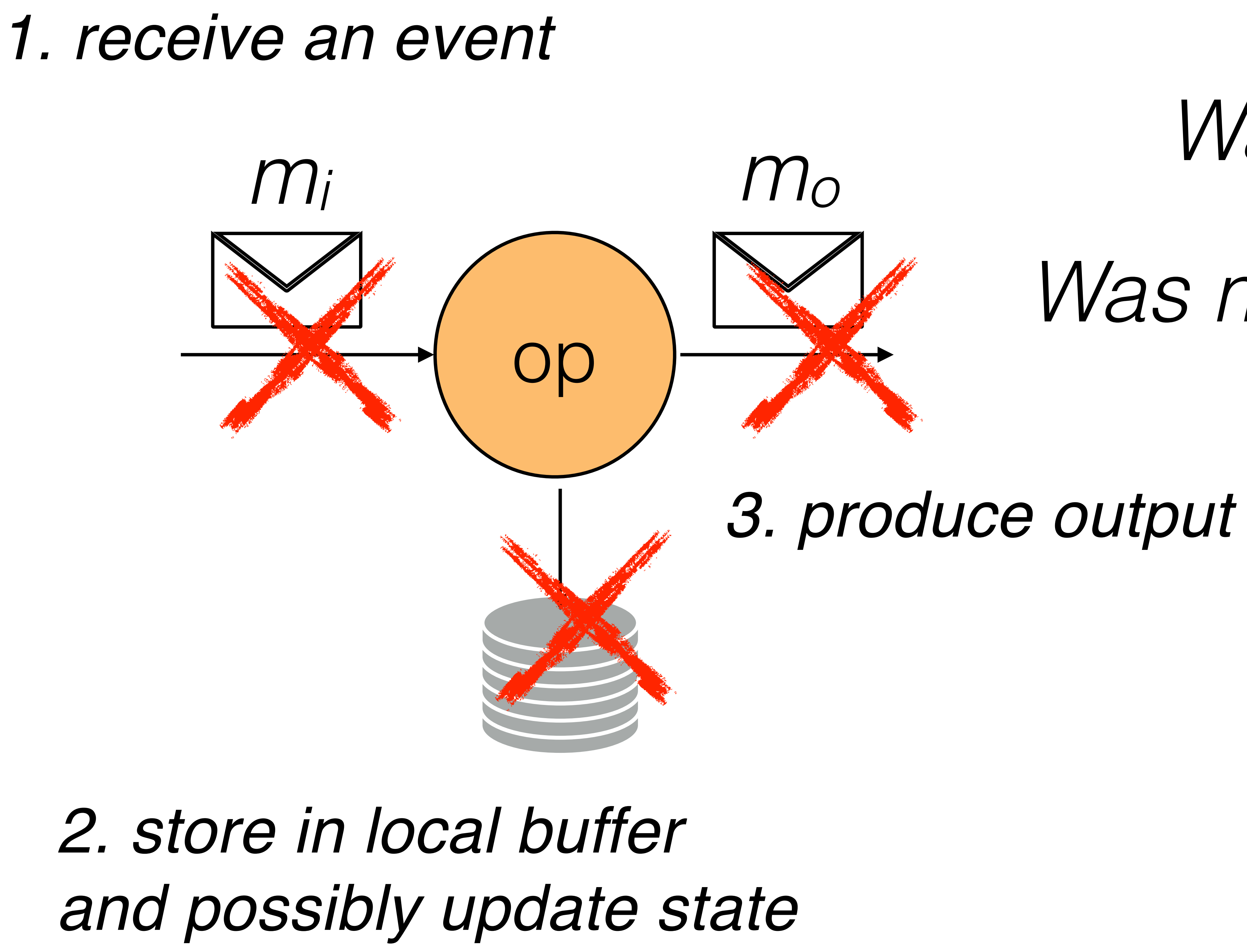
3. produce output

2. store in local buffer
and possibly update state

What is a failure?



What is a failure?

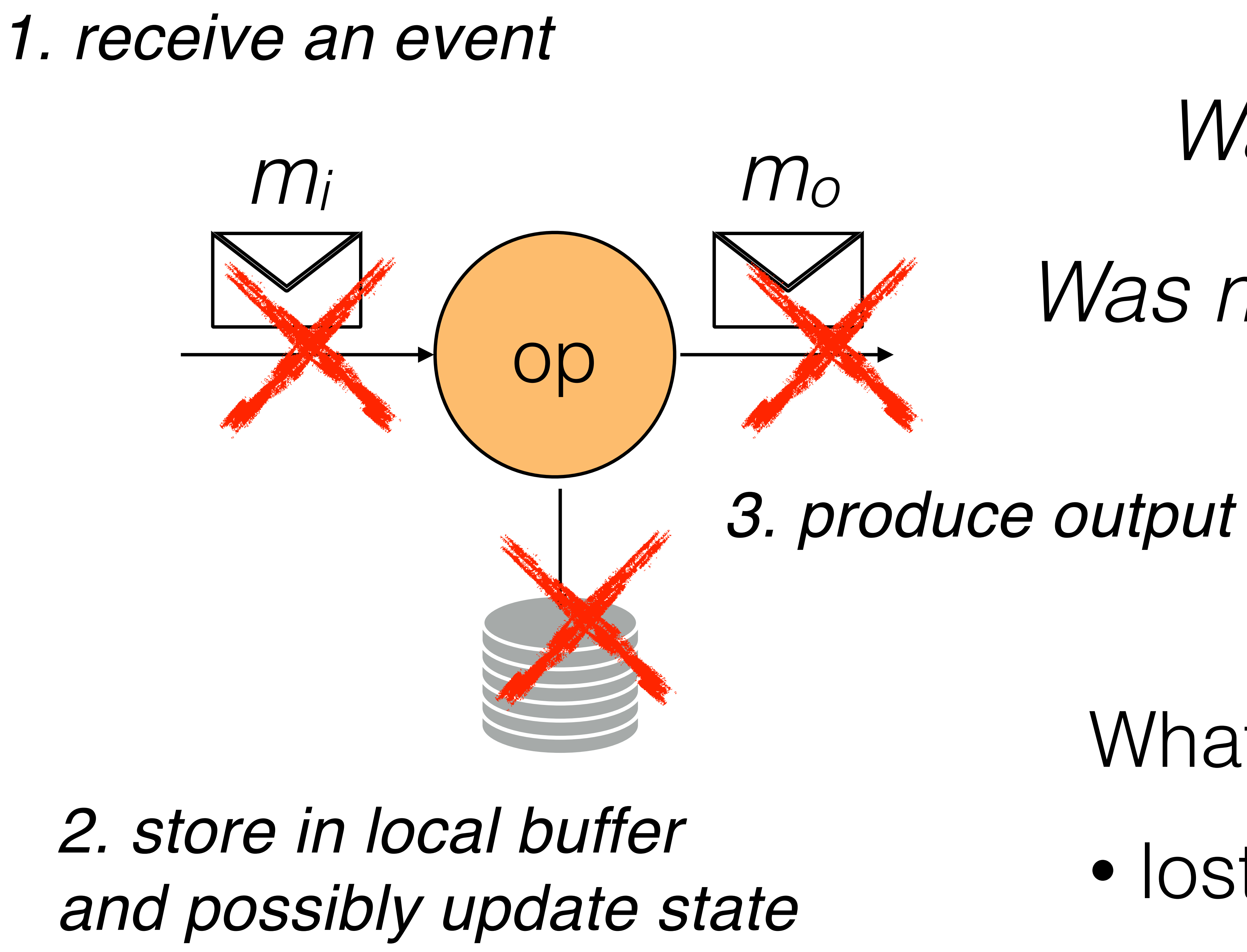


Was m_i fully processed?

Was m_o delivered downstream?

Vasiliki Kalavri | Boston University 2020

What is a failure?



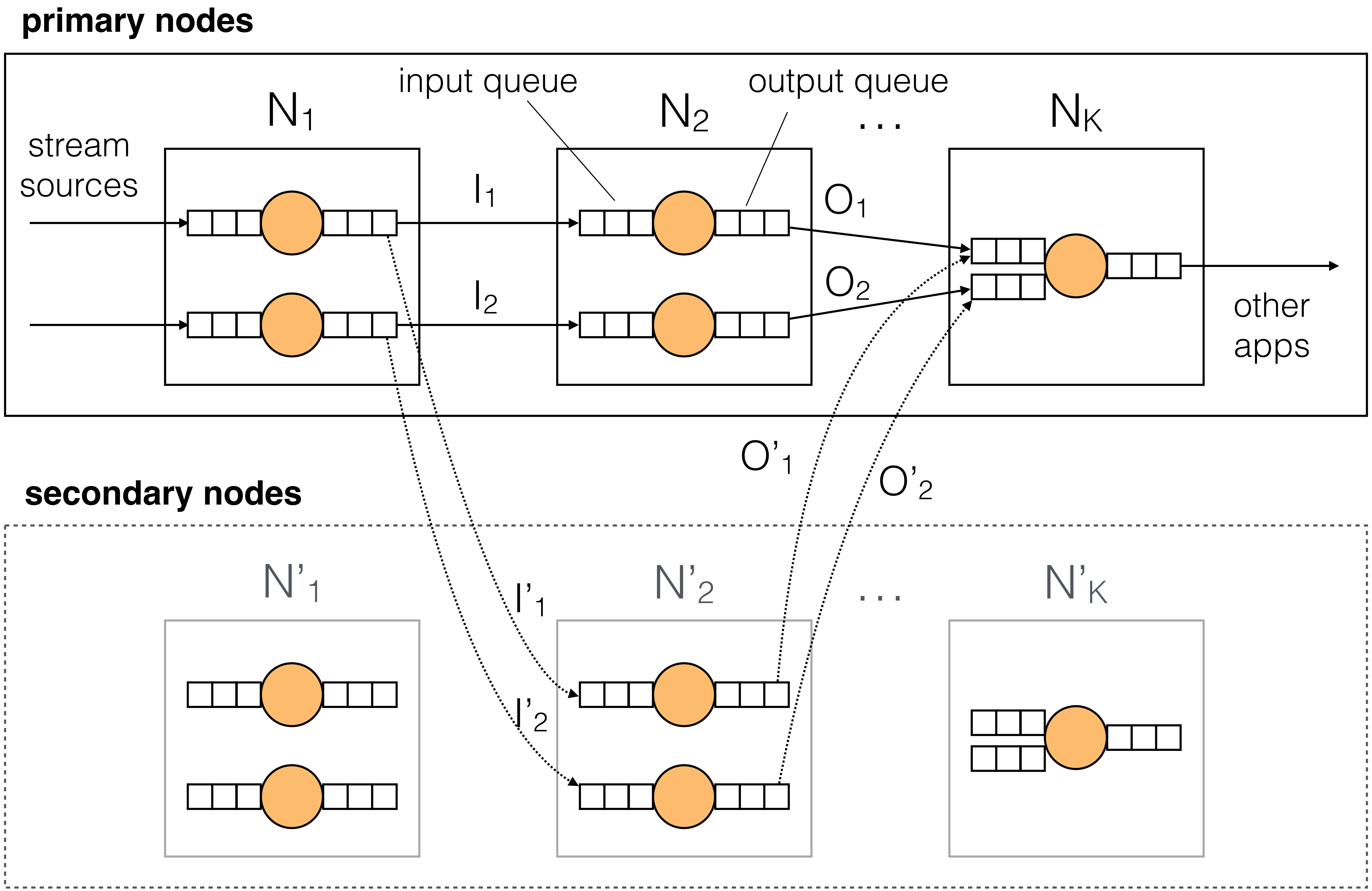
Was m_i fully processed?

Was m_o delivered downstream?

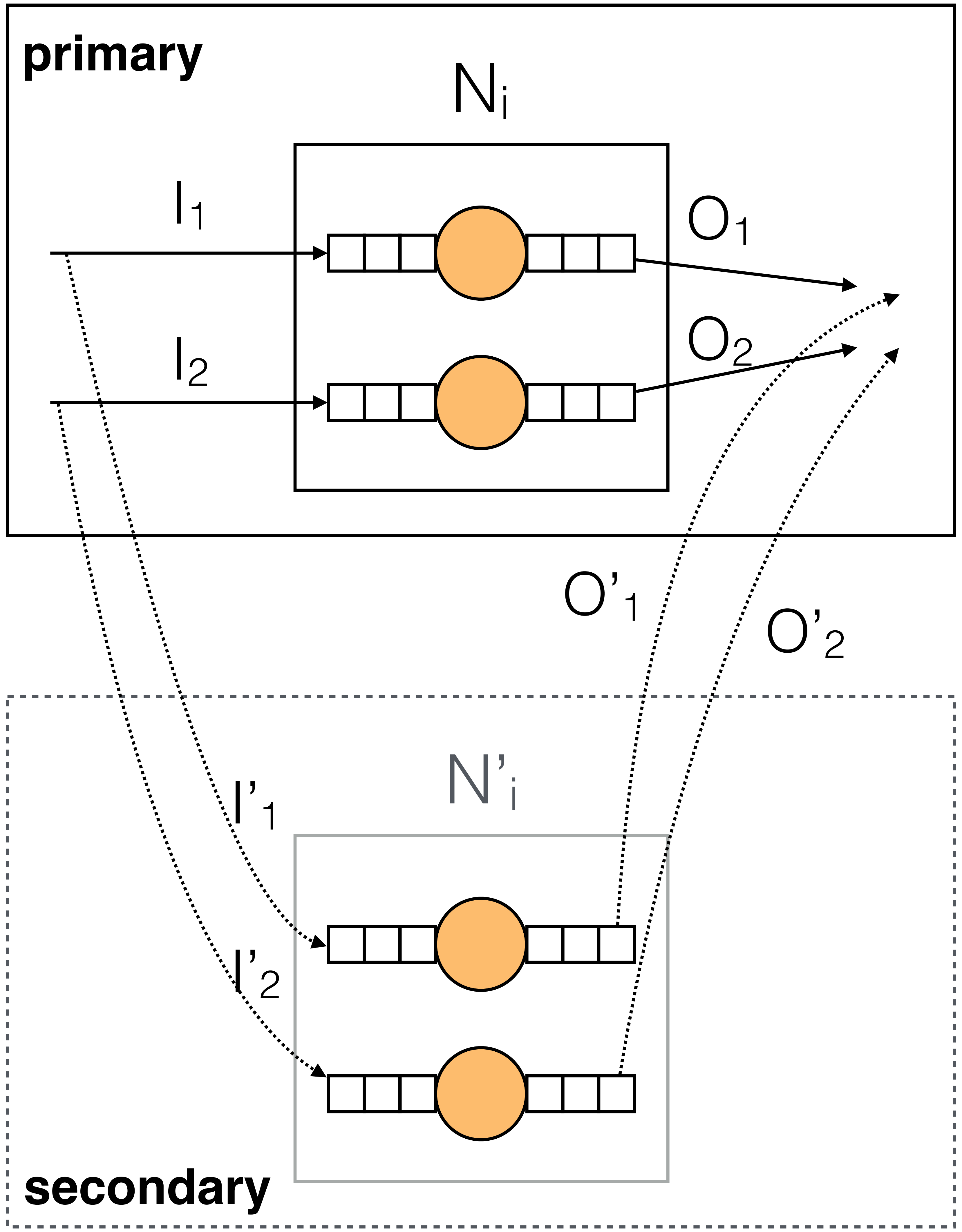
What can go wrong:

- lost events
- duplicate or lost state updates
- wrong result

A simple system model



Assumptions



- The communication network ensures order-preserving, reliable message transport, e.g. TCP.
- Failures are single-node and fail-stop, i.e. no network partitions or multiple simultaneous failures are considered.
- The secondary node uses keep-alive requests to detect primary failures.

Recovery types

Recovery types

- **Precise** recovery (**exactly-once**)
  - It hides the effects of a failure perfectly
  - Post-failure output is identical to no-failure

Recovery types

- **Precise** recovery (**exactly-once**)
  - It hides the effects of a failure perfectly
  - Post-failure output is identical to no-failure
- **Rollback** recovery (**at-least-once**)
  - It avoids information loss
  - The output may contain duplicates
  - A backup needs to rebuild state of the failed node

Recovery types

- **Precise** recovery (**exactly-once**)
  - It hides the effects of a failure perfectly
  - Post-failure output is identical to no-failure
- **Rollback** recovery (**at-least-once**)
  - It avoids information loss
  - The output may contain duplicates
  - A backup needs to rebuild state of the failed node
- **Gap** recovery (**at-most-once**)
  - It drops data during failure
  - The backup starts from most recent information

Recovery semantics

Given a dataflow *Q*, let O_e be the output stream produced by input e. In the event of a failure, let O_f be the pre-failure execution of the primary and *O'* the output produced by the secondary after recovery.

- **Precise** recovery guarantees $O_f + O' = O_e$
- **Rollback** recovery allows duplicate tuples downstream:
  - **repeating**: duplicate tuples are identical to those produced by the primary
  - **convergent**: duplicate tuples are different but eliminating them leads to output identical to an output without failure
  - **divergent**: duplicate tuples are different and eliminating them produces different output

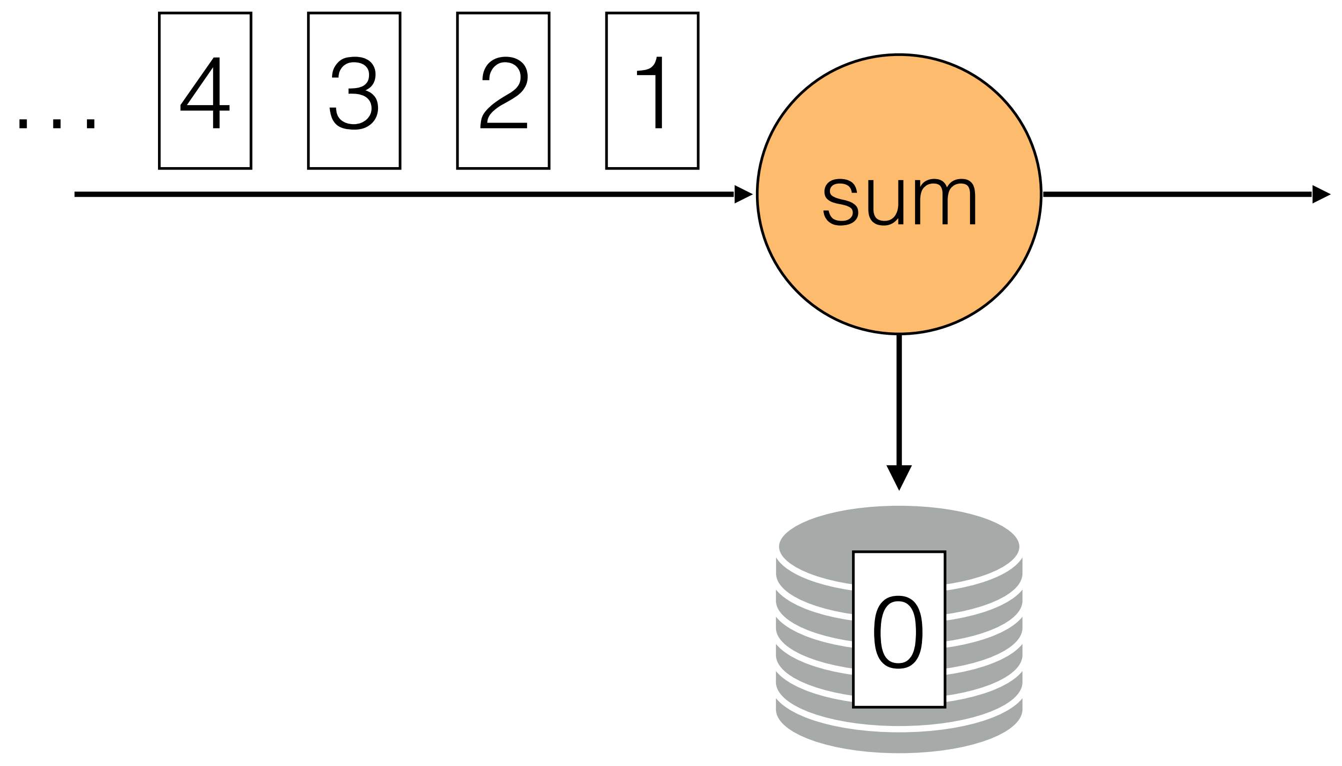
Outputs after recovery

Recovery type	Before failure	After failure
Precise	t_1 t_2 t_3	t_4 t_5 t_6 ...
Gap	t_1 t_2 t_3	t_5 t_6 ...
Rollback-repeating	t_1 t_2 t_3	$\mathbf{t_2}$ $\mathbf{t_3}$ t_4 ...
Rollback-convergent	t_1 t_2 t_3	$\mathbf{t'_2}$ $\mathbf{t'_3}$ t_4 ...
Rollback-divergent	t_1 t_2 t_3	$\mathbf{t'_2}$ $\mathbf{t'_3}$ $\mathbf{t'_4}$ ...

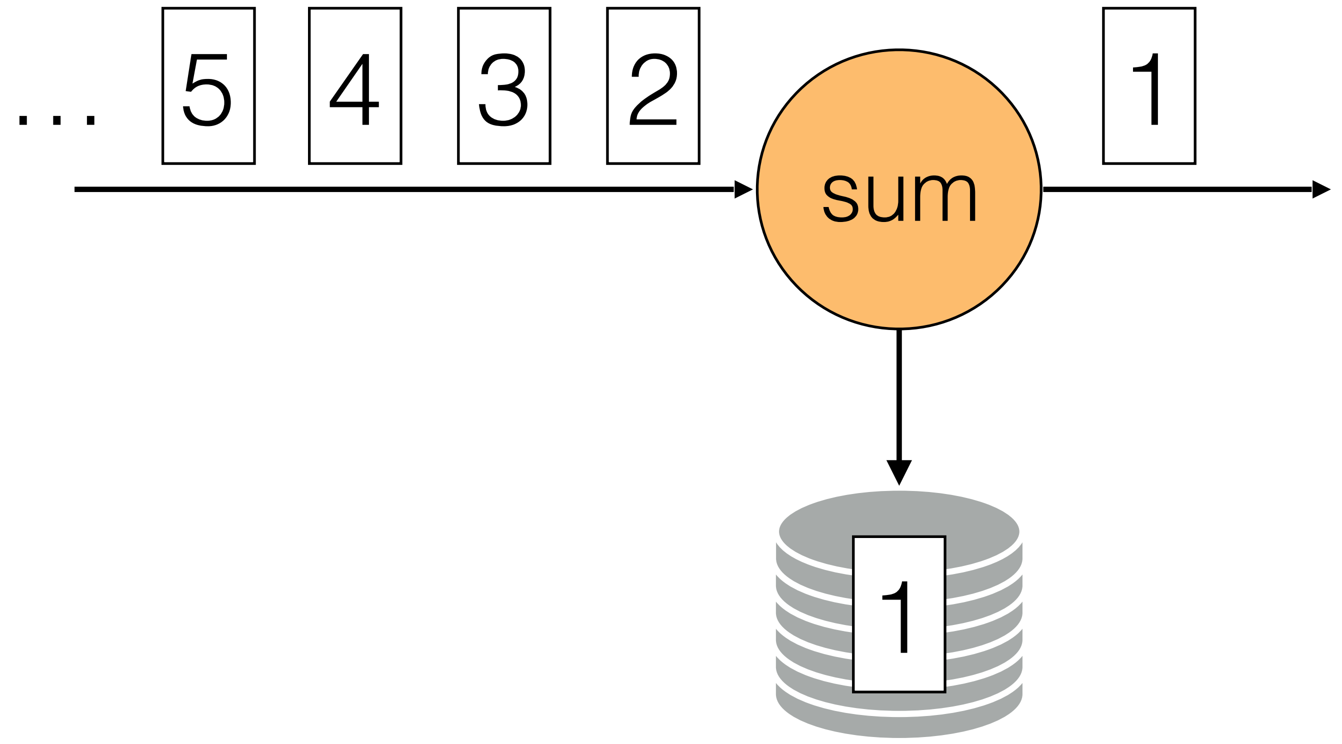
The output semantics depend on the operator type:

- **arbitrary**: it depends on order, randomness, or external system
- **deterministic**: it produces the same output when starting from the same initial state and given the same sequence of input tuples
- **convergent-capable**: it can re-build internal state in a way that it eventually converges to a non-failure execution output
- **repeatable**: it produces identical duplicate tuples

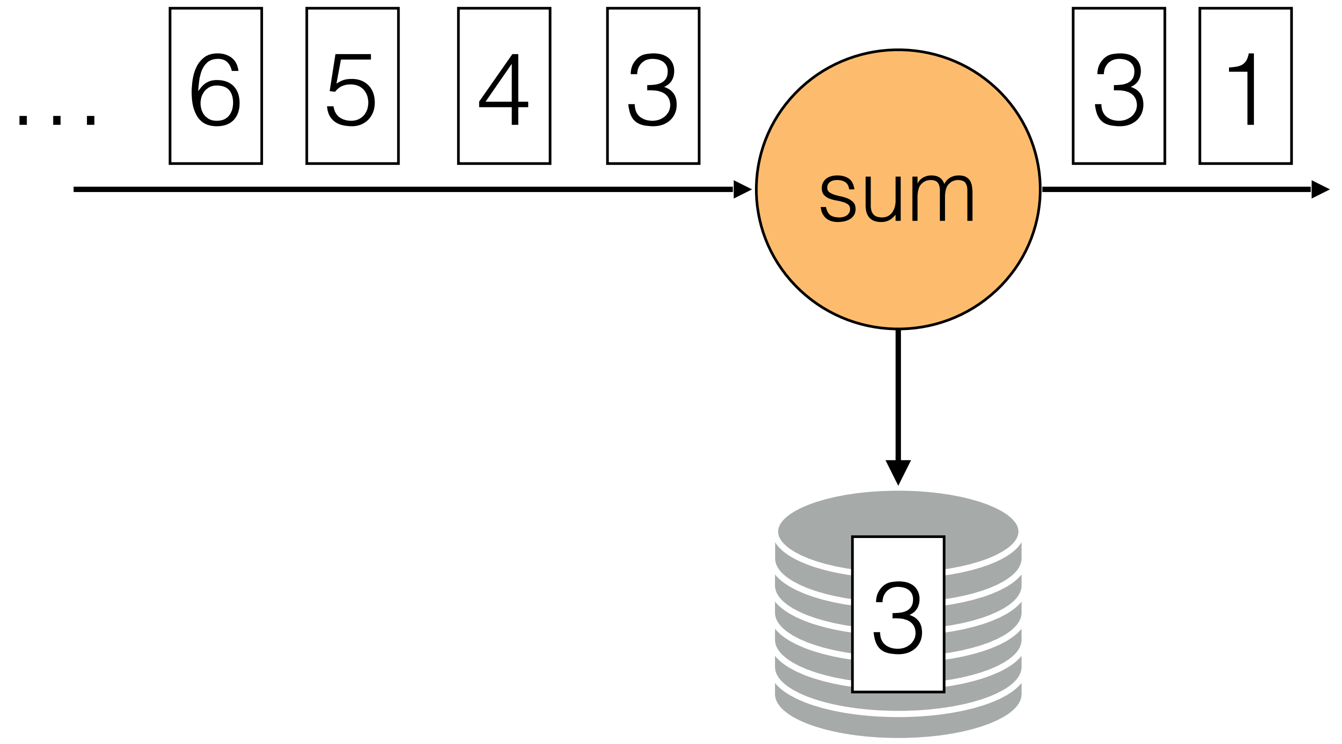
Processing guarantees and result semantics



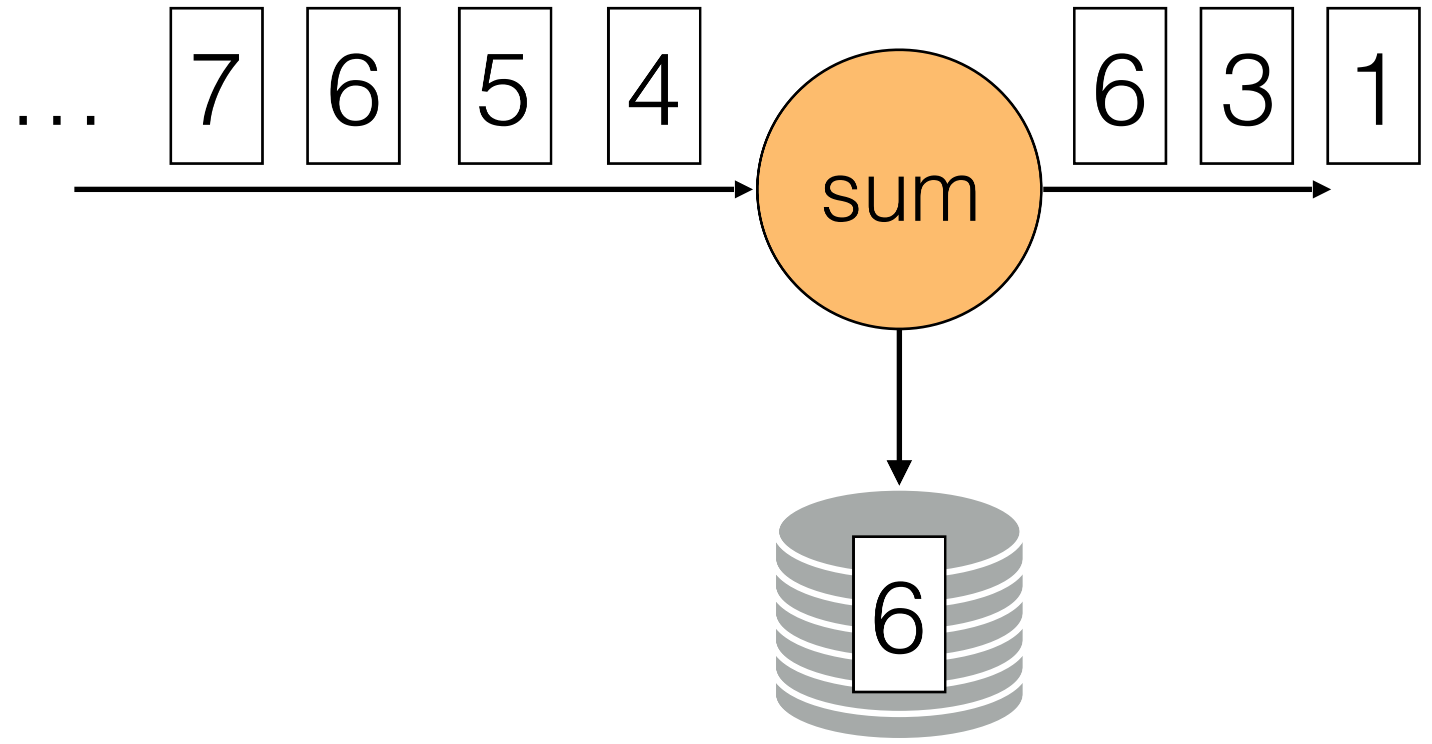
Processing guarantees and result semantics



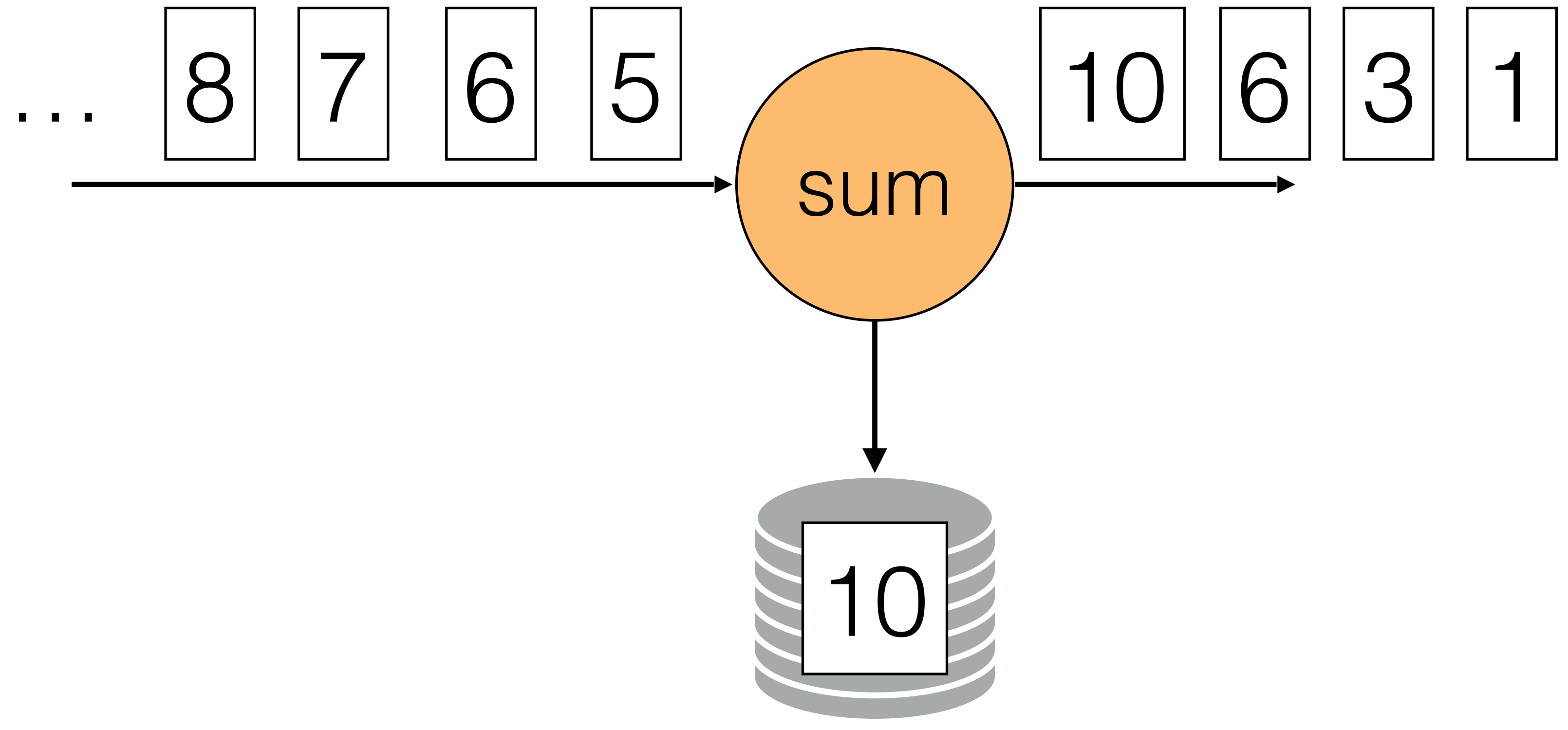
Processing guarantees and result semantics



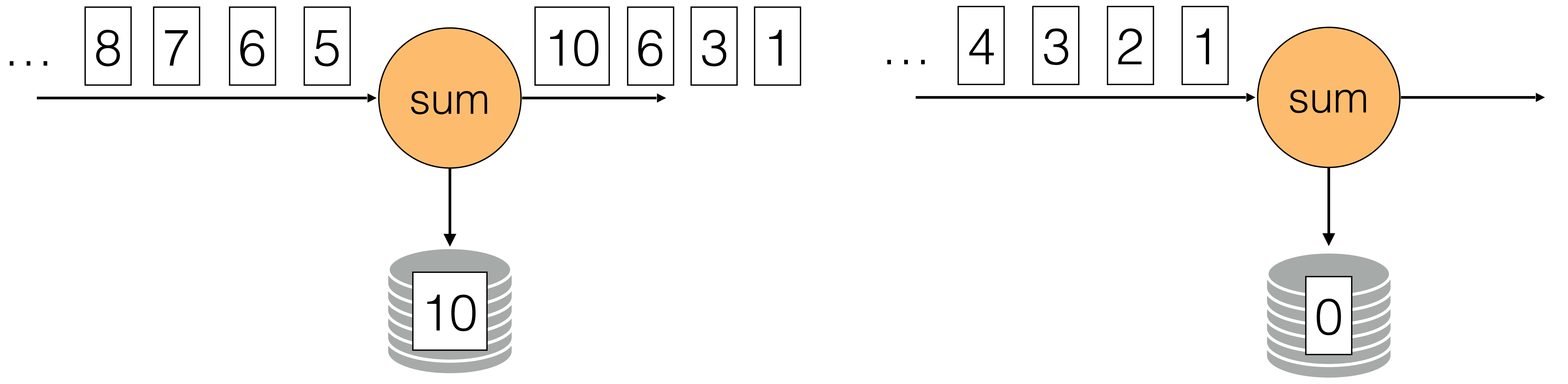
Processing guarantees and result semantics



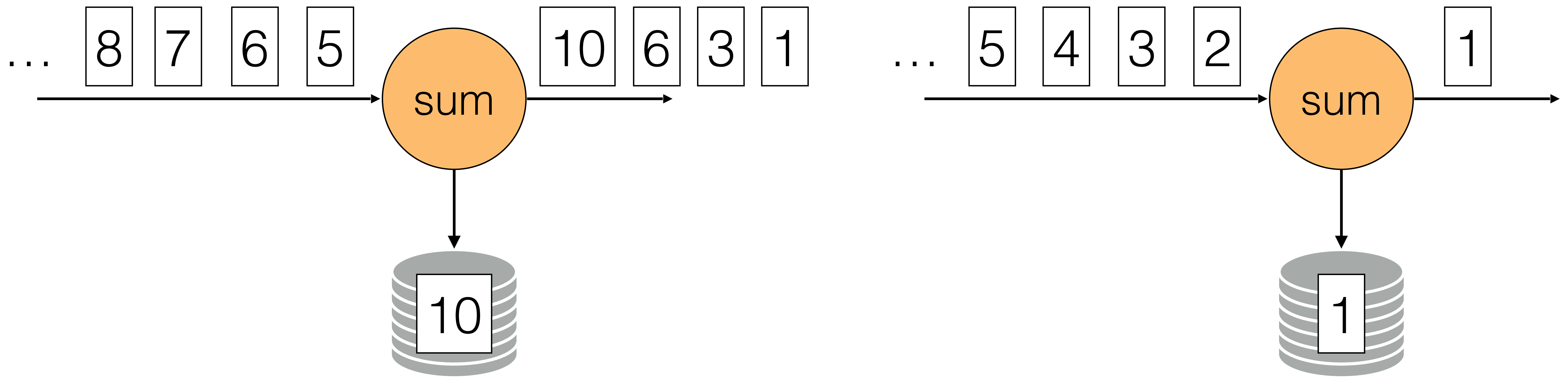
Processing guarantees and result semantics



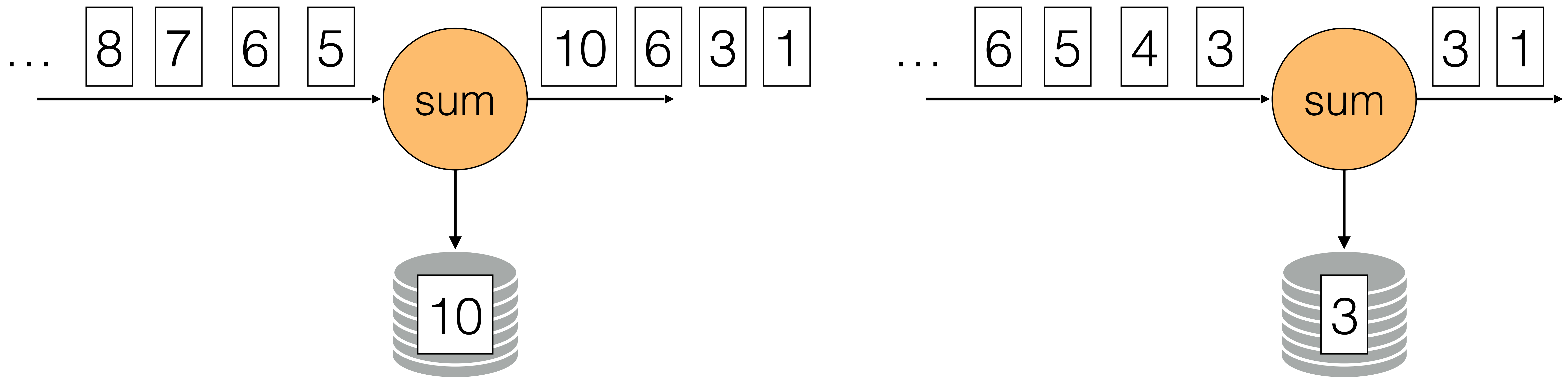
Processing guarantees and result semantics



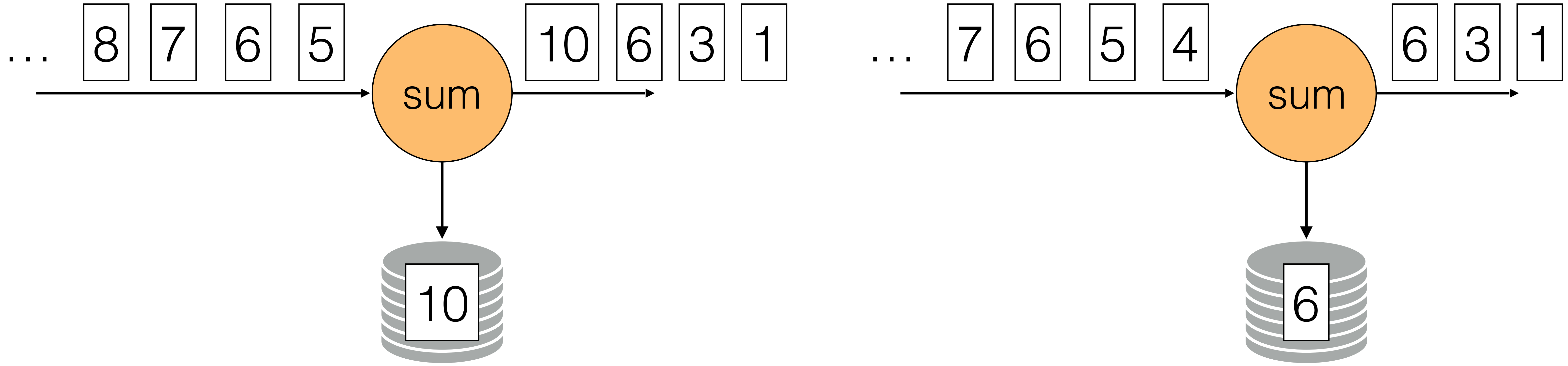
Processing guarantees and result semantics



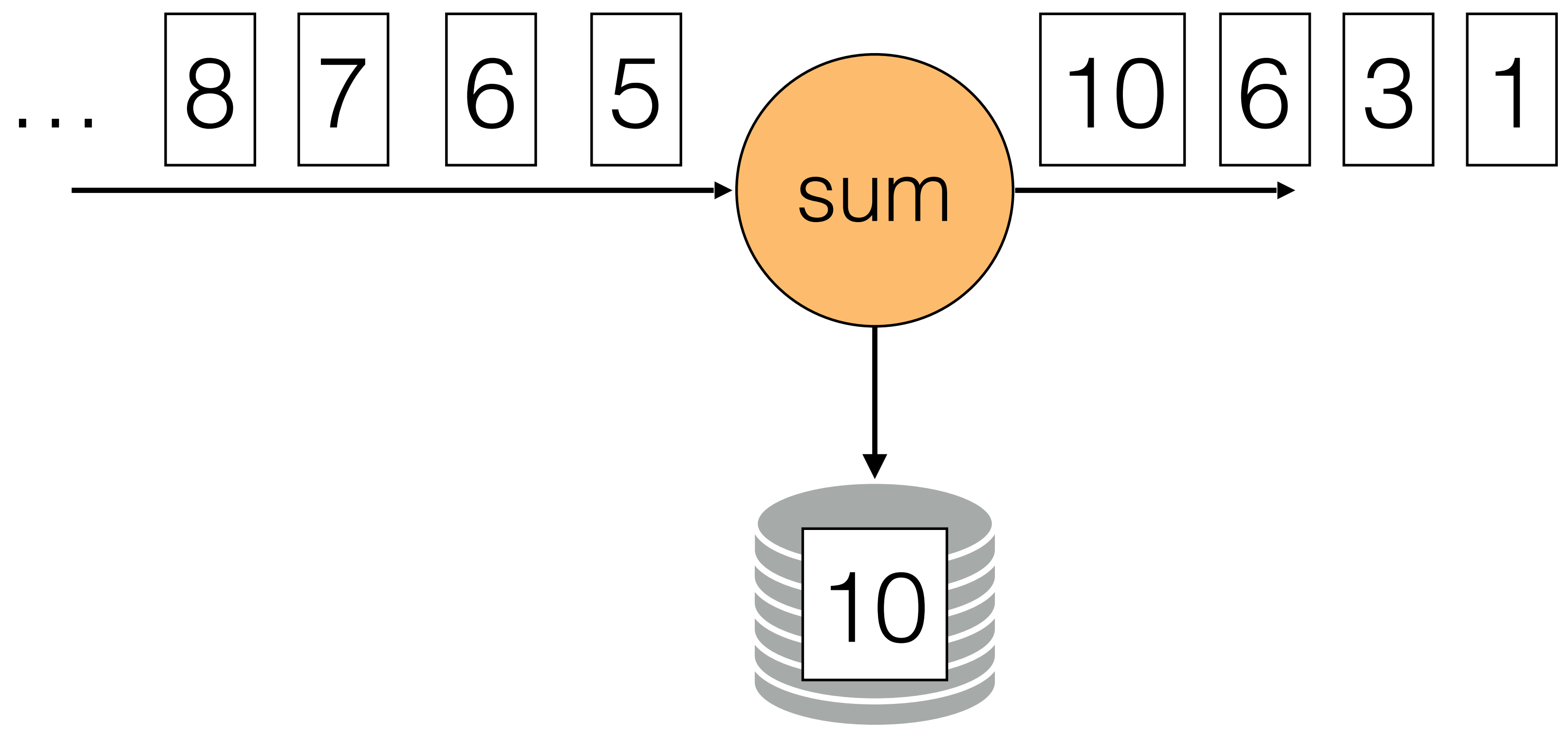
Processing guarantees and result semantics

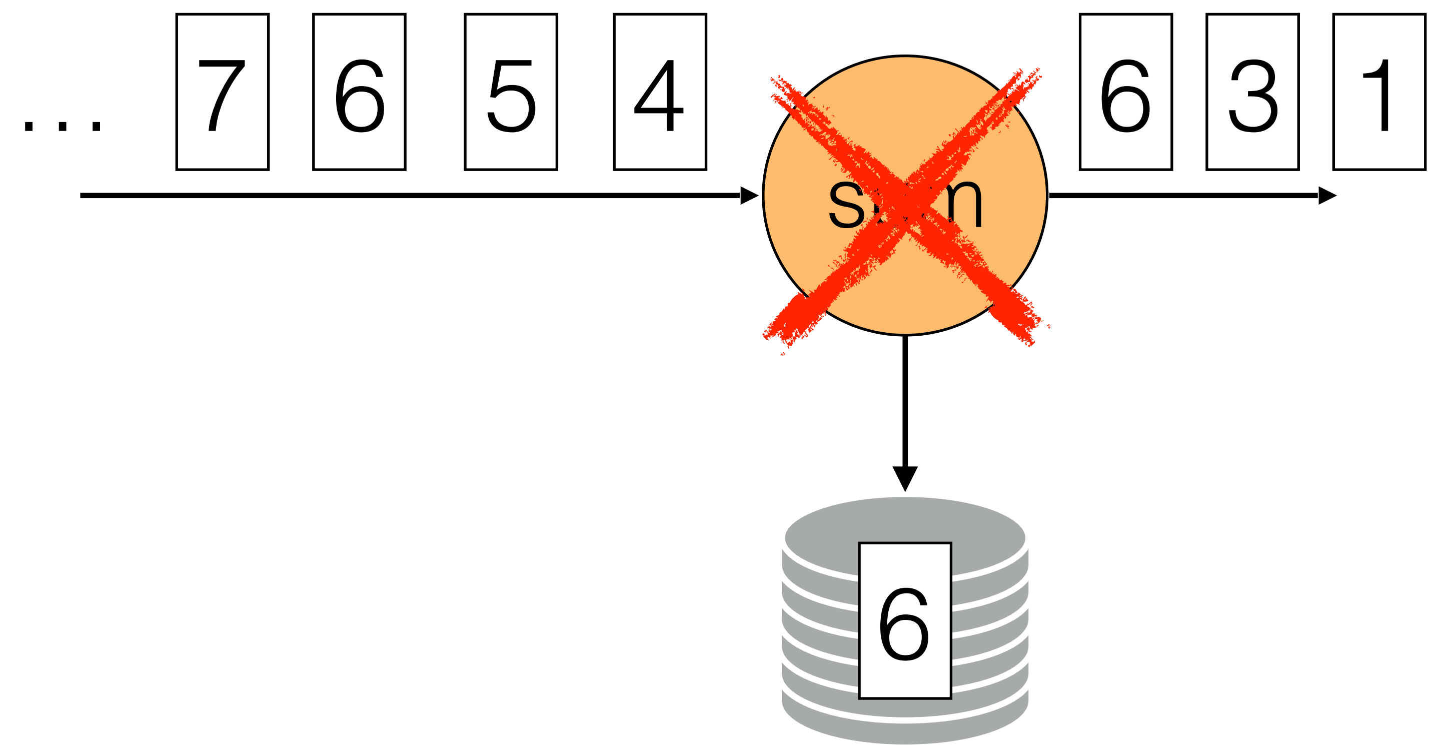


Processing guarantees and result semantics

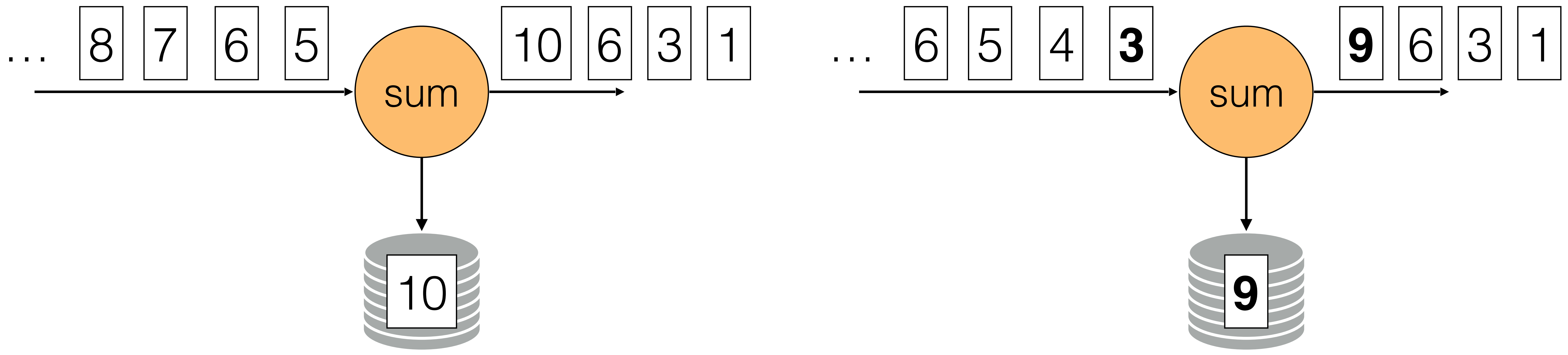


Processing guarantees and result semantics

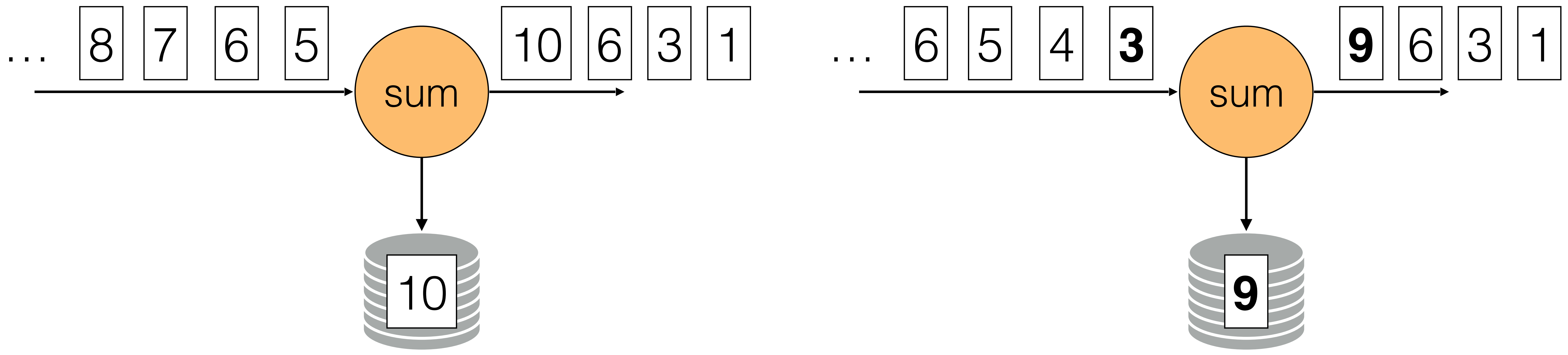


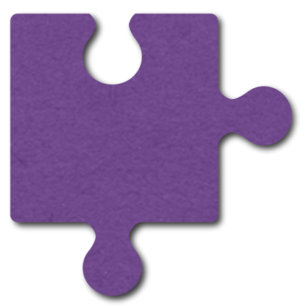


Processing guarantees and result semantics



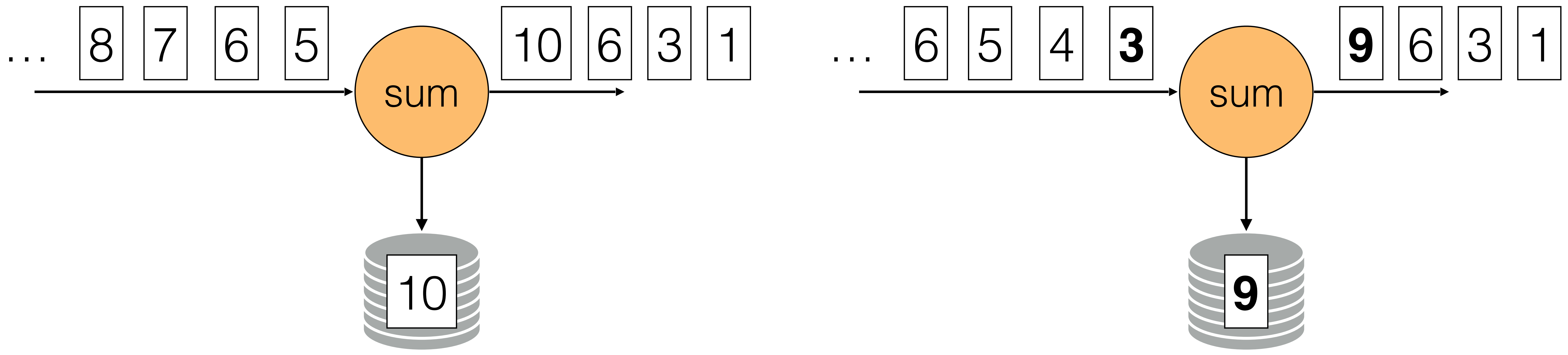
Processing guarantees and result semantics





Can you think of an operator that provides correct, possibly repeating, results even if it re-processes tuples after recovery?

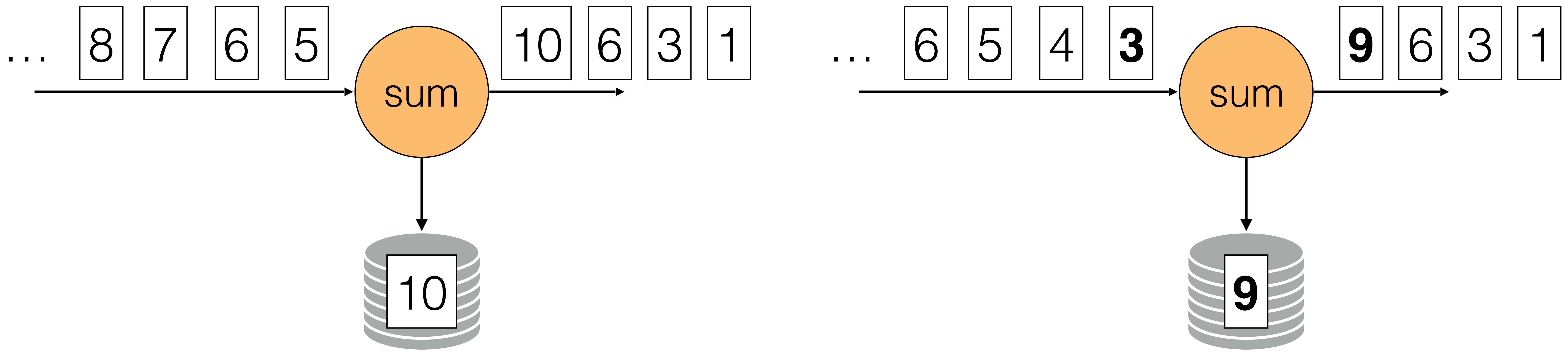
Processing guarantees and result semantics



Can you think of an operator that provides correct, possibly repeating, results even if it re-processes tuples after recovery?

Can you think of an operator that will converge to the correct result?

Processing guarantees and result semantics



Can you think of an operator that provides correct, possibly repeating, results even if it re-processes tuples after recovery?

Can you think of an operator that will converge to the correct result?

Can you think of an operator that will diverge?

Fault-tolerance trade-offs

Steady-state overhead

- How is performance affected by the fault-tolerance mechanism under normal, failure-free operation?
- How much memory or disk space is required to maintain input tuples and state?

Recovery speed

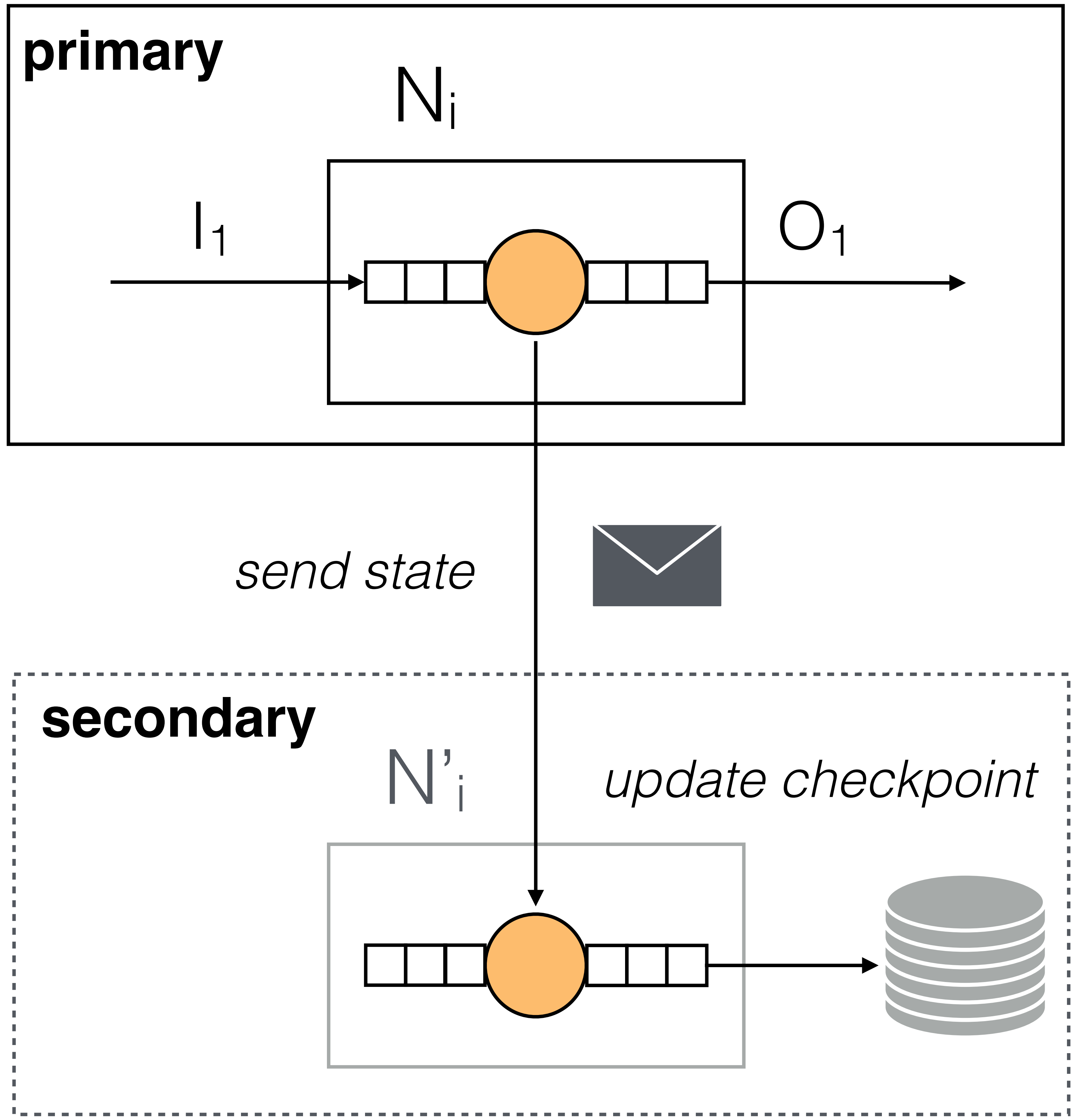
- How long does it take for the computation to catch up after a failure and recovery?
- How much input do we need to re-play? How expensive is it to re-construct the state? How fast can we de-duplicate output?

Gap Recovery

- Restart the operator from an empty state
- Drop events during recovery
- The number of lost events depends on
  - failure detection delay
  - stream input rates
  - state size
- No runtime overhead

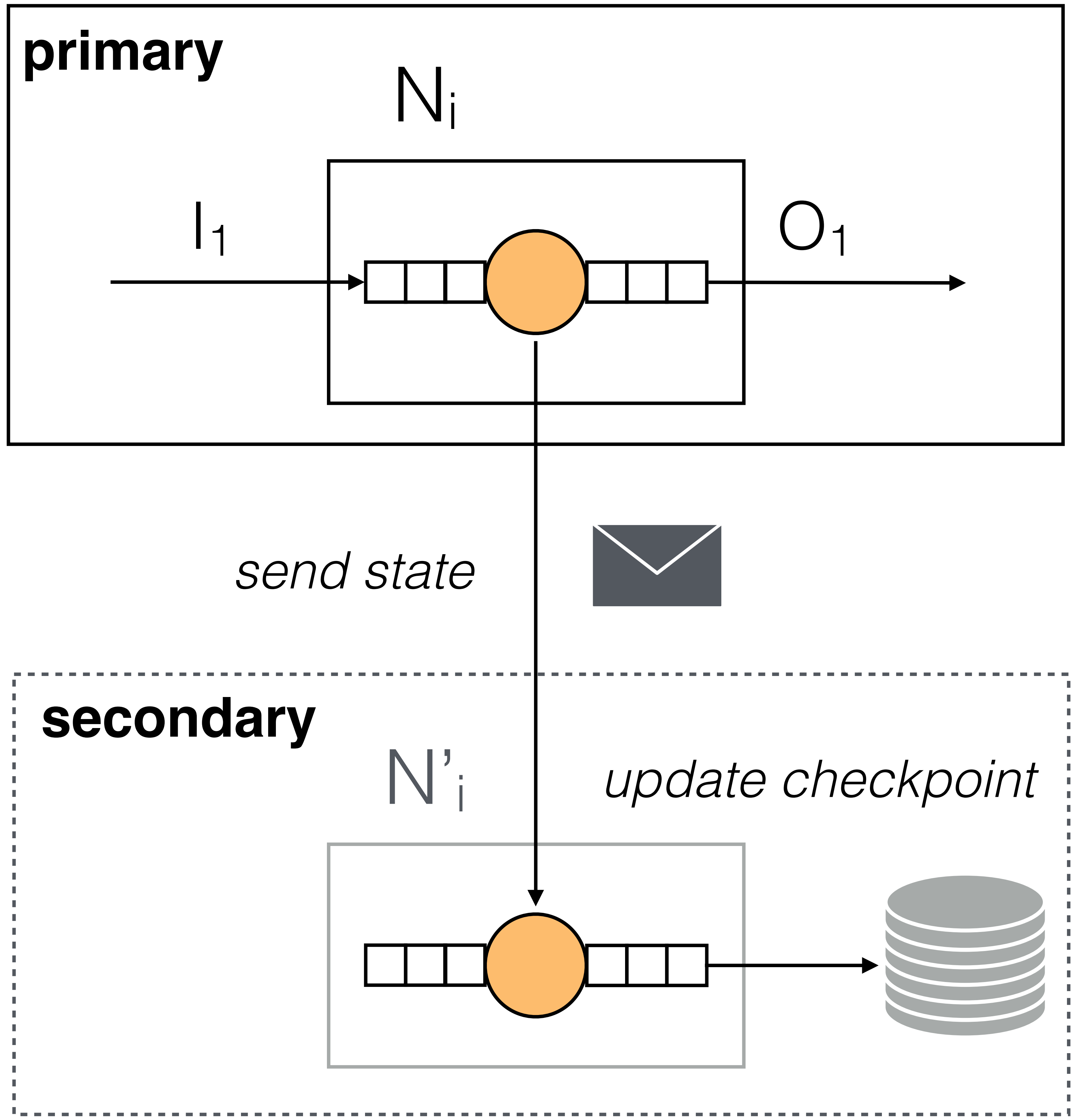
Passive Standby

- Each primary periodically **checkpoints** its state and sends it to the secondary
- The state consists of
  - input queues
  - operator state
  - output queues
- Short recovery time
- High runtime overhead
- The checkpoint interval determines the trade-off



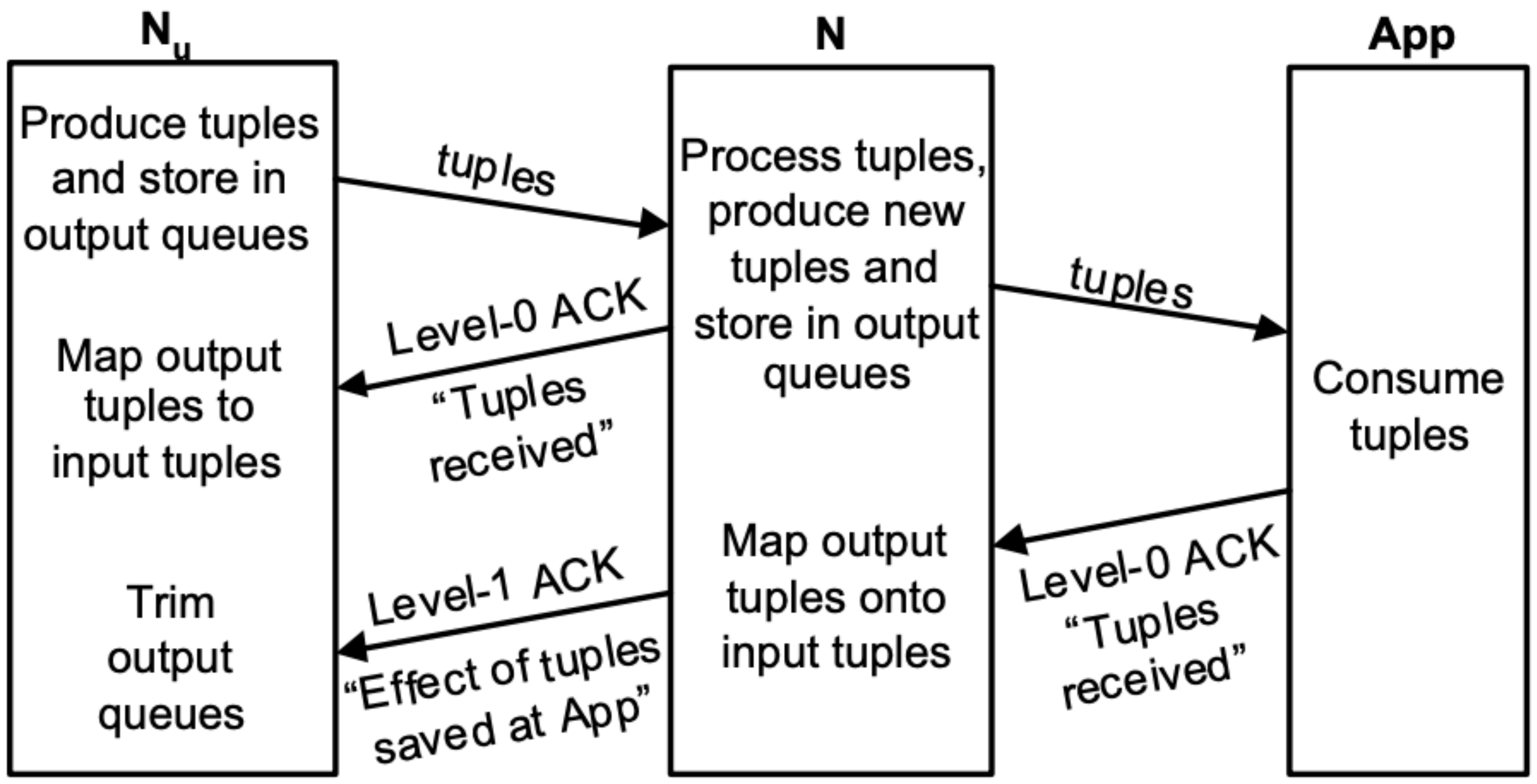
Passive Standby

- Each primary periodically **checkpoints** its state and sends it to the secondary
- The state consists of
  - input queues
  - operator state
  - output queues
- Short recovery time
- High runtime overhead
- The checkpoint interval determines the trade-off



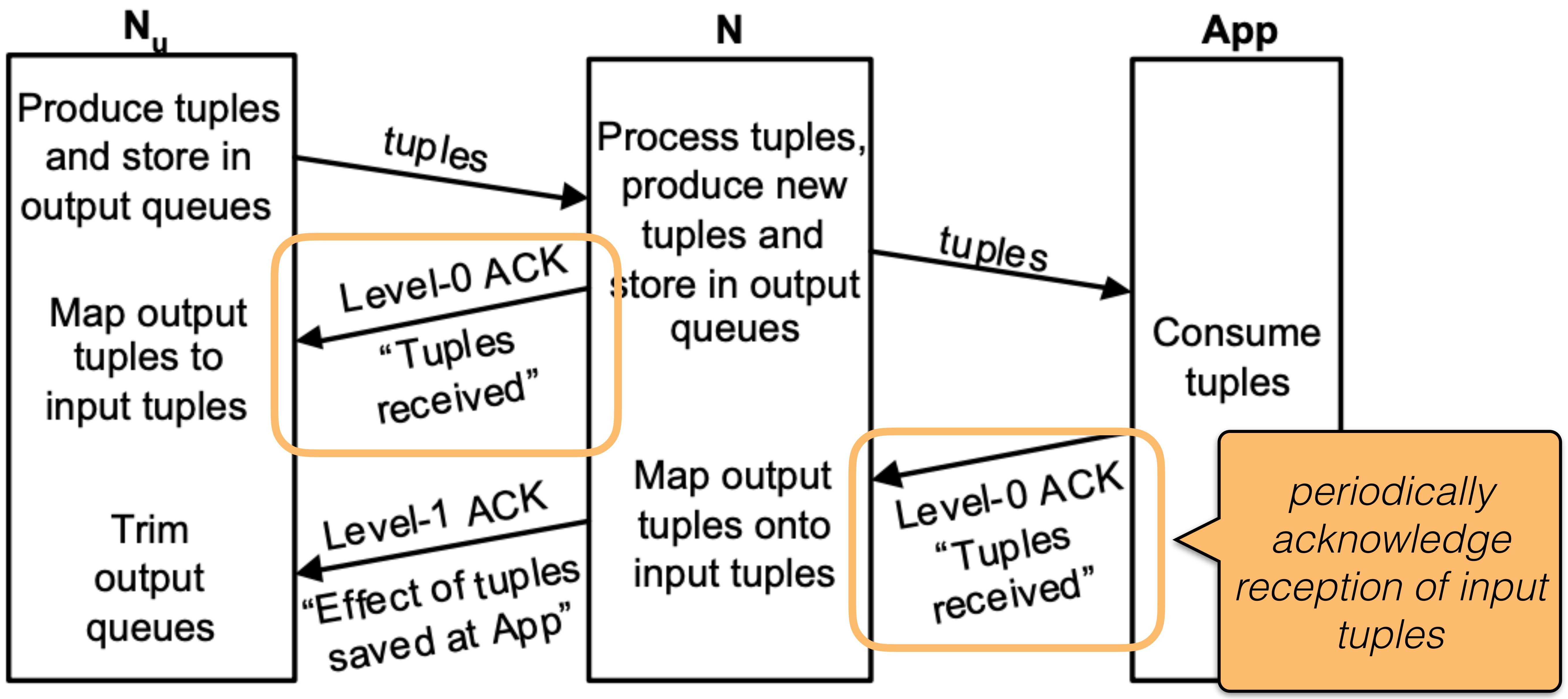
Can you see any disadvantage in this approach?

Upstream Backup



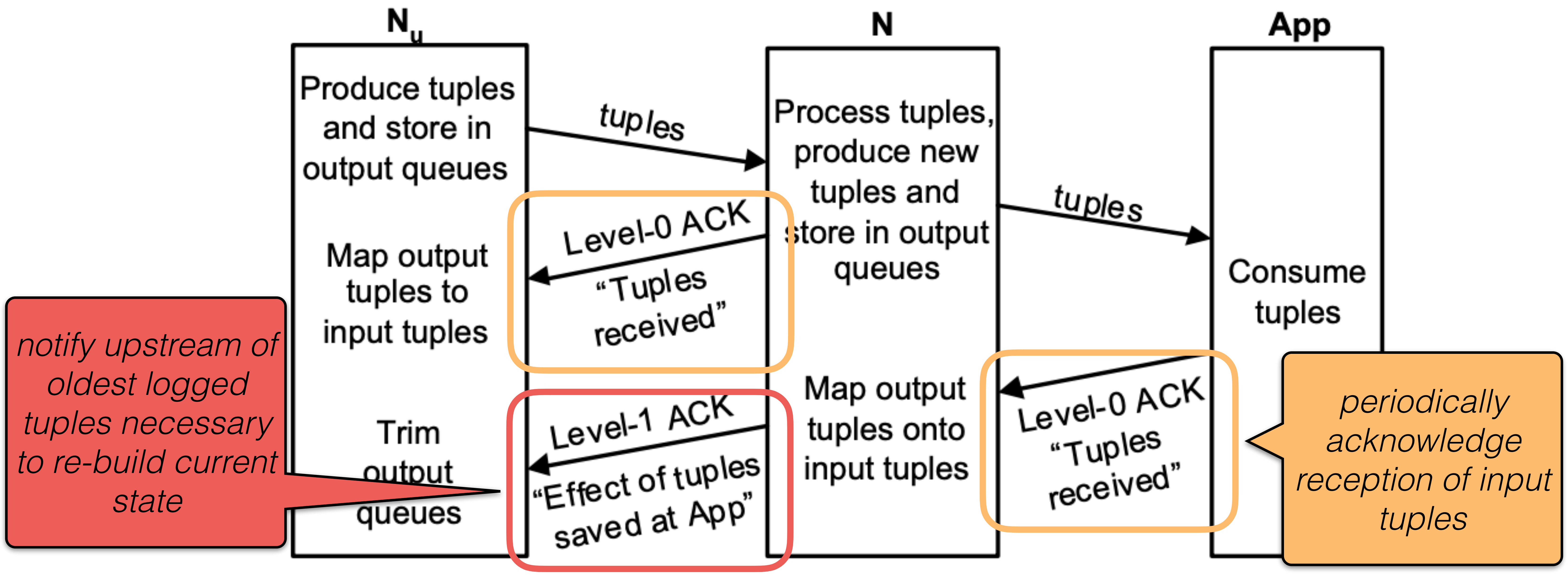
Upstream nodes act as backups for their downstream operators by logging tuples in their output queues until downstream operators have completely processed them.

Upstream Backup



Upstream nodes act as backups for their downstream operators by logging tuples in their output queues until downstream operators have completely processed them.

Upstream Backup



Upstream nodes act as backups for their downstream operators by logging tuples in their output queues until downstream operators have completely processed them.

Upstream backup

Recovery time

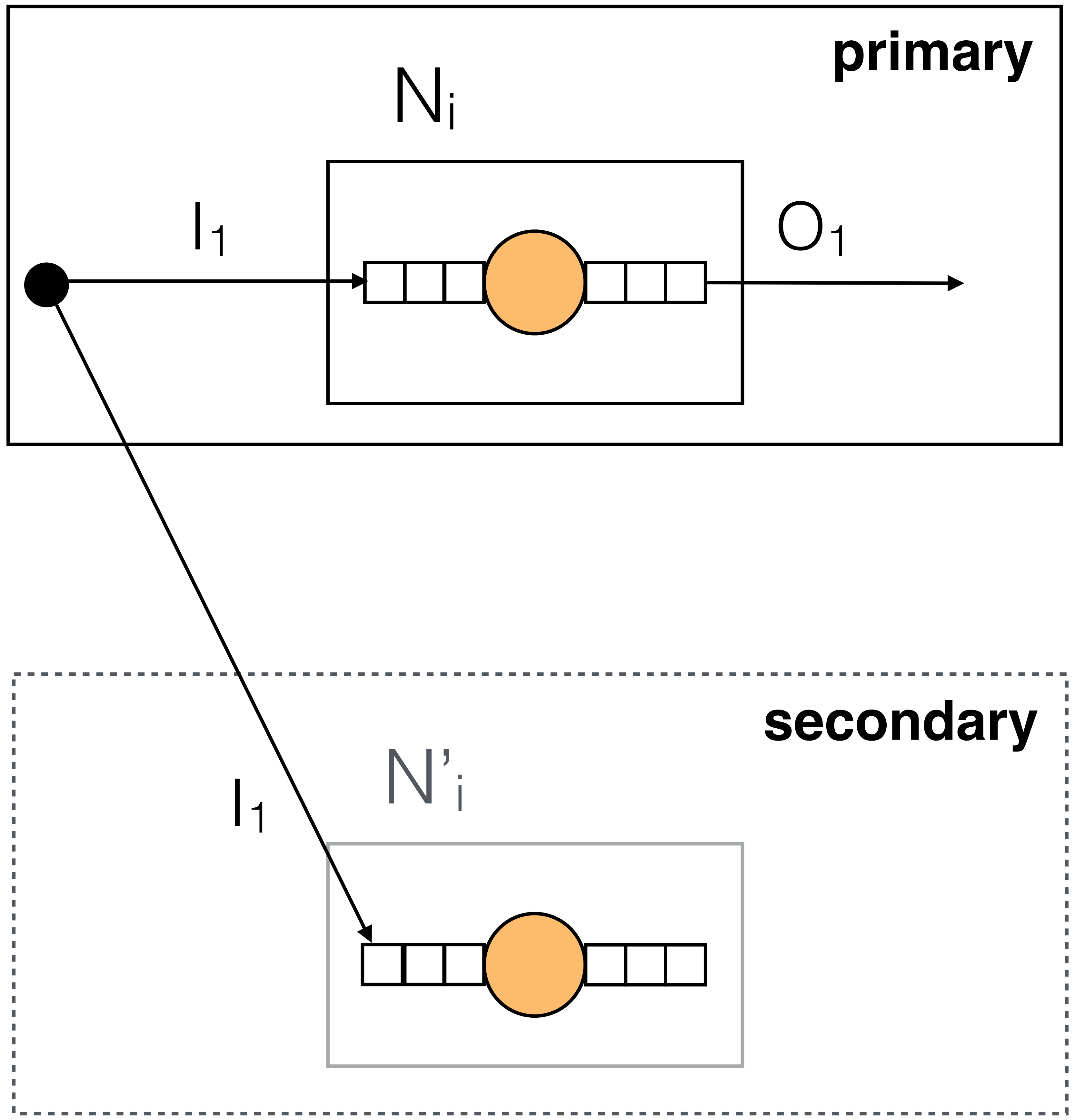
- The recovery node may need to re-process many tuples
  - all tuples that contributed to lost state
  - a complete queue-trimming interval worth of tuples, if level-0 and level-1 acks are periodically transmitted

Overhead

- Low bandwidth overhead
  - acks contain only tuple ids and are much smaller than checkpoint messages
- Low processing overhead
  - operators need to remember the oldest tuple (on each of their input streams) that contributed to the current state

Active Standby

- The secondary receives tuples from upstream and processes them **in parallel with the primary** but it doesn't output results
- Watermarks are used to **identify duplicate output tuples** and trim the secondary's output queue
- Negligible recovery time
- High overhead since **all processing is duplicated**



Precise recovery

To provide precise recovery, we need **duplicate elimination** methods:

- In passive and active standby, the failover node must ask downstream operators for the identifiers of the last tuples they received.
- In upstream backup, operators need to track and log tuple provenance / result lineage.

Can such techniques be efficiently implemented?

What if more than one nodes fail at the same time?

Exactly-once in Google Cloud Dataflow

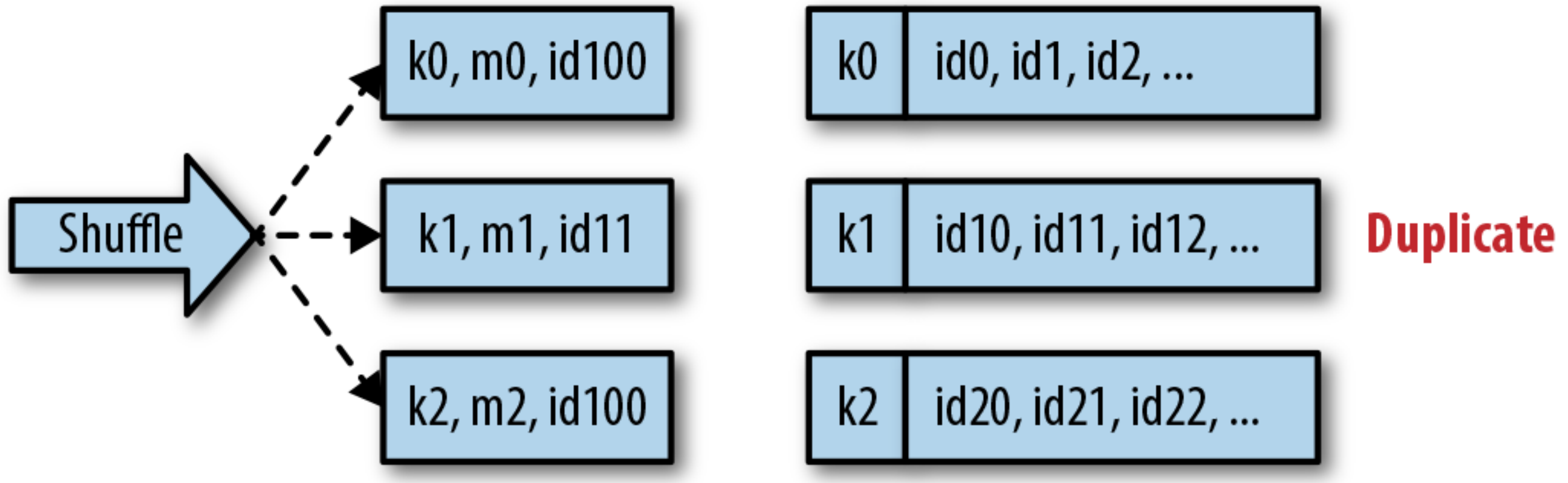
Google Dataflow uses RPC for data communication

- the sender will retry RPCs until it receives a positive ack
- the system ensures retrying even if the sender crashes
- this technique guarantees *at-least-once* delivery

RPC retries might create duplicates

- RPCs can sometimes succeed even if they appear to have failed, i.e. a sender can only trust a success status
- Dataflow tags messages with unique IDs

- Receivers store a catalog of all identifiers they have seen and processed.
- The de-duplication catalog is stored in a scalable key/value store.



http://streamingbook.net/fig/5-2

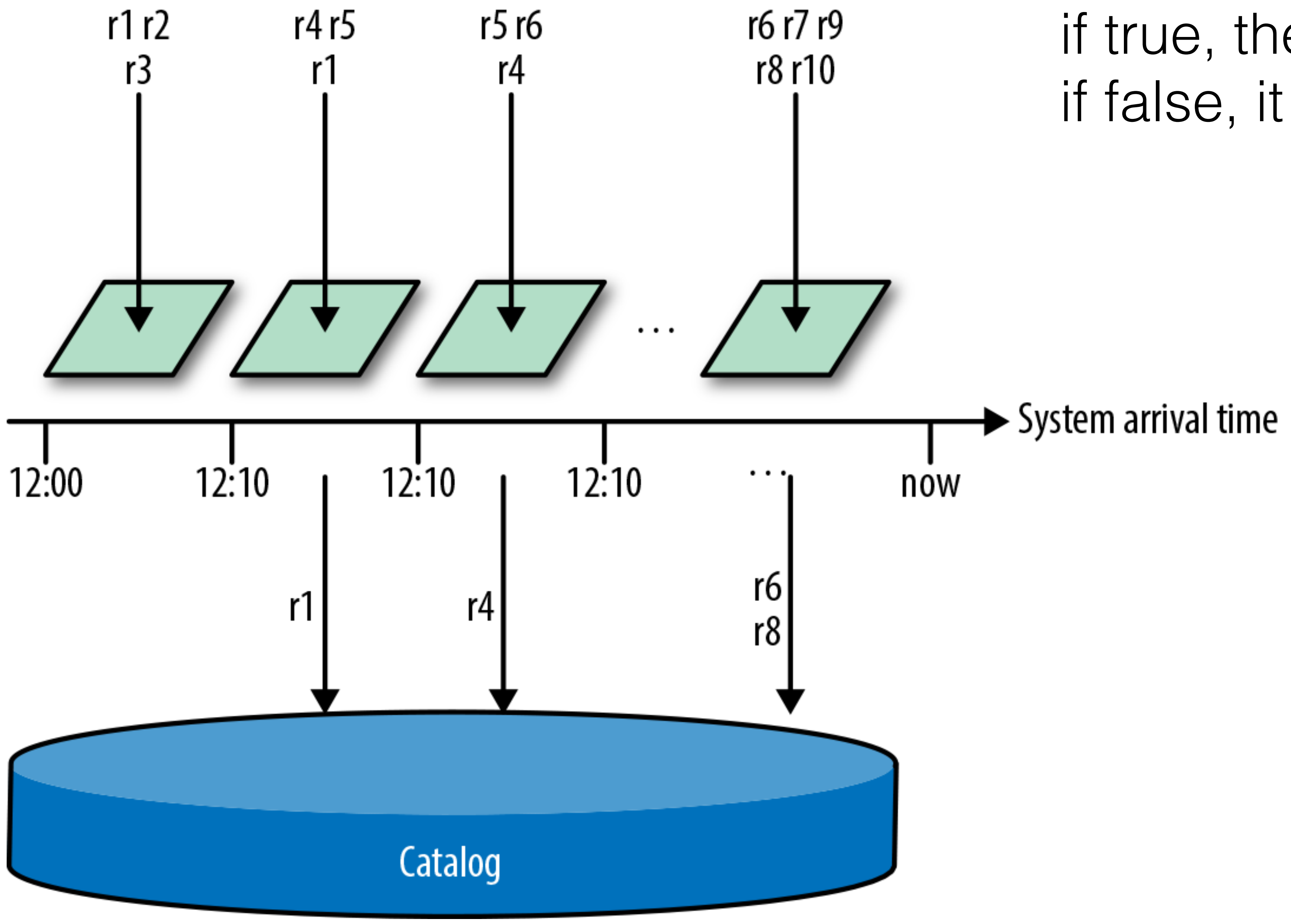
Exactly-once in Google Cloud Dataflow

Checkpointing to address non-determinism

- Each output is checkpointed together with its unique ID to stable storage *before* being delivered to the next stage
- Retries simply replay the output that has been checkpointed, i.e. the user's non-deterministic code is not re-executed

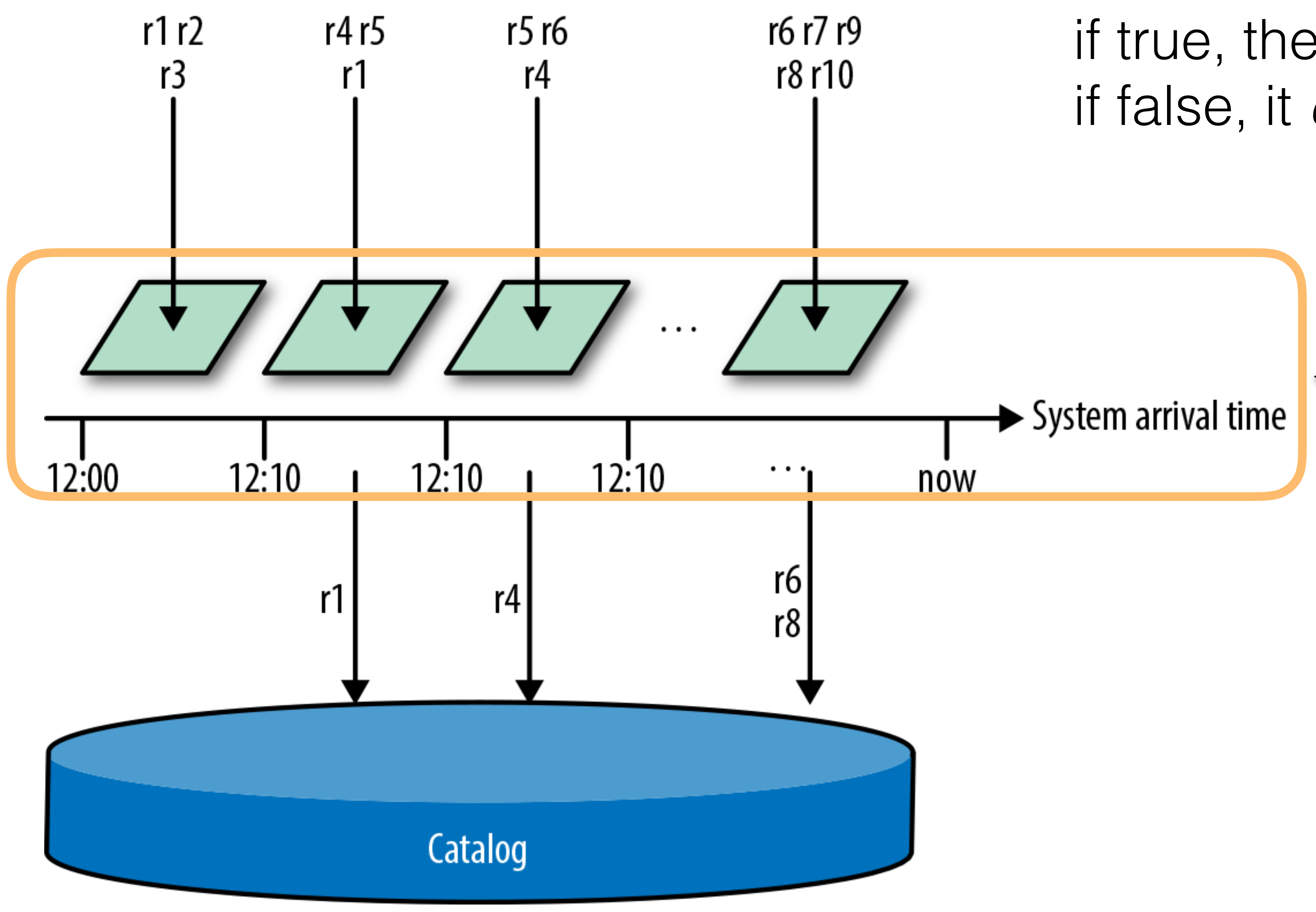
Bloom filters for performance

- Maintaining a catalog of all IDs ever seen and checking it for de-duplication is expensive
- In a *healthy* pipeline though, most records will not be duplicates
- Each worker maintains a Bloom Filter of all IDs it has seen:
  - if the filter returns false the record is not a duplicate
  - if it returns true, the worker sends a lookup to stable storage



http://streamingbook.net/fig/5-5

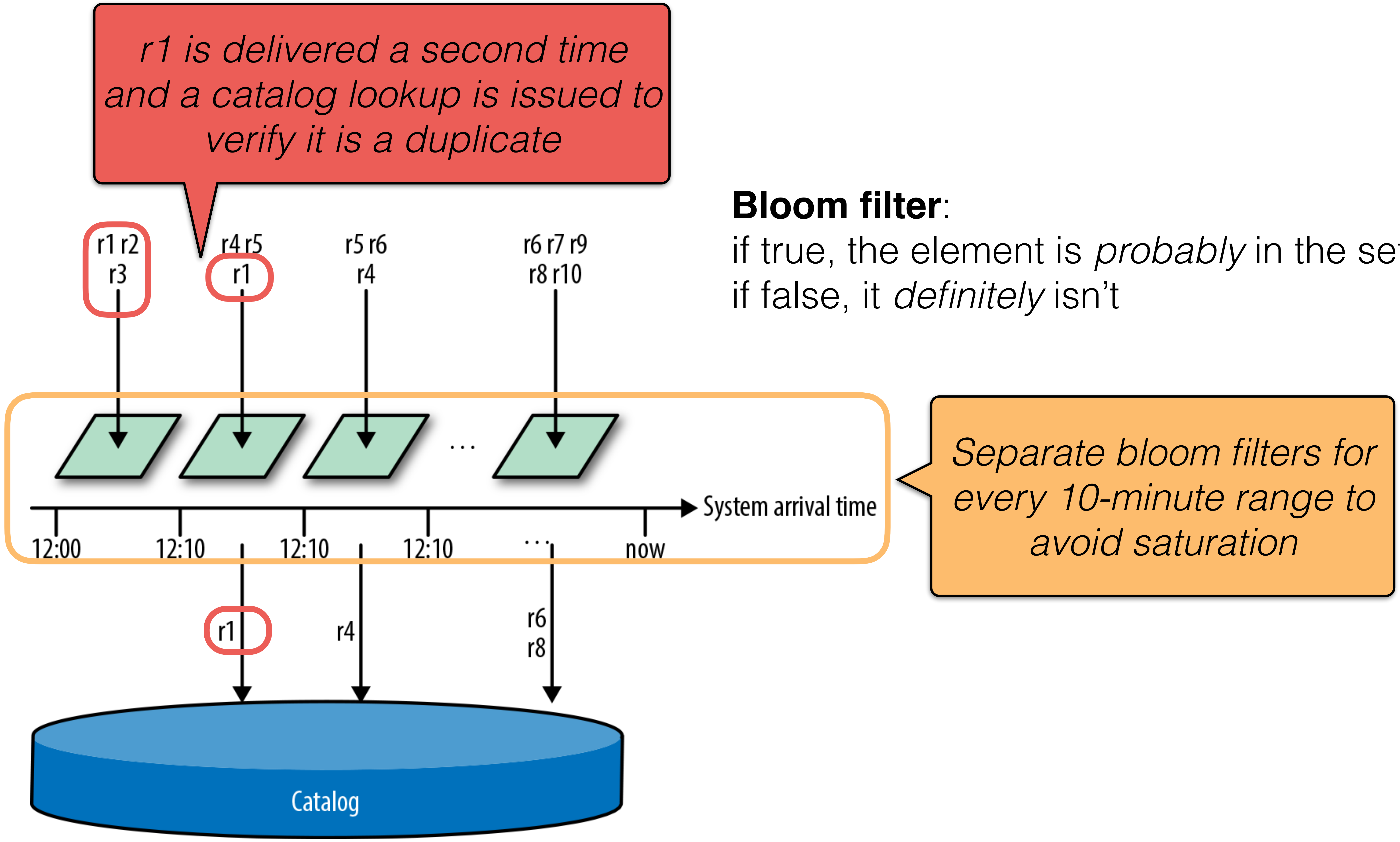
Bloom filter:
if true, the element is *probably* in the set
if false, it *definitely* isn't



http://streamingbook.net/fig/5-5

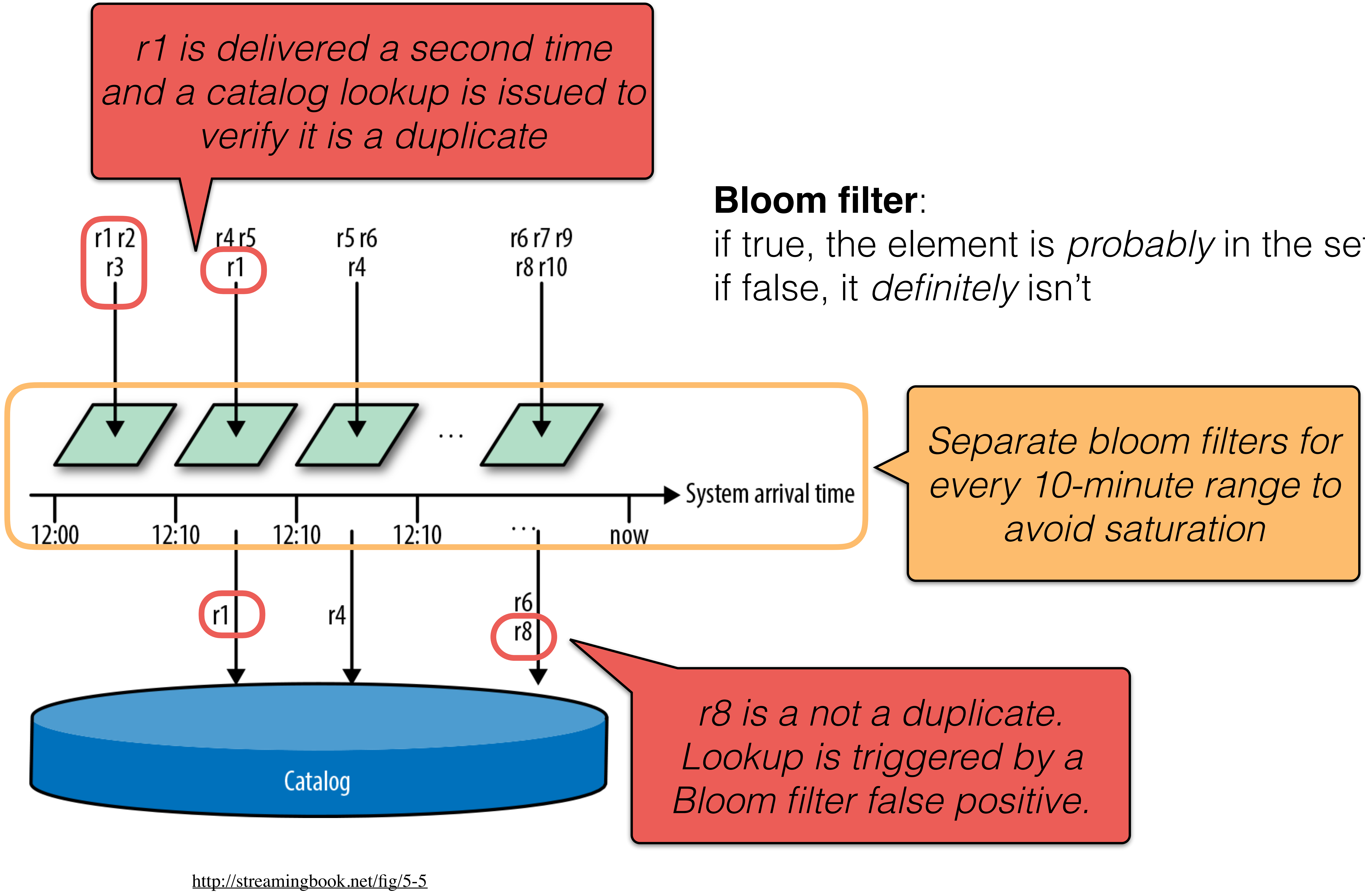
Bloom filter:
if true, the element is *probably* in the set
if false, it *definitely* isn't

Separate bloom filters for every 10-minute range to avoid saturation



http://streamingbook.net/fig/5-5

Bloom filter:
if true, the element is *probably* in the set
if false, it *definitely* isn't



Bloom filter:
if true, the element is *probably* in the set
if false, it *definitely* isn't

Further resources

- Jeong-Hyon Hwang et al. **High-Availability Algorithms for Distributed Stream Processing**. (ICDE ’05).
  - http://cs.brown.edu/research/aurora/hwang.icde05.ha.pdf
- Tyler Akidau et. al. **MillWheel: Fault-Tolerant Stream Processing at Internet Scale** (PVLDB’13)
  - https://research.google/pubs/pub41378/