

Beyond Analytics

The Evolution of Stream Processing Systems

Fault recovery and high availability

Paris Carbone, Marios Fragkoulis, Vasiliki Kalavri, Asterios Katsifodimos

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

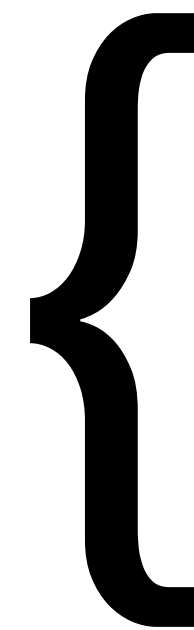


Tutorial overview

- 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)

Overview

- I. Fault recovery
- II. High availability



- Definition
- Approaches
- Challenges

Fault recovery

Fault recovery

Overview

- Definition
- Main approaches
- Challenges

Informal definition

How to recover from a failure as if the failure had never happened

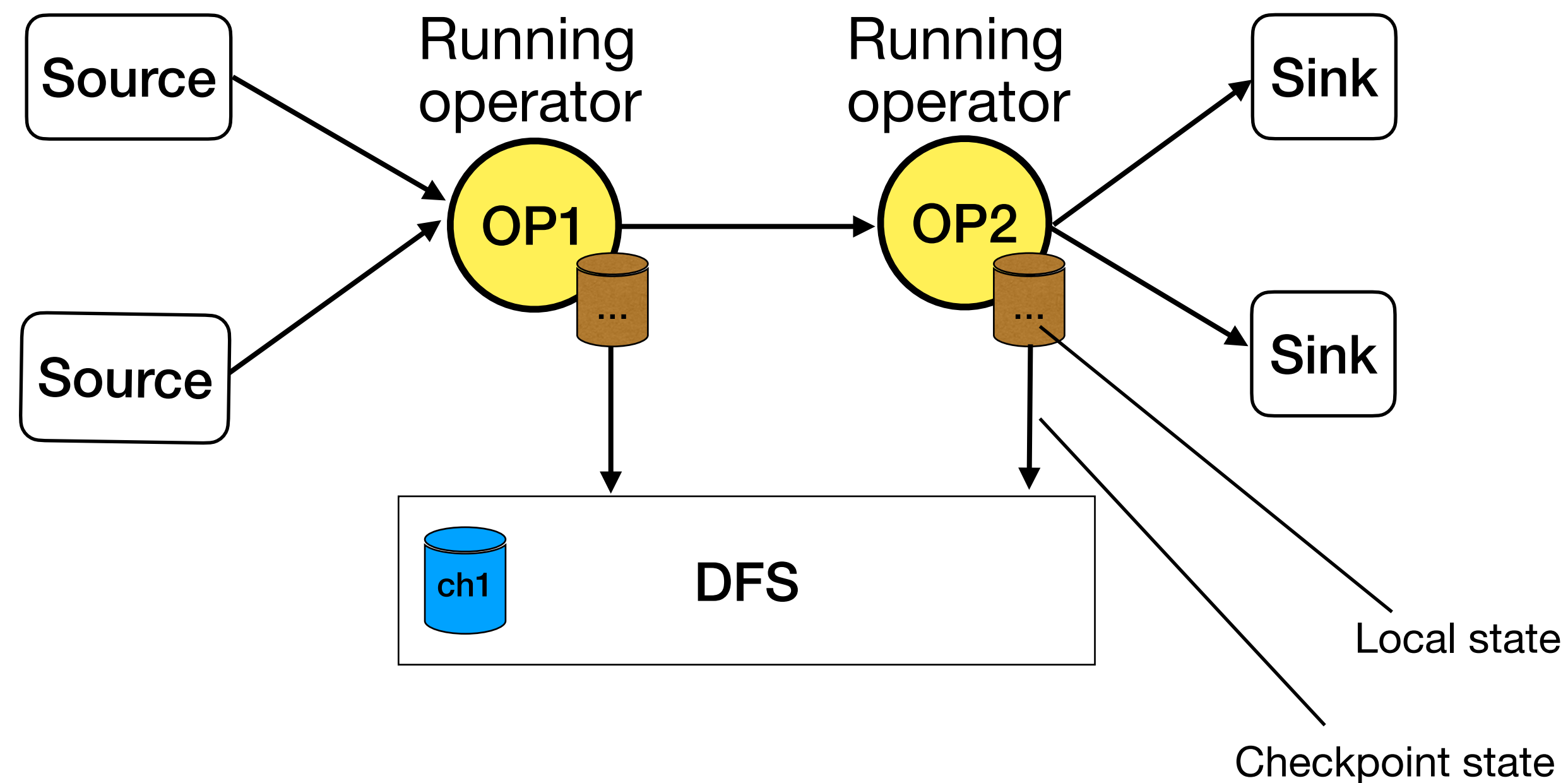
- The essence of fault tolerance
- Why the term fault recovery?
 - We put emphasis on the recovery part leaving out failure models and diagnostics
- The definition implies a strict notion of consistency
 - It's all about state (remember **processing guarantees?**)
 - **State management** with *Paris*

Main approaches

- Checkpoint-based recovery
 - Accumulated state
- Lineage-based recovery
 - State updates

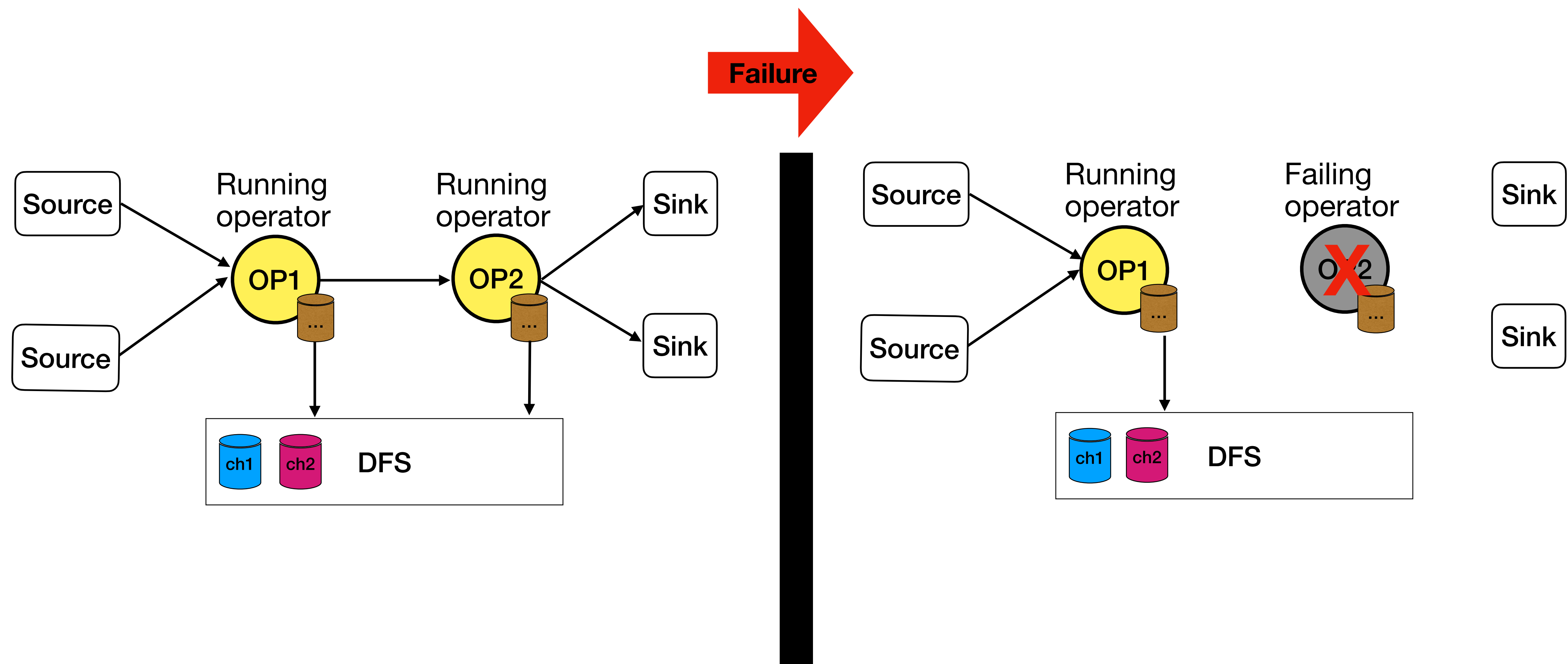
Which approach is better? An analysis is provided in Samza (VLDB '17)

Checkpoint-based recovery

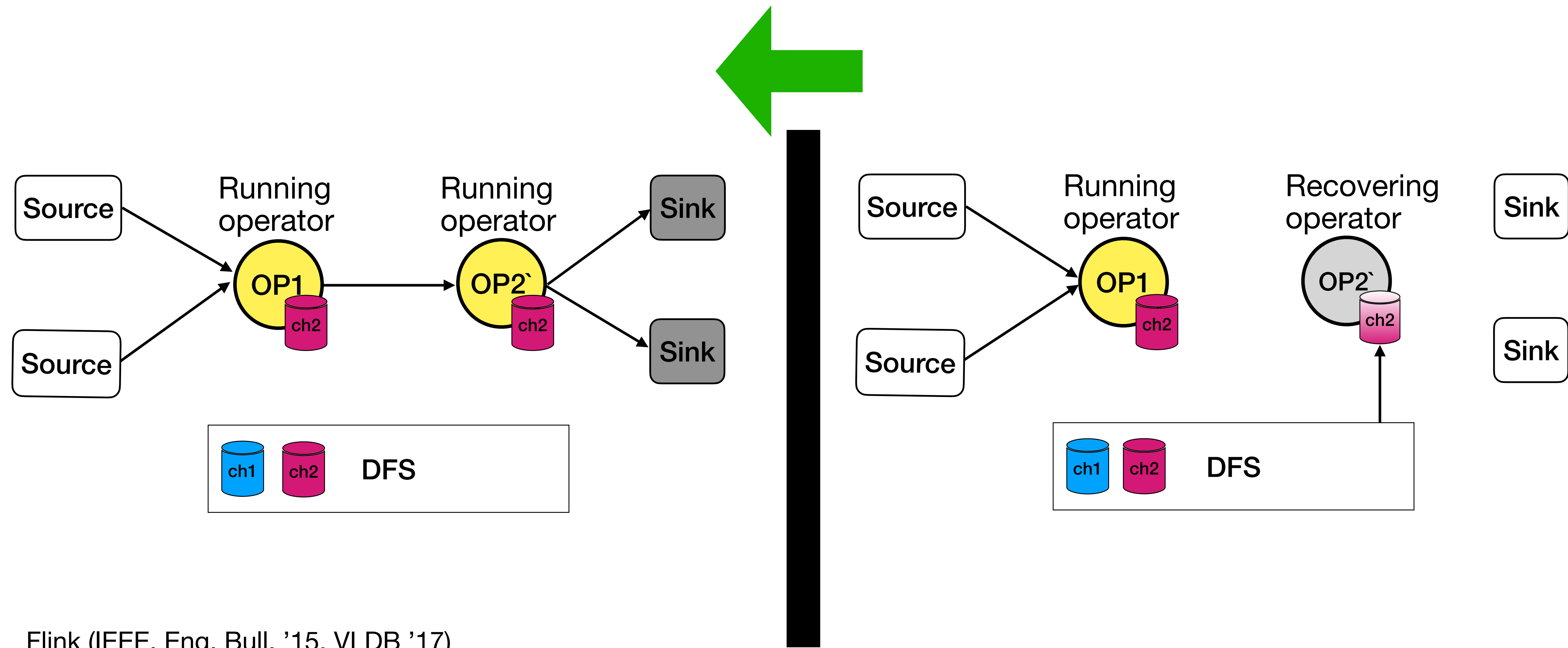


- OP1 receives tuples from sources
- OP1 and OP2 take periodic coordinated checkpoints recording their local state
- Checkpoints are written to resilient storage

Checkpoint-based recovery

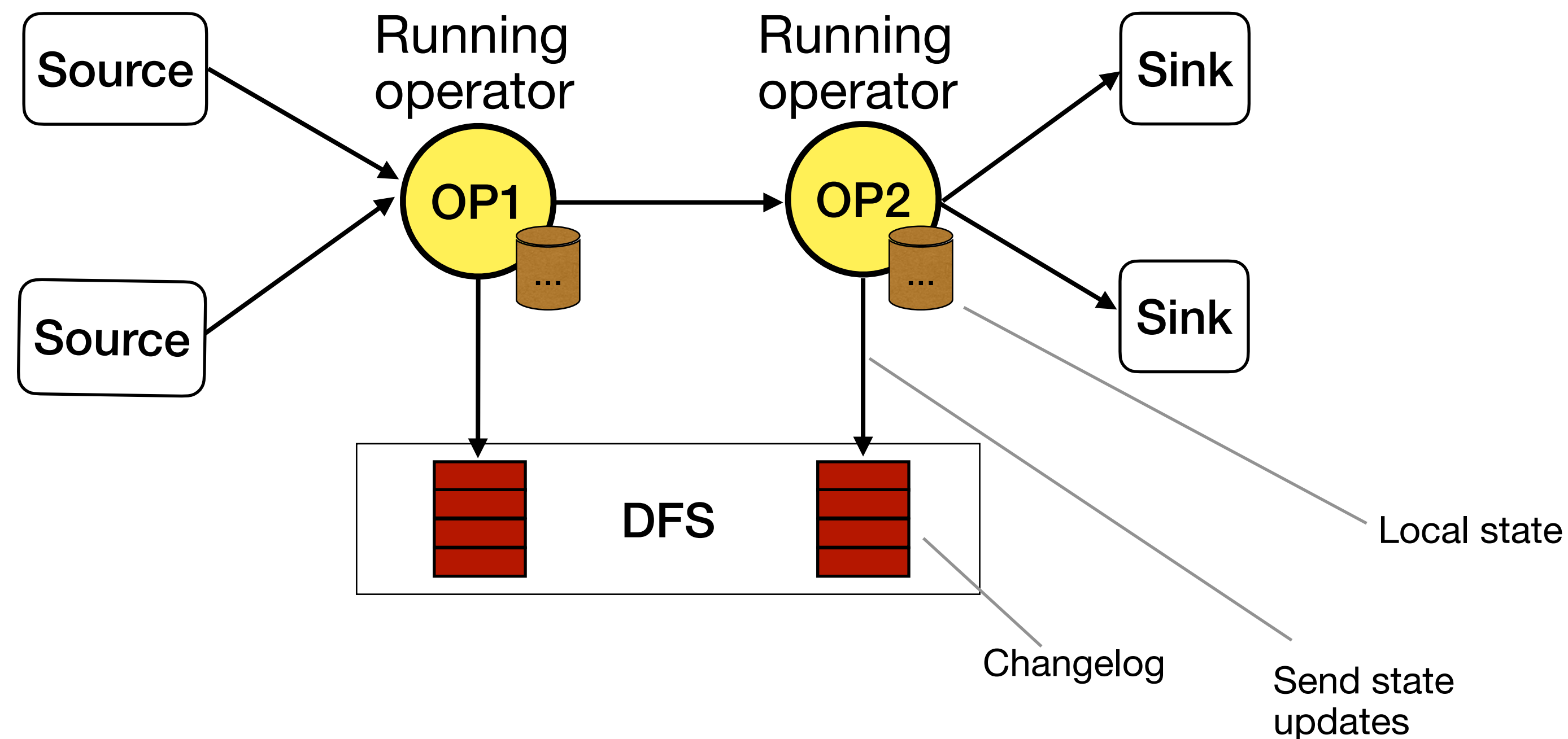


Checkpoint-based recovery



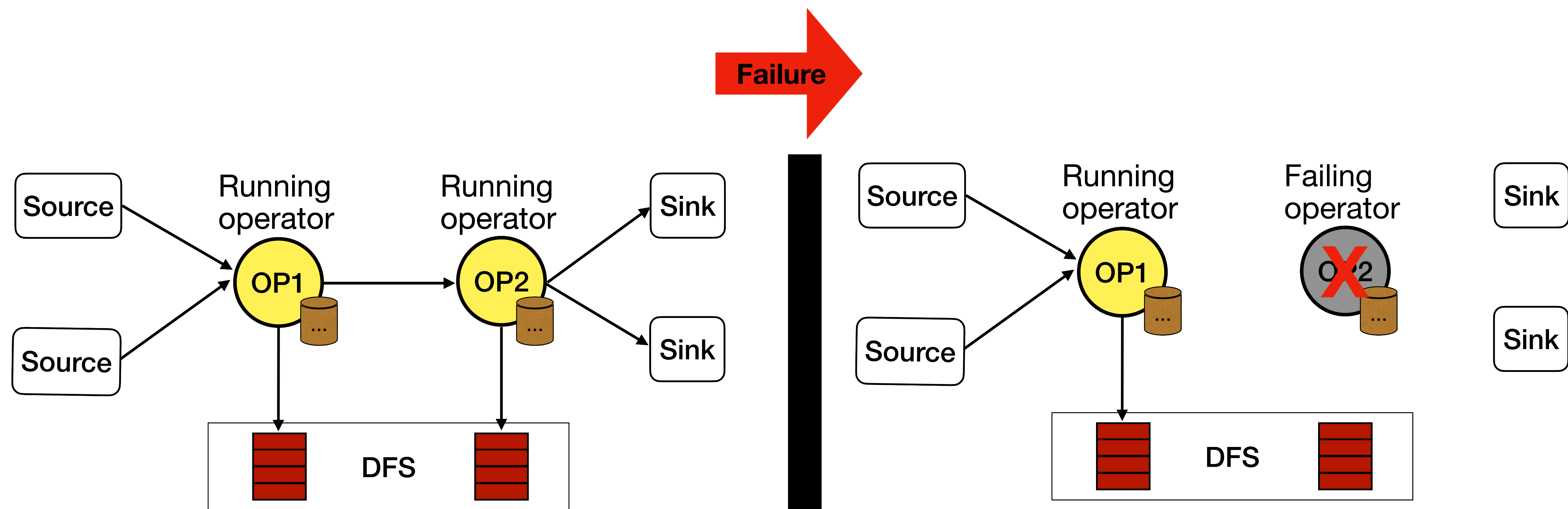
Flink (IEEE. Eng. Bull. '15, VLDB '17)
IBM Streams (VLDB '16)
Spark (SIGMOD '18)

Lineage-based recovery

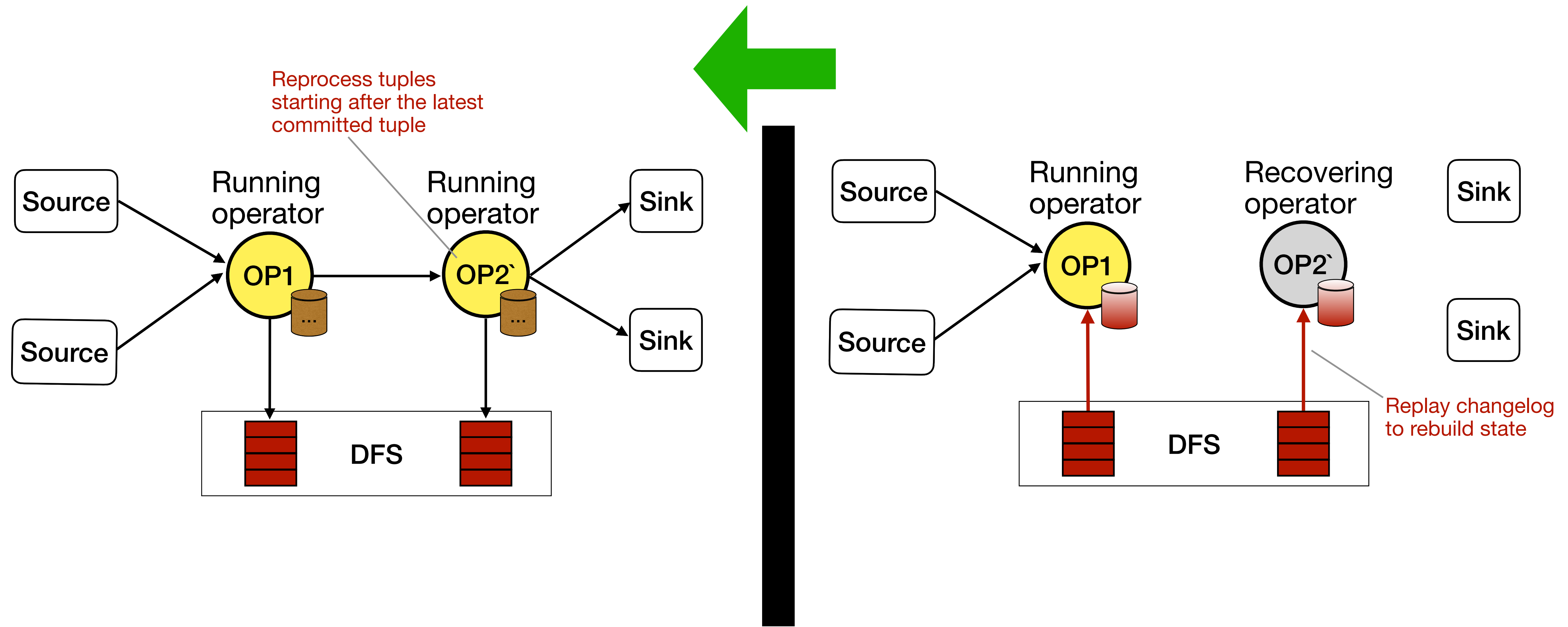


- OP1 receives tuples from sources
- OP1 and OP2 send state updates in batches to the resilient store and commit the latest processed tuple for each batch

Lineage-based recovery



Lineage-based recovery



Challenges

Consistency

- State management with Paris
 - Processing guarantees
 - At-most once
 - At-least once
 - Exactly-once
 - State
 - **Output**

Millwheel (VLDB '13)
Flux (SIGMOD '04)
HA algorithms (ICDE '05)

The output commit problem in streaming

Overview

- Definition
- Solution categories
- Assumptions

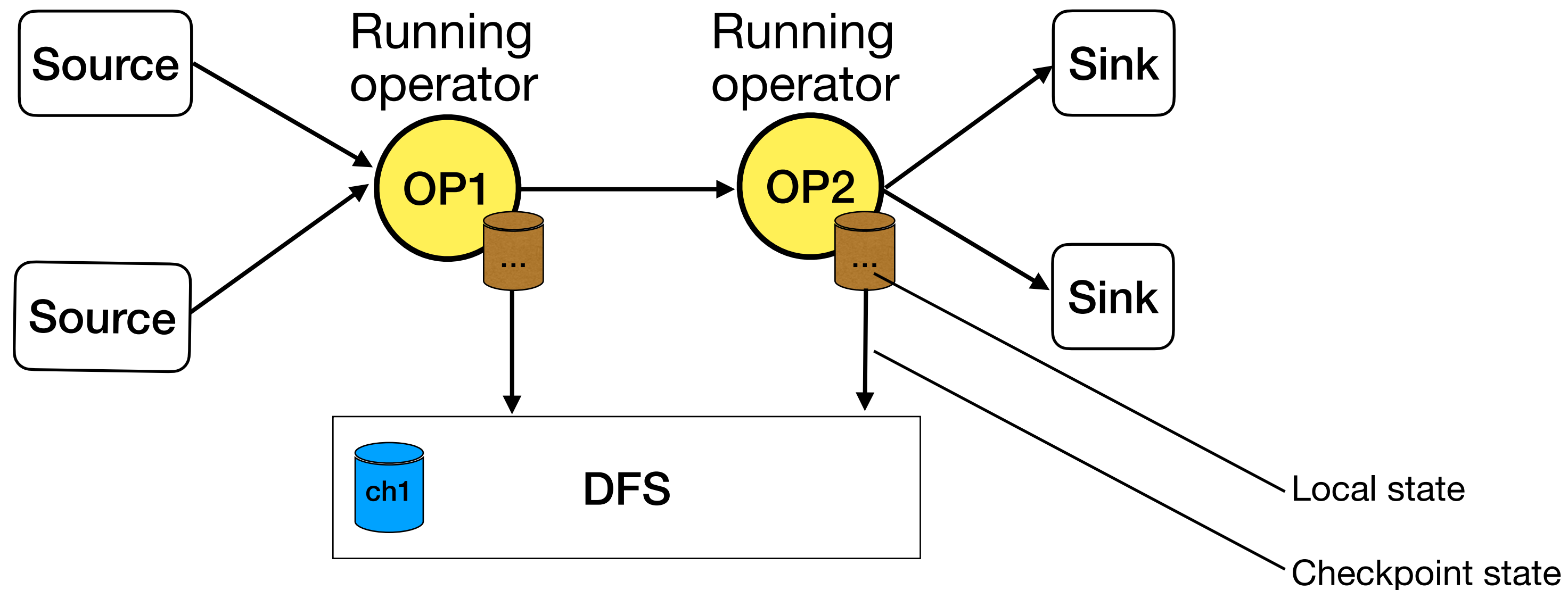
The output commit problem in streaming

Definition

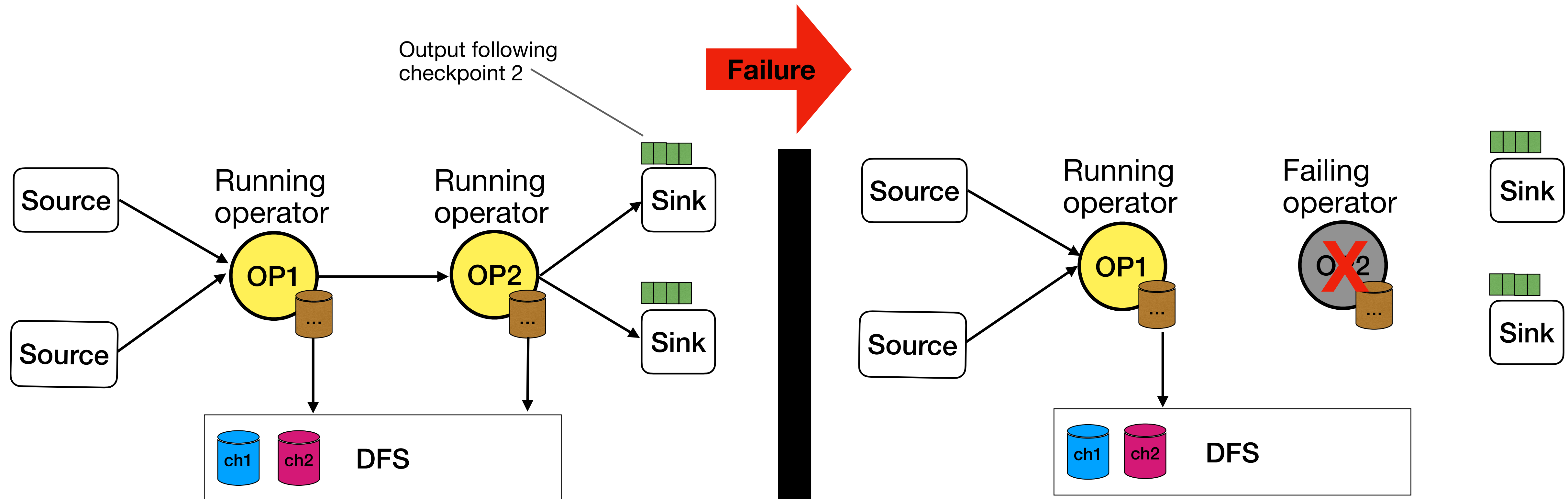
- A system should only publish output to the outside world when it is certain that the system can recover the state from where the output was published so that every output is only published once
 - A survey of rollback recovery in message-passing systems (ACM Comput. Sur. '02)
- The problem manifests when a system is restoring some previous consistent state due to a failure
- Other terms that refer to the same problem
 - Exactly-once processing on output
 - Precise recovery
 - HA algorithms (ICDE '05)
 - Strong productions
 - Millwheel (VLDB '13)

Example of the output commit problem

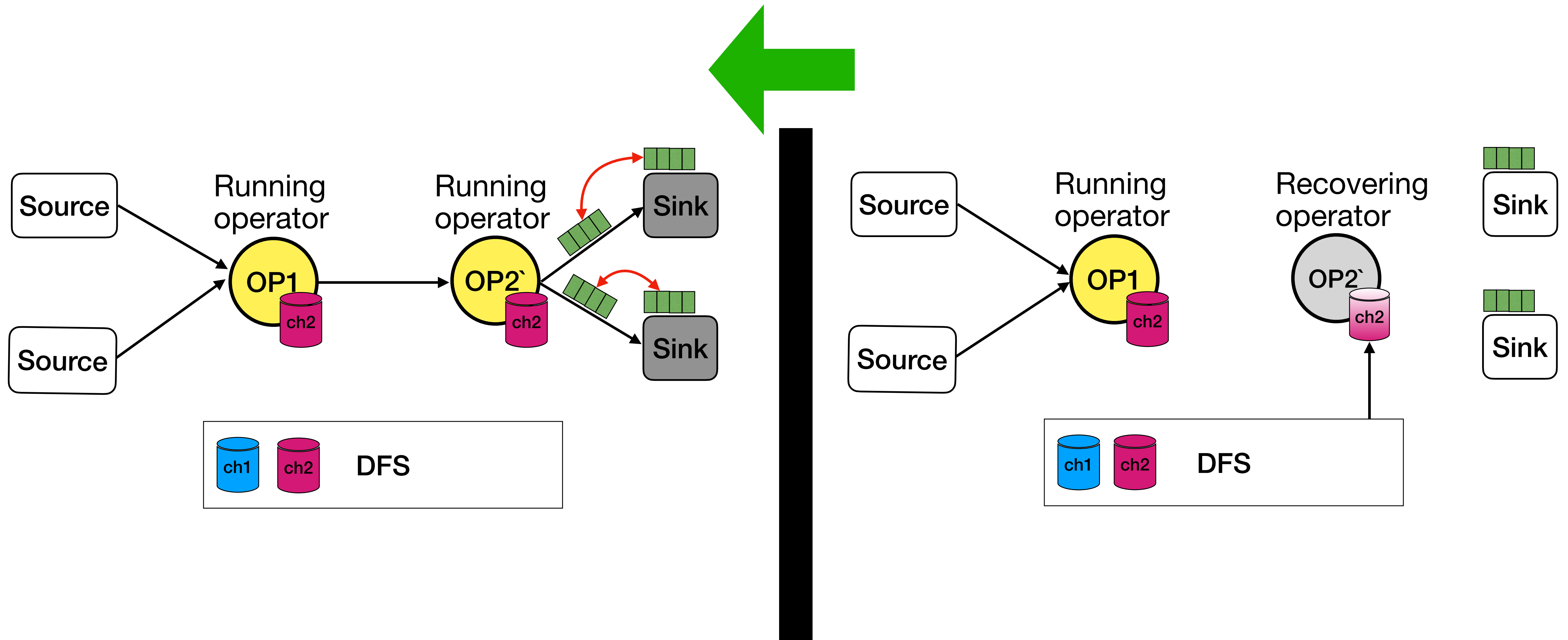
Checkpoint approach



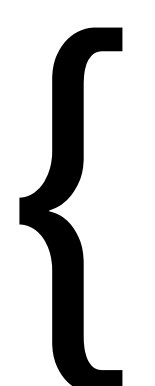
Example of the output commit problem



Example of the output commit problem



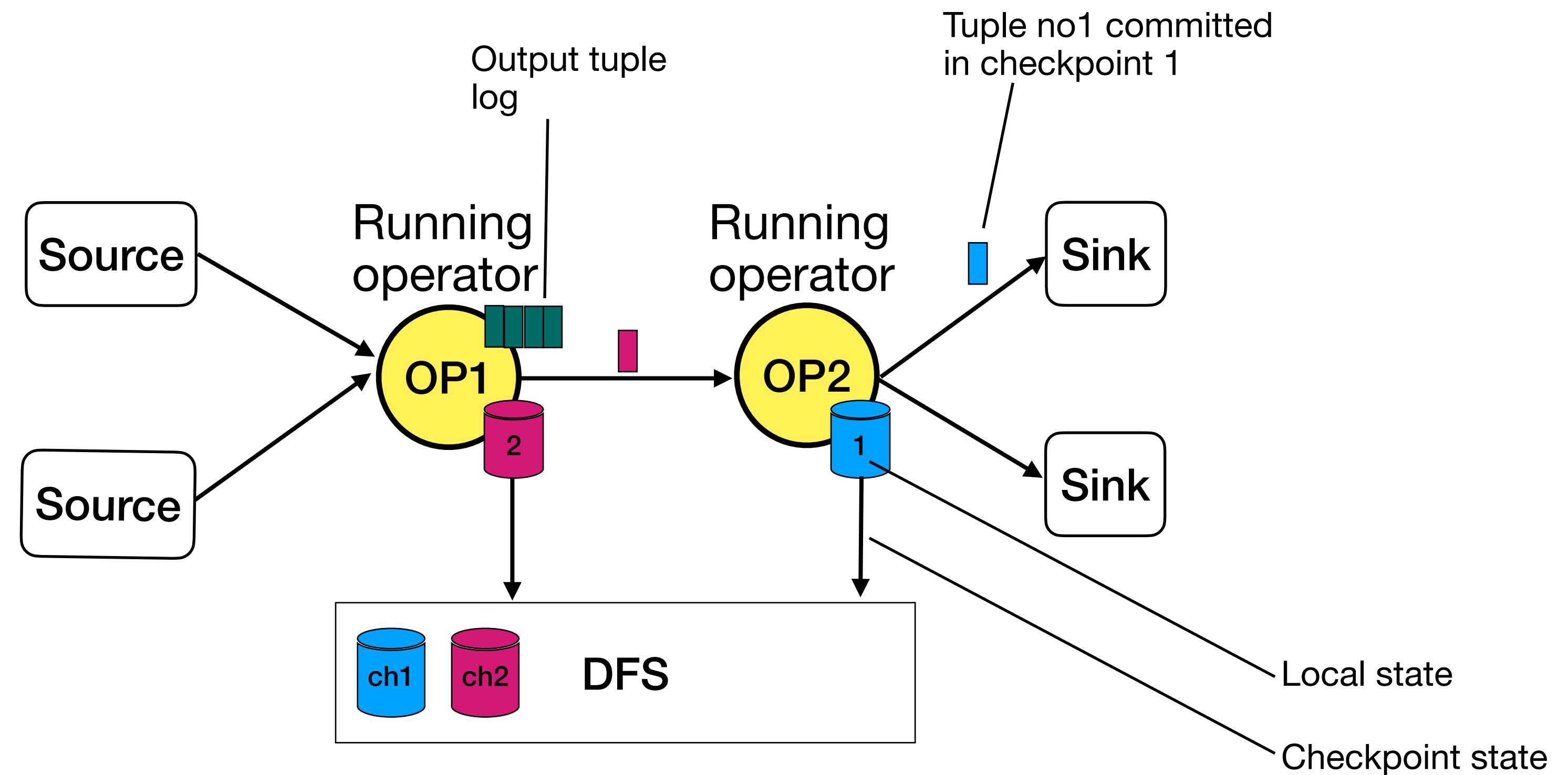
Solutions to the output commit problem

- Transaction-based techniques
 - Using *tuple identity*
 - Time-based techniques
 - Using *progress*
 - Lineage-based techniques
 - Using *input-output dependencies*
 - Special sink operators
 - External sinks
-  Solutions that don't satisfy the problem

Transaction-based

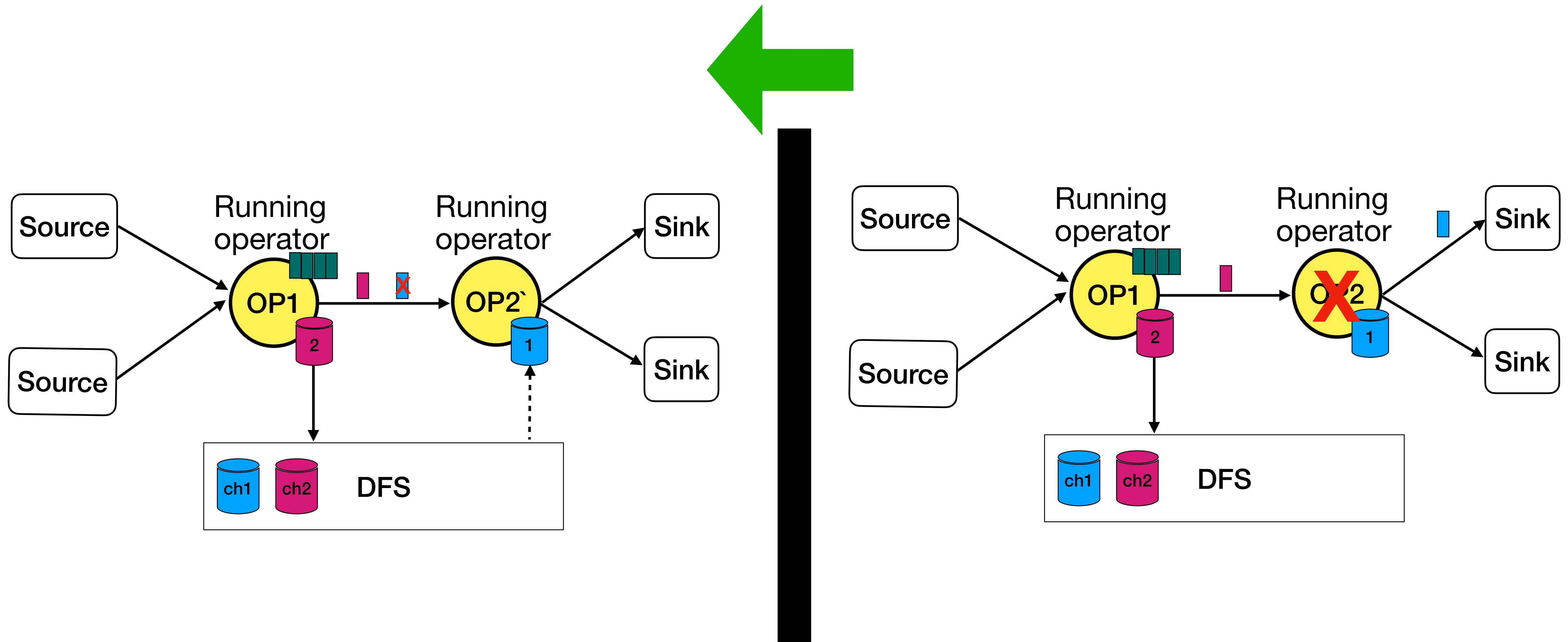
Using tuple identity

- The system commits/checkpoints a unique id with each tuple or a batch of tuples and discards retries on recovery based on tuple/batch id



Transaction-based

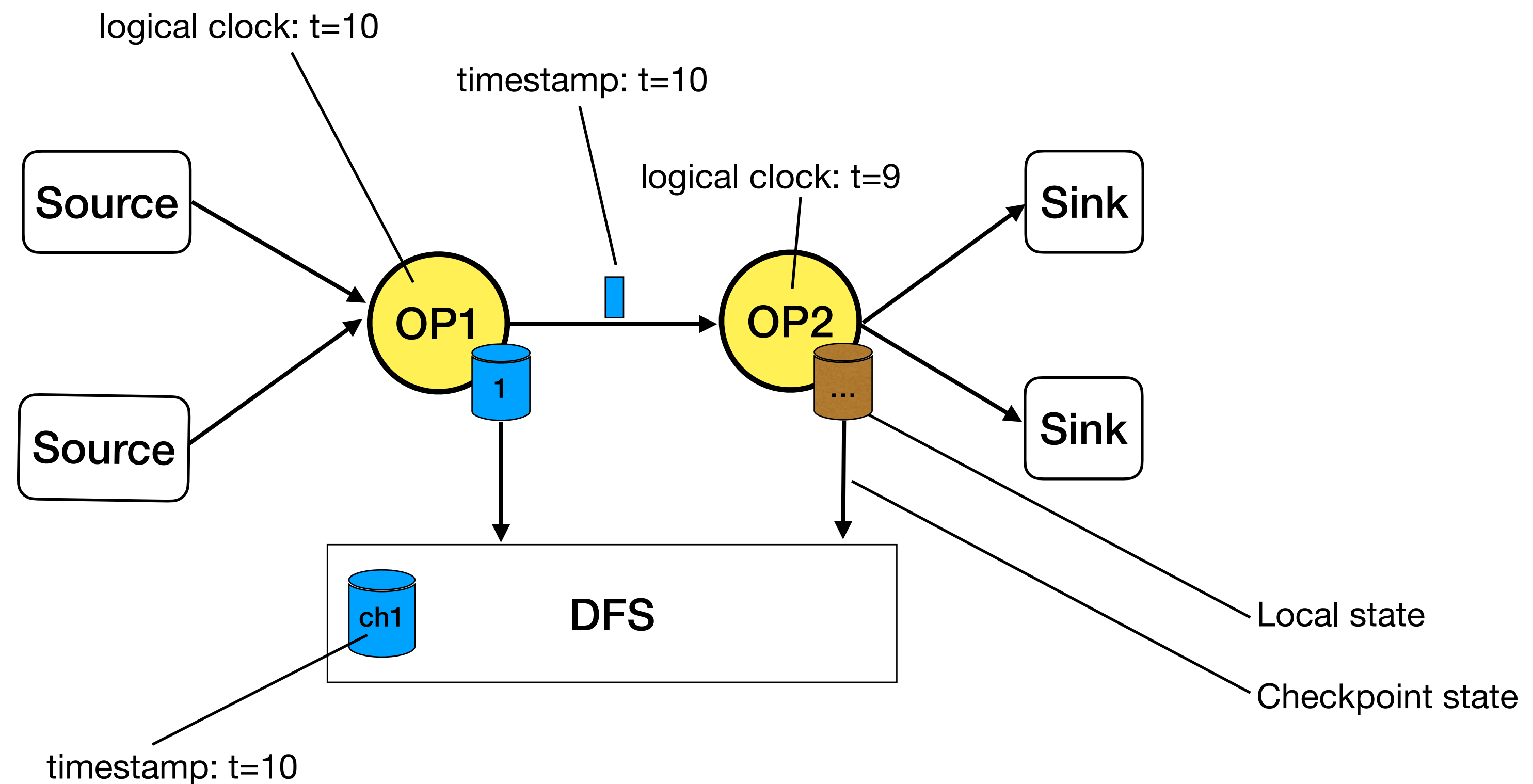
Using tuple identity



Time-based

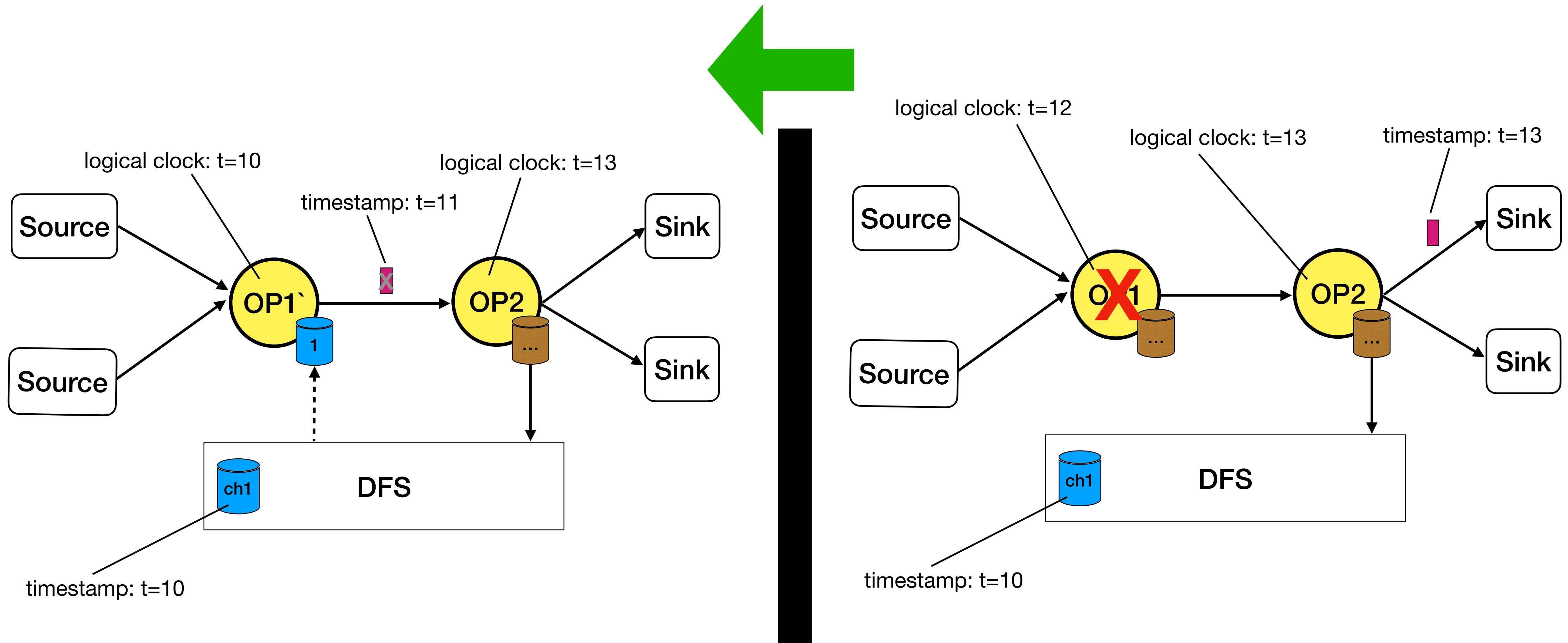
Using progress

- The system uses a logical clock in each operator that attaches monotonically-increasing timestamps to tuples
- Each operator checkpoints also the timestamp of the last tuple that modified the operator's state
- On recovery, the operator tunes its clock to the checkpoint timestamp and starts processing tuples following that checkpoint
- Downstream operators discard tuples with older timestamp than the current timestamp of their logical clock



Time-based

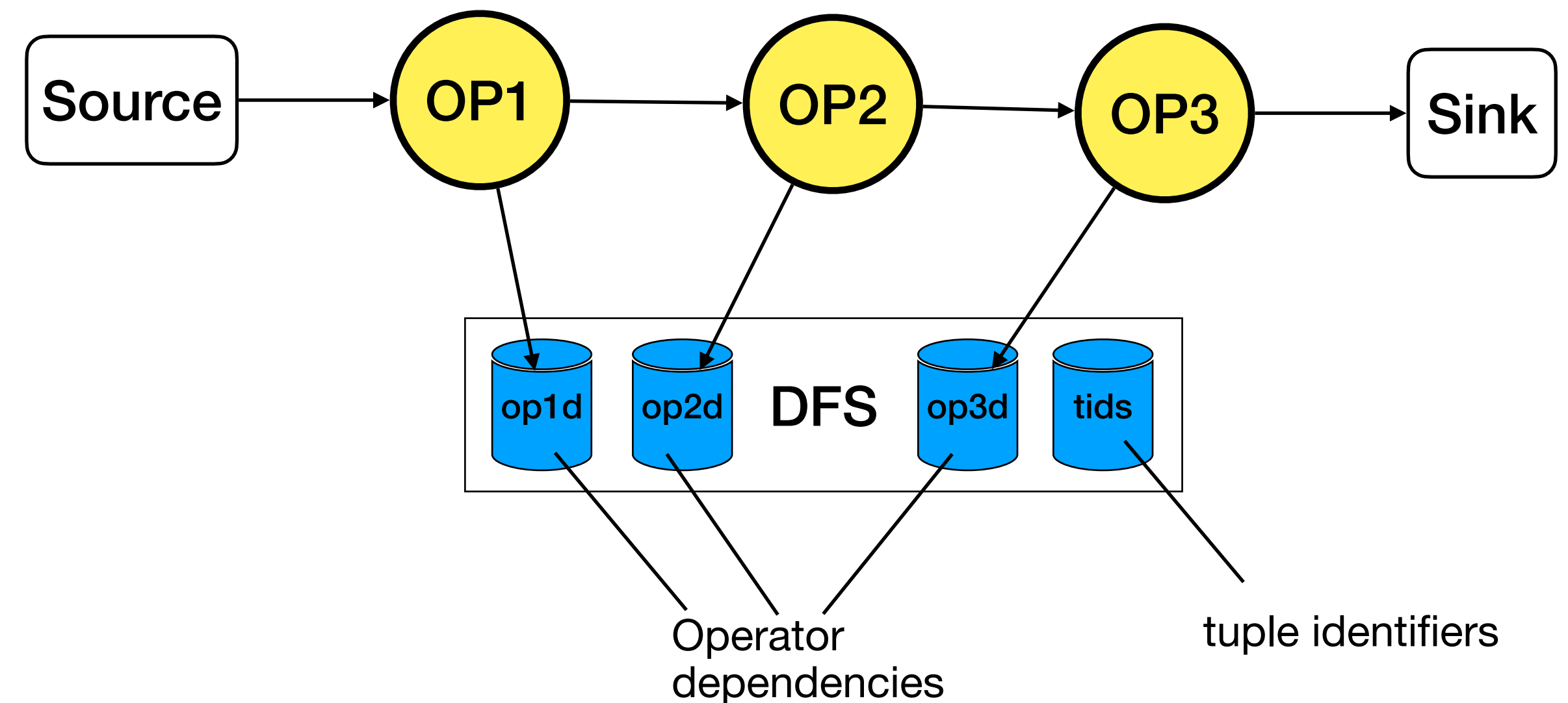
Using progress



Lineage-based

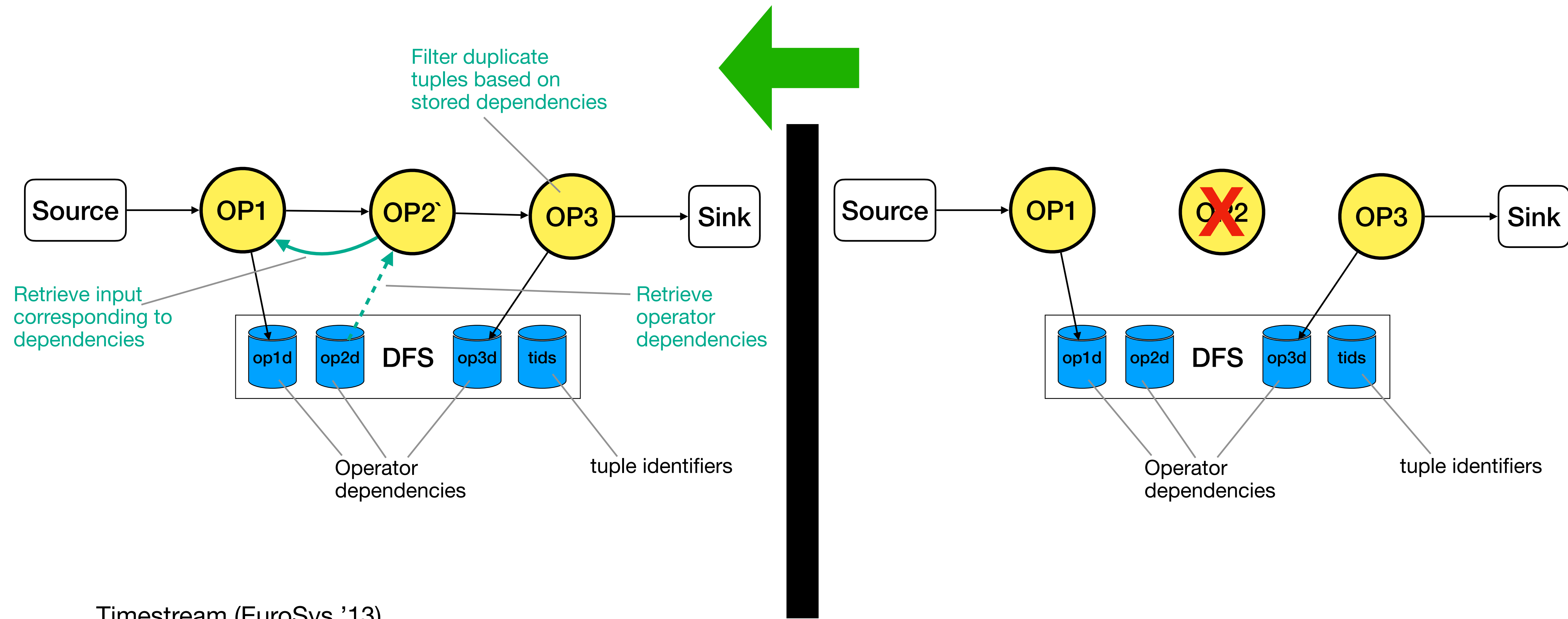
Using input-output dependencies

- The system persists tuple identifiers and input dependencies for each output tuple per operator asynchronously
- A recovering operator receives its persisted input dependencies and asks upstream operators to provide the tuples that correspond to the dependencies
- Upstream operators may ask their upstream operators until all input required to rebuild the recovering operator's state are provided
- Asynchronous storing of dependencies incurs recomputations, which are filtered out by operators downstream of a failure
- The system uses garbage collection to retire dependencies not required any more



Lineage-based

Using input-output dependencies



Timestream (EuroSys '13)
StreamScope (NSDI '16)
Seep (USENIX ATC '14)

Special sink operators

- Can in some cases retract output
 - Truncate a file
 - Remove a record from a database
- Update the versions of a resource (MVCC) and flag it as a dirty read until it is permanently committed
 - Entails that consuming applications can benefit from this capability
- However, solution does not address the problem's specification

{ IBM Streams (VLDB '17)

External sinks

- Outsource output deduplication to an external system
 - Transactional sink
 - Apache Kafka

Special sink operators and external sinks

Optimistic and pessimistic techniques

- Optimistic output techniques (MVCC)
 - Versioned or modifiable (truncatable) output destinations
- Pessimistic output techniques (non-MVCC) using a form of a WAL
 - Transactional sinks

Assumptions of systems' solutions

System	Assumptions
Millwheel	
Timestream, Streamscope	<ul style="list-style-type: none">• Deterministic computation• Deterministic input
Trident	<ul style="list-style-type: none">• Deterministic computation• Deterministic input• Order of transactions
Seep	<ul style="list-style-type: none">• Deterministic computation• Monotonically-increasing logical clock• Timestamp-ordered operator input

High availability

High availability

Overview

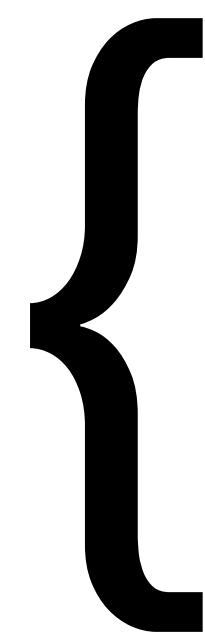
- Definition
- Key word: replication
- Main approaches
 - Active standby (Replicate computations)
 - Passive standby (Replicate state)
 - Upstream backup (Replicate streams)
- Definition revisited — what is availability in data stream processing

Informal definition and profile

- A system is highly available when it can sustain failures with minimal (ideally zero) service interruption
- Availability is categorized in availability classes according to the *period of time that a system operates normally*
 - *High availability* — 99.999% (5 minutes/year service interruption)
 - *Ultra availability* — 99.99999% (0.05 minutes/year service interruption)
- The most popular metric for measuring availability empirically is **recovery time**

High availability approaches

Overview

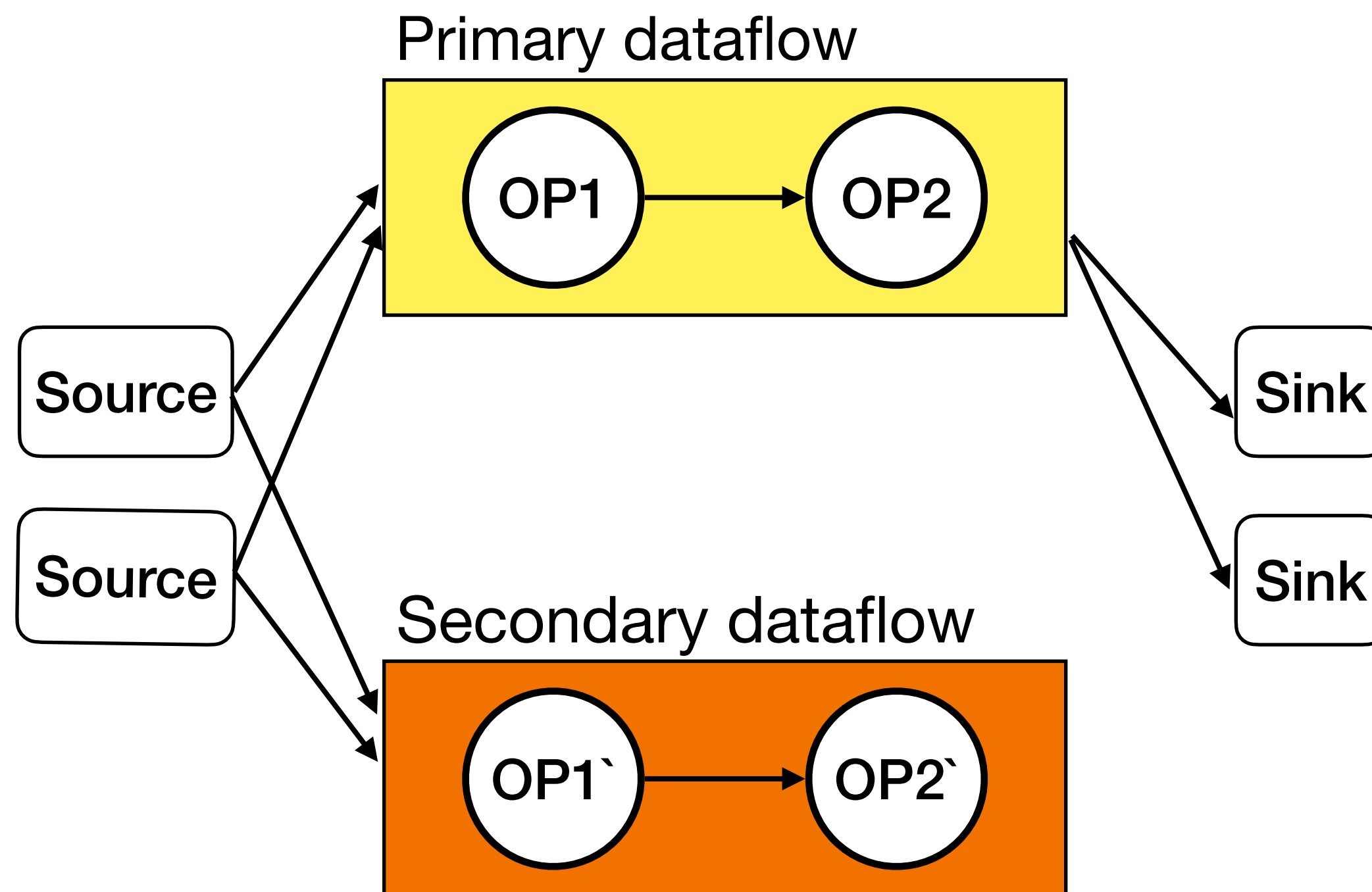
- Active standby
 - Passive standby
 - Upstream backup
- 
- Definition
 - Architectures
 - Challenges

Active standby

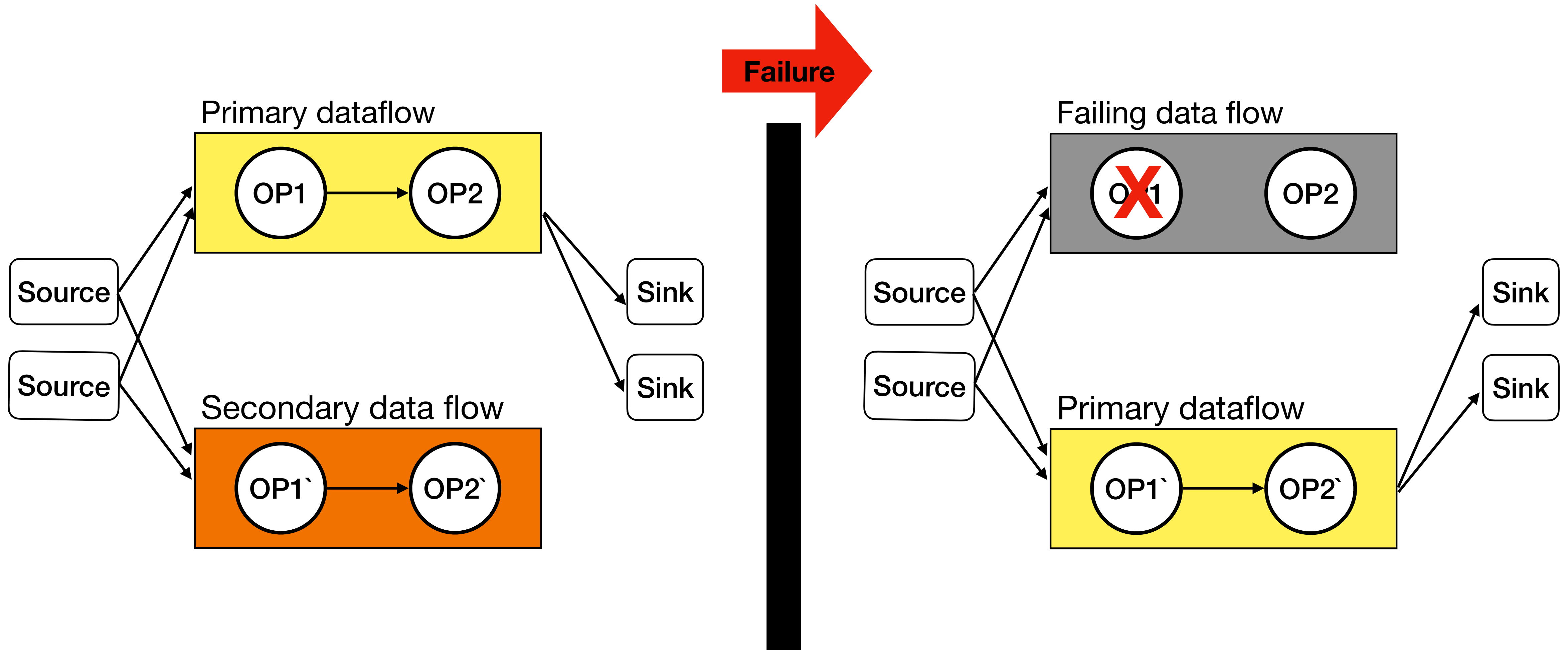
Replicating computations

- Definition
 - Active standby involves running two or more instances of a computation in parallel and switching from the primary to the secondary in case of a failure
- Typical architectures
 - Dataflow replicas (dataflow pairs)
 - Node replicas (node pairs)
- Challenges

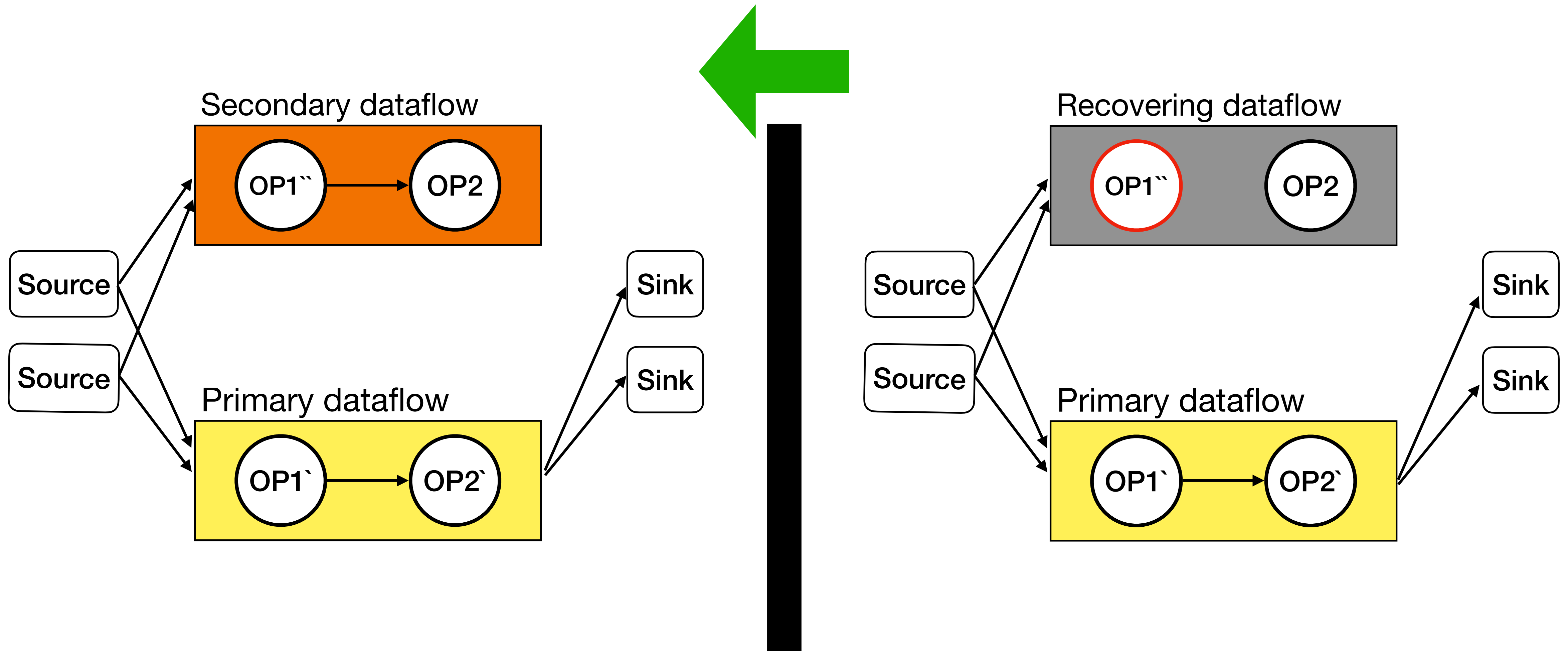
Dataflow replicas



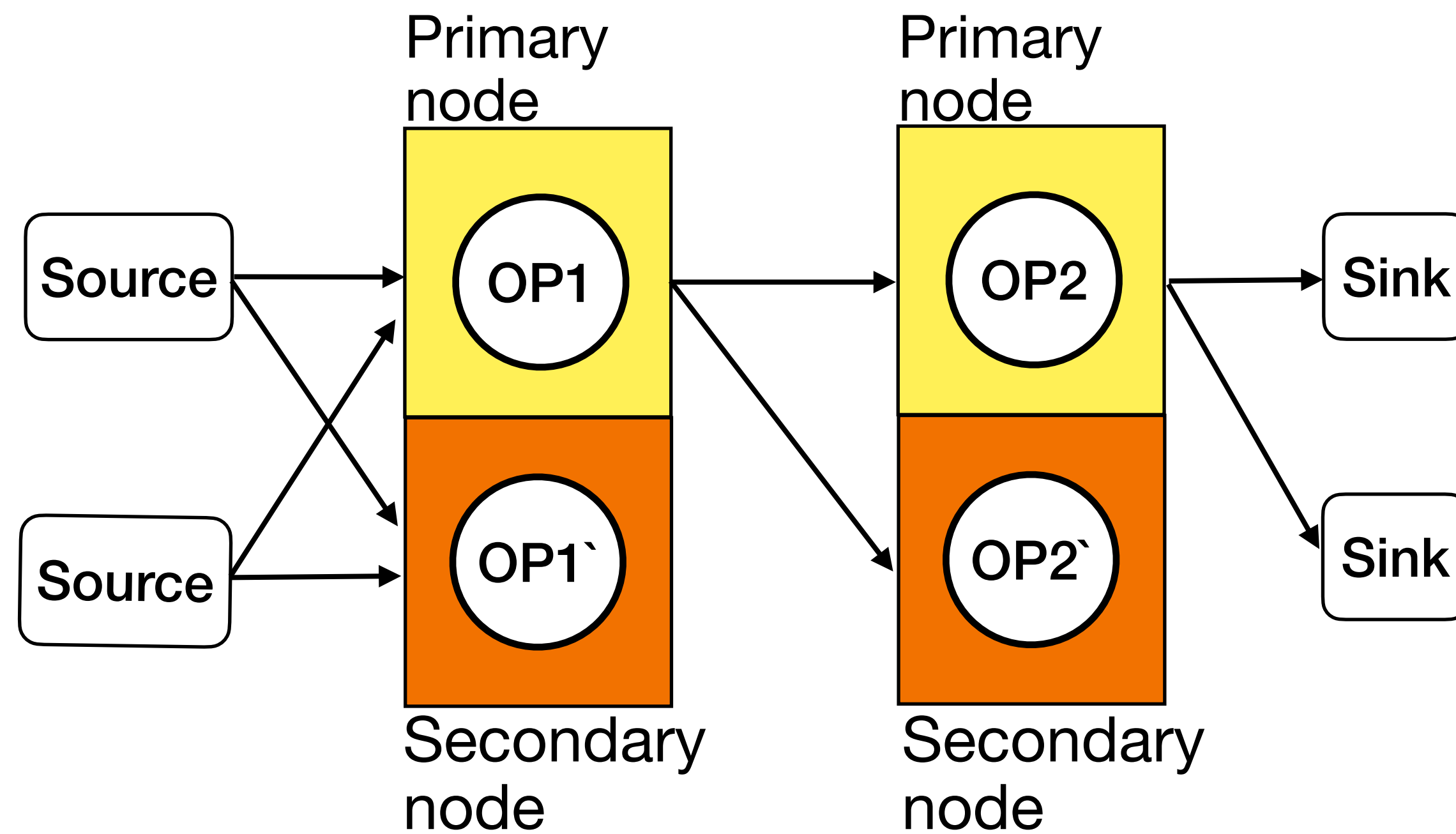
Dataflow replicas



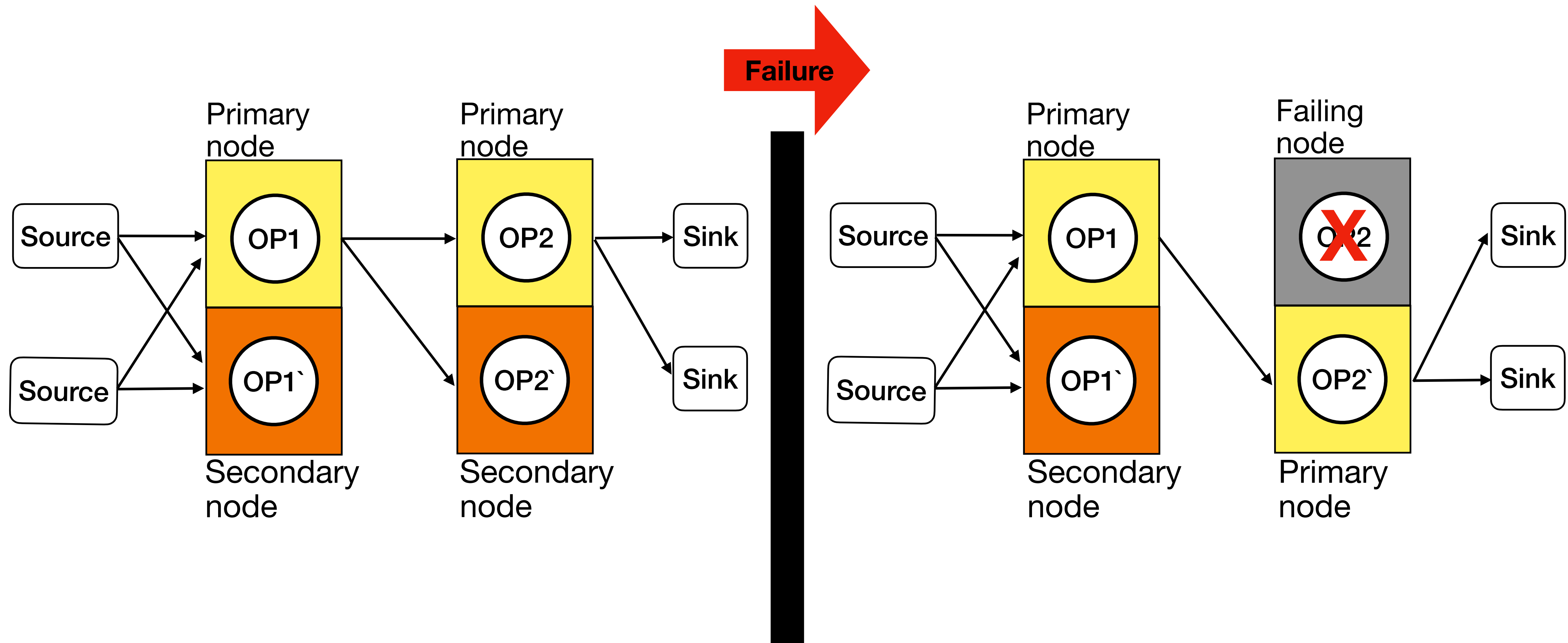
Dataflow replicas



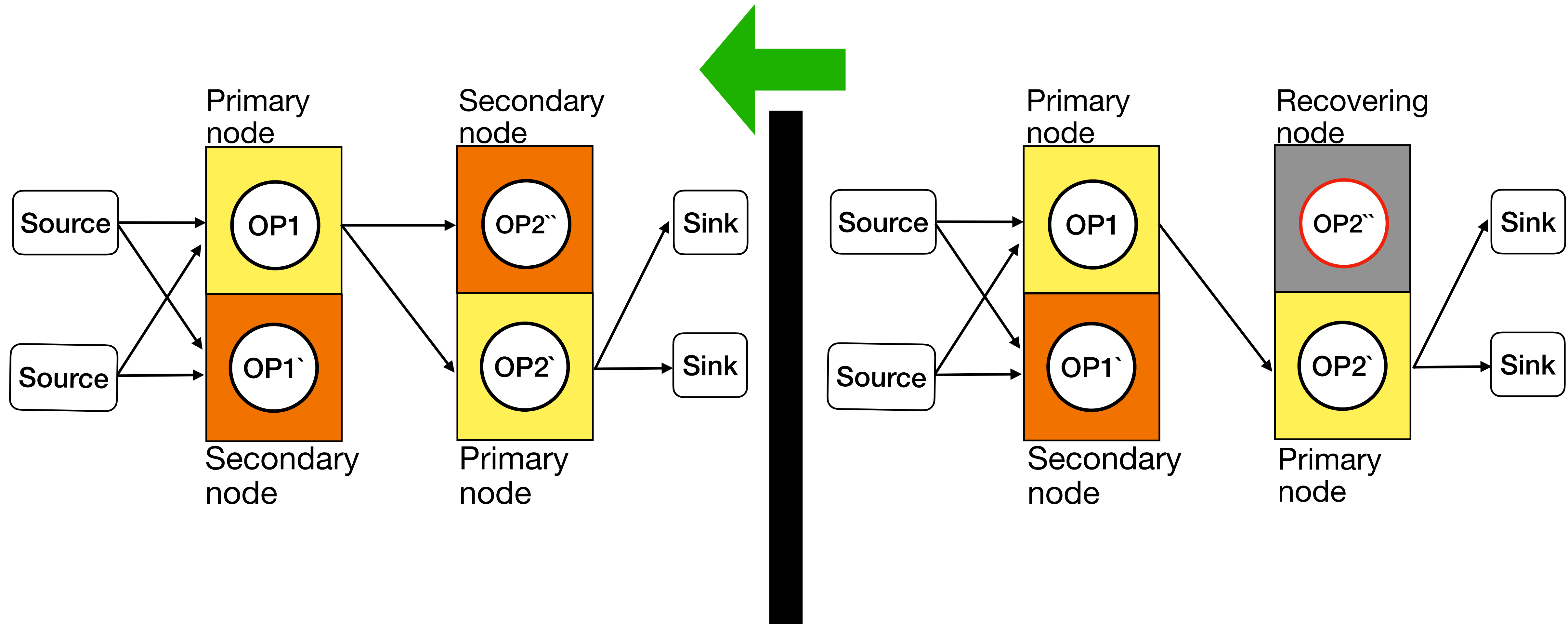
Node replicas



Node replicas



Node replicas



Easy right?

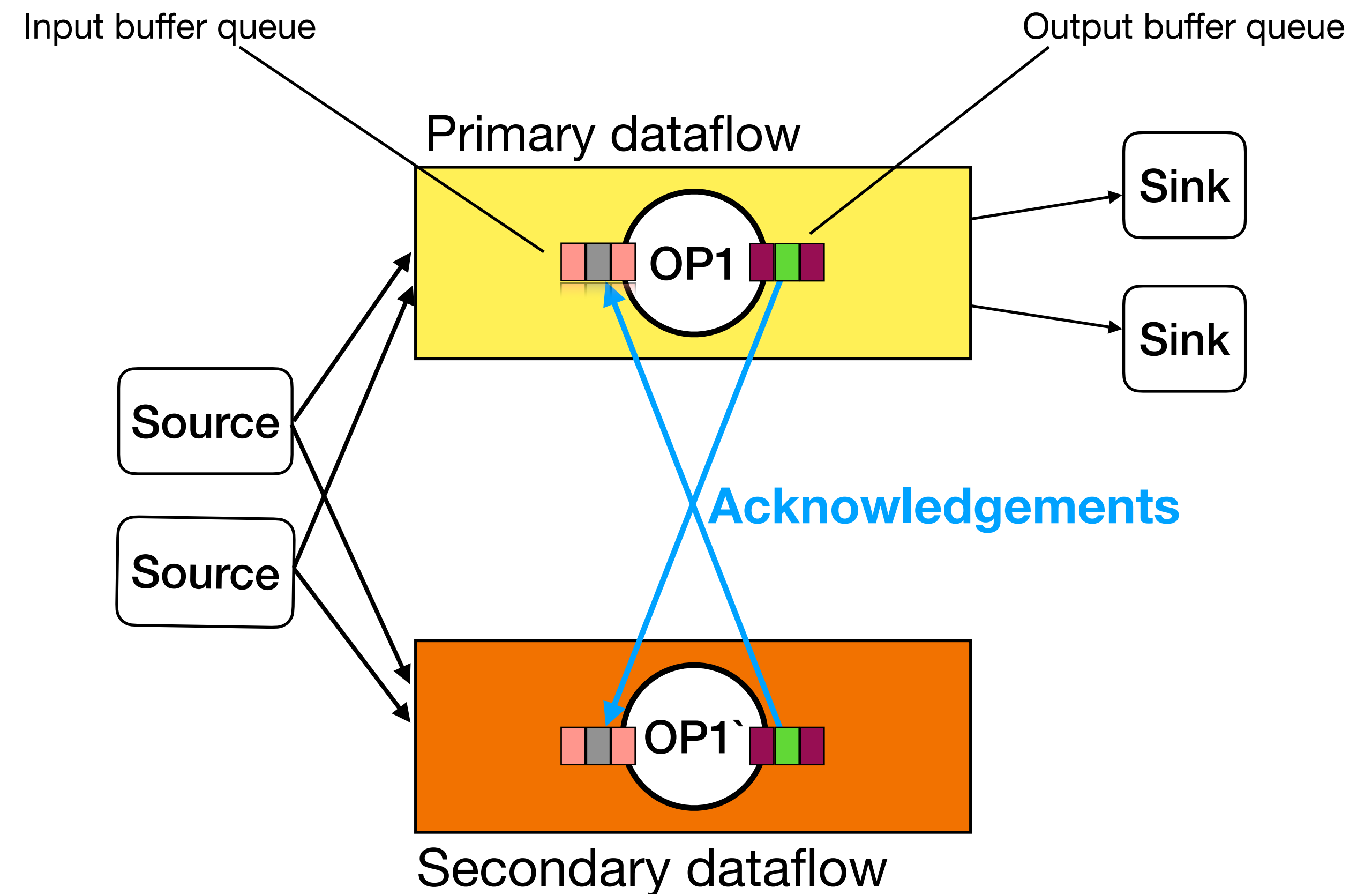


Challenges in active standby

- Coordinating replicas for exactly-once output
- Catching up with input

Coordinating replicas for exactly-once output

- Coordination via **acknowledgements**
- Primary acknowledges tuple's sequence number to secondary and vice versa
- When acknowledge is received tuple is removed from input buffer queue.



Catching up with input: One way

The Good Samaritan

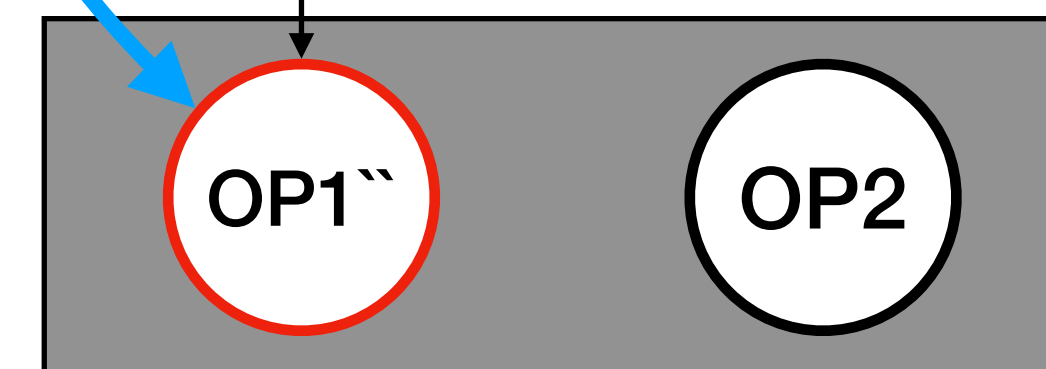
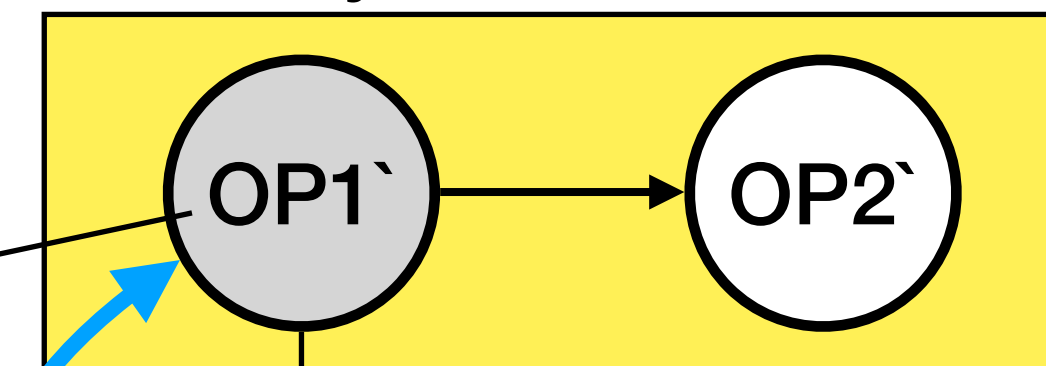
- Piecemeal-catch up per segment of (partitioned) operators in the data flow

1. Quiesce operators in the primary segment

2. Transfer state to the secondary segment

3. Synchronize connections between primary and secondary segments

Primary dataflow

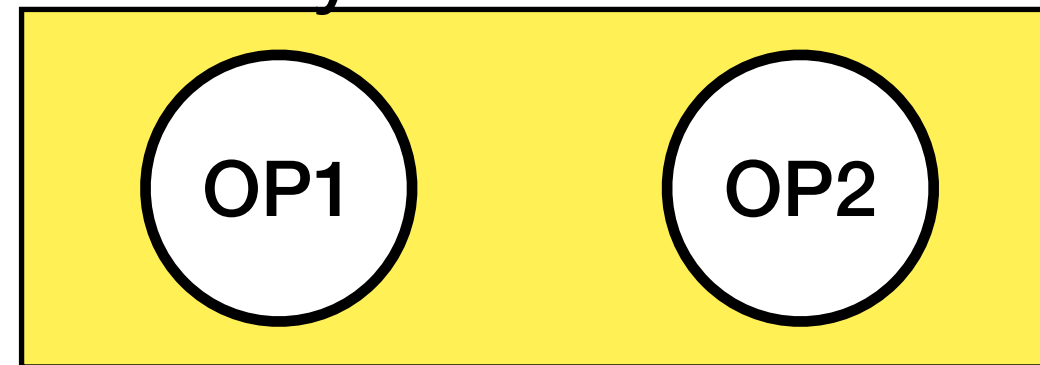


Recovering dataflow

Catching up with input: Second way

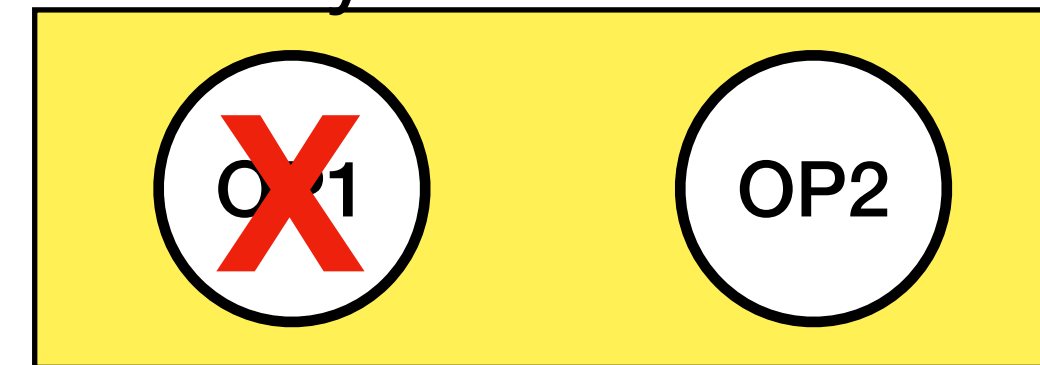
Call AAA

Primary dataflow

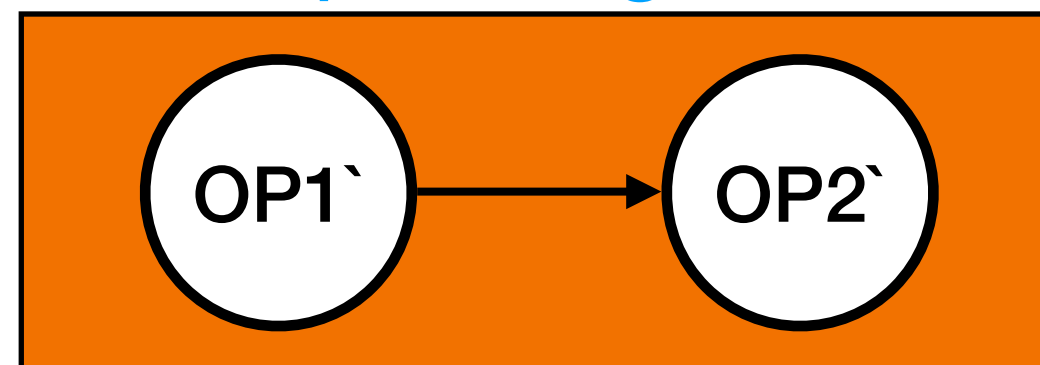


Failure

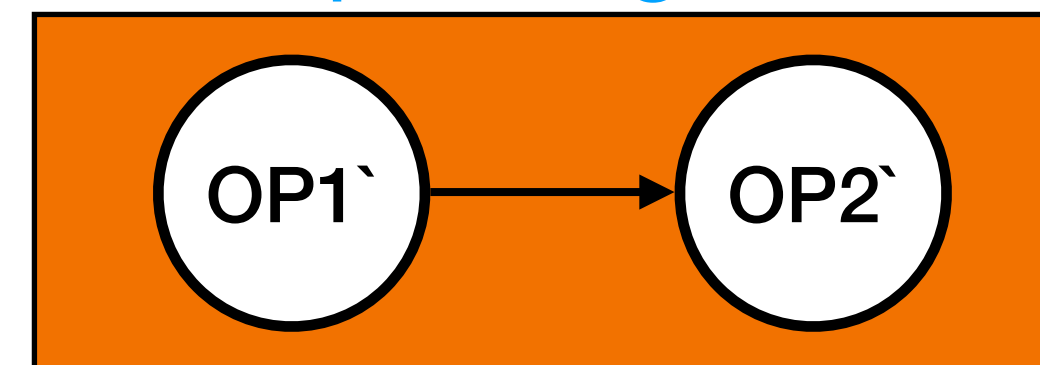
Primary dataflow



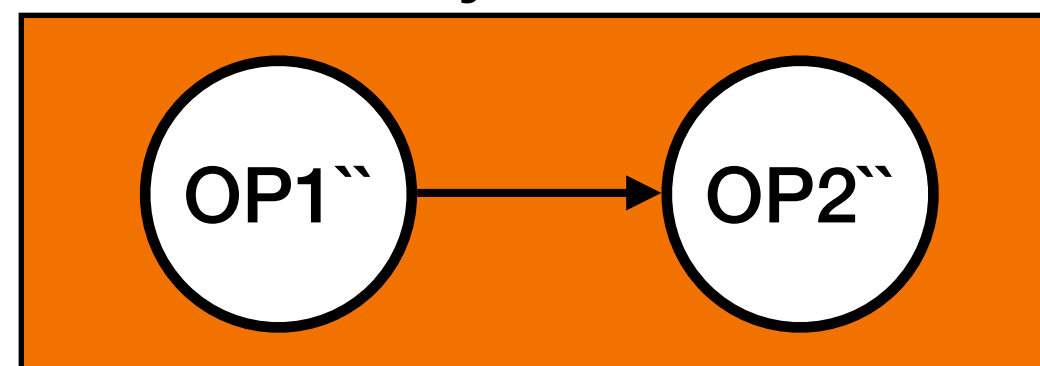
Checkpointing dataflow



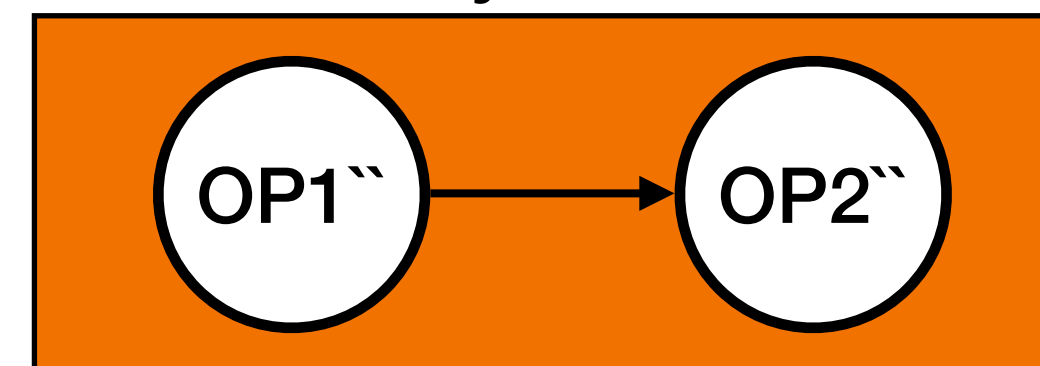
Checkpointing dataflow



Secondary dataflow



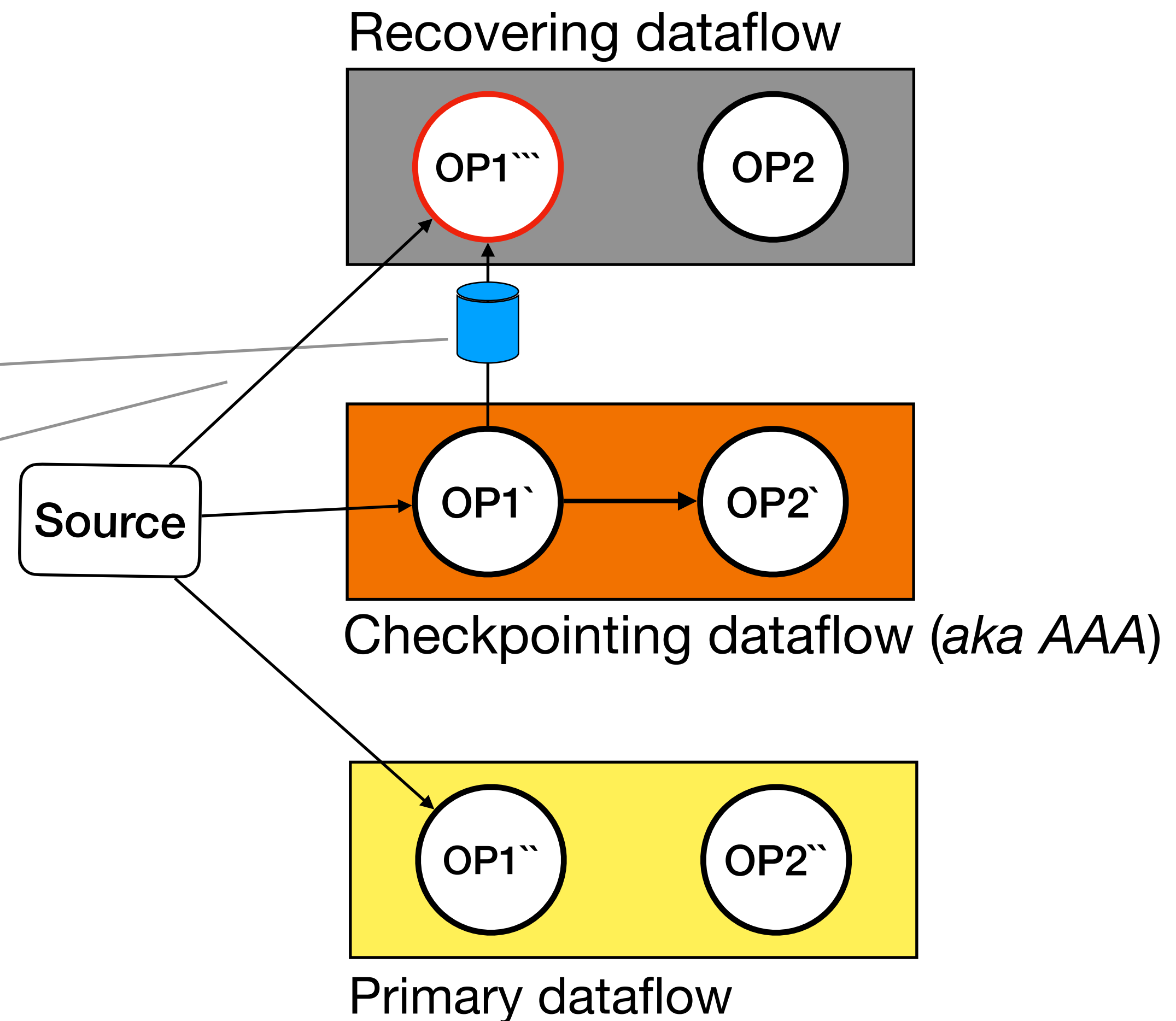
Secondary dataflow



Catch up with input: Second way

Call AAA

- Recover from checkpoint
 1. Transfer checkpoint to the recovering operator
 2. Process tuples from the latest checkpoint
 3. But how is catch up guaranteed since primary doesn't stop?

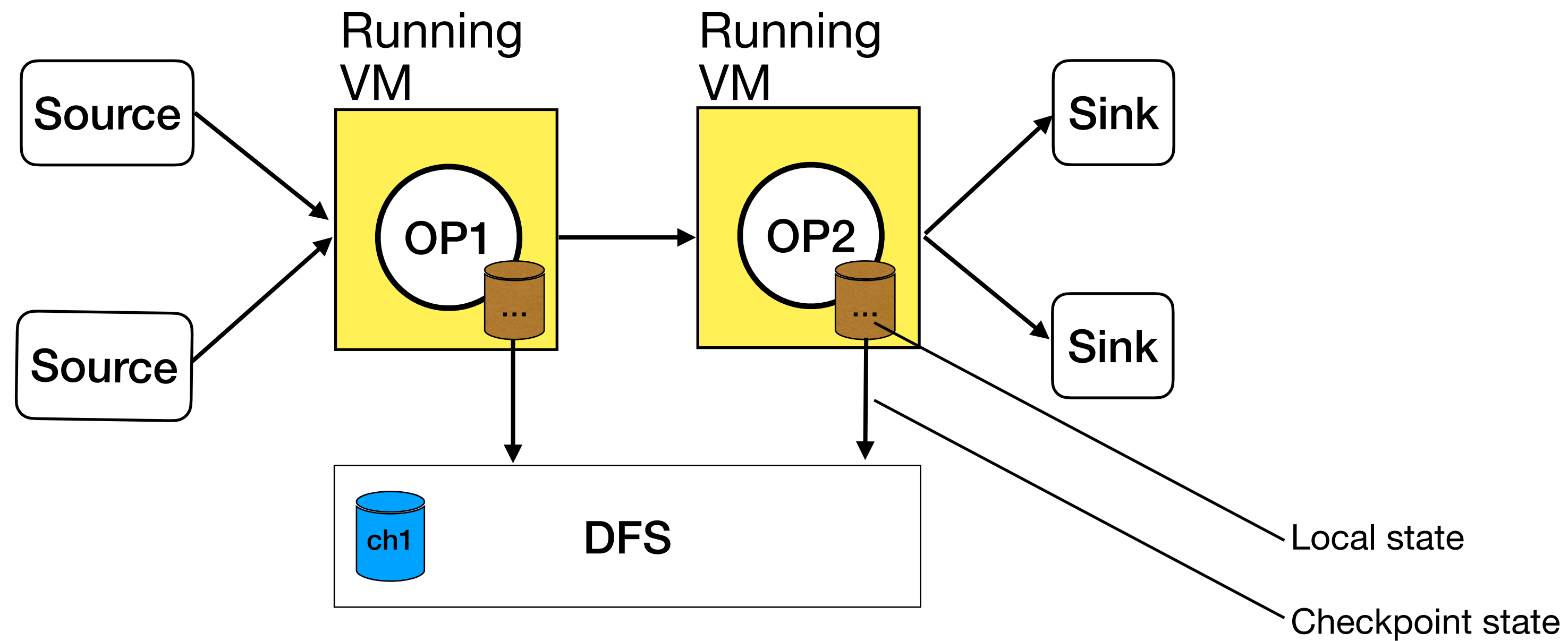


Passive standby

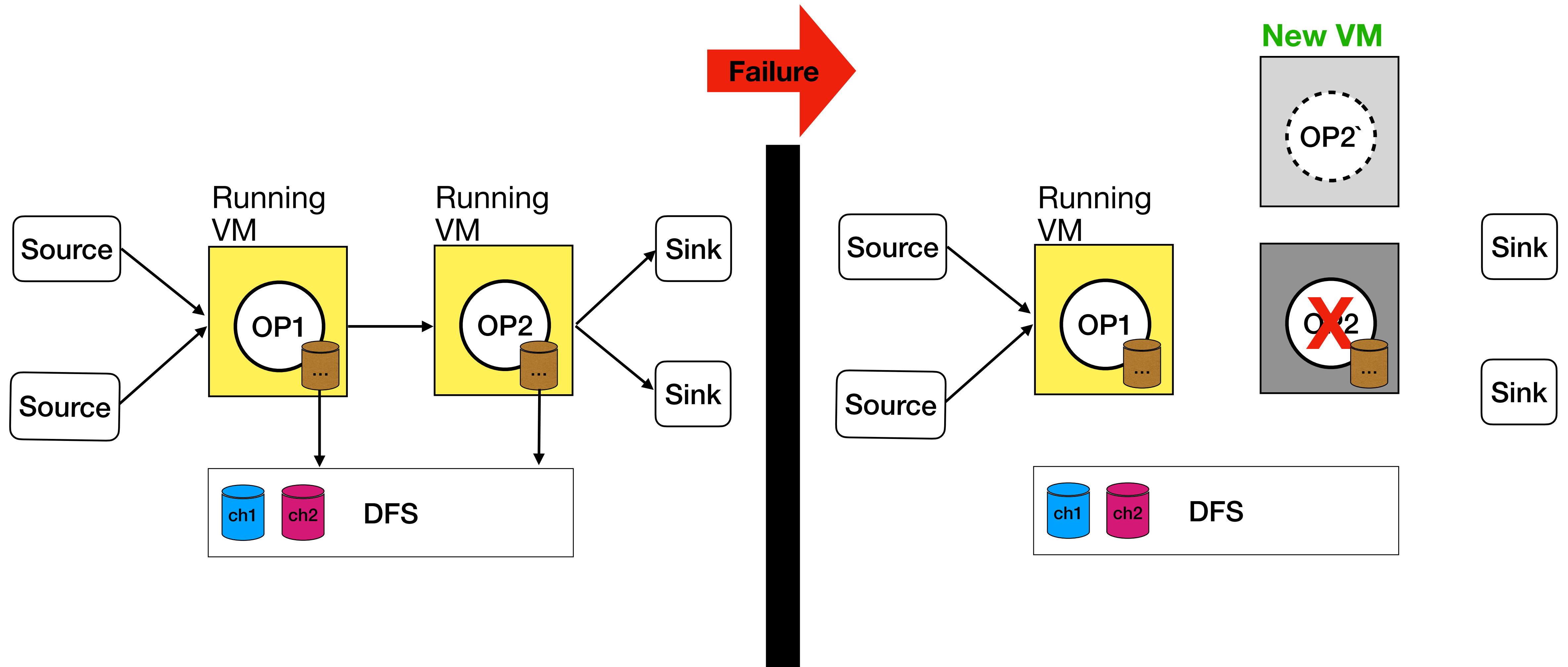
Replicating state

- Definition
 - Passive standby involves running a single instance of a computation and sending state updates of running nodes to replicated idle nodes
- Typical architectures
 - Optimized for the cloud (resource utilization)
 - Optimized for availability (recovery time)
- Challenges

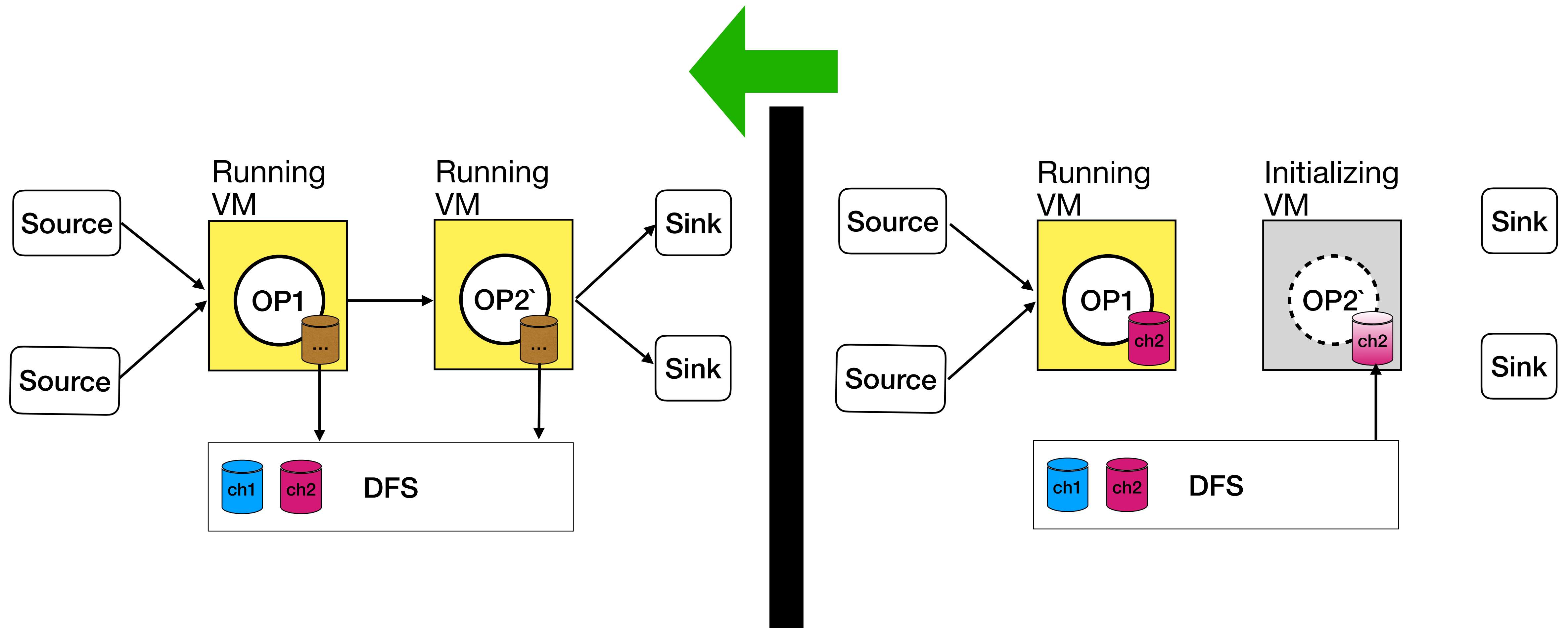
Cloud-optimized architecture



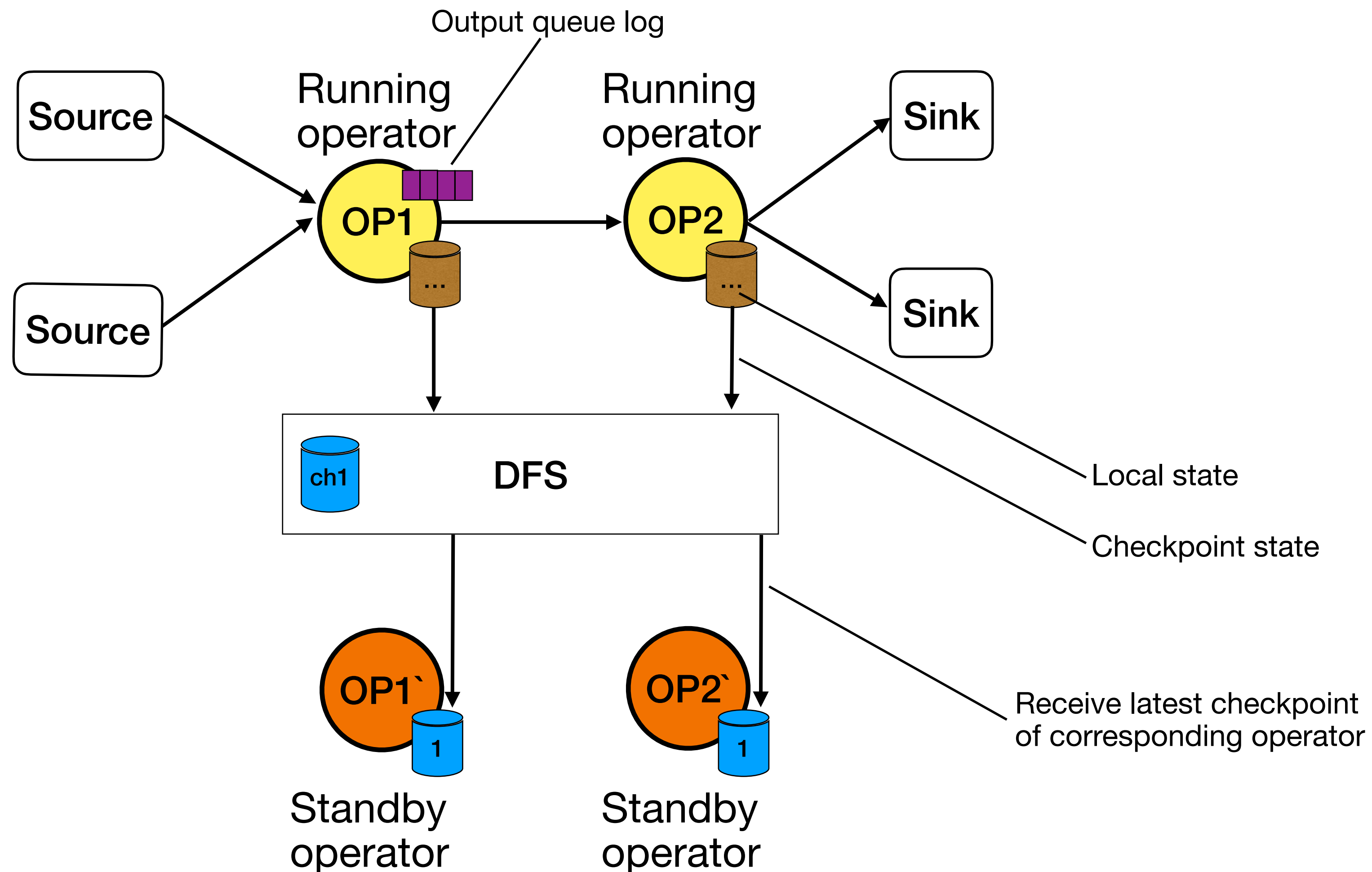
Cloud-optimized architecture



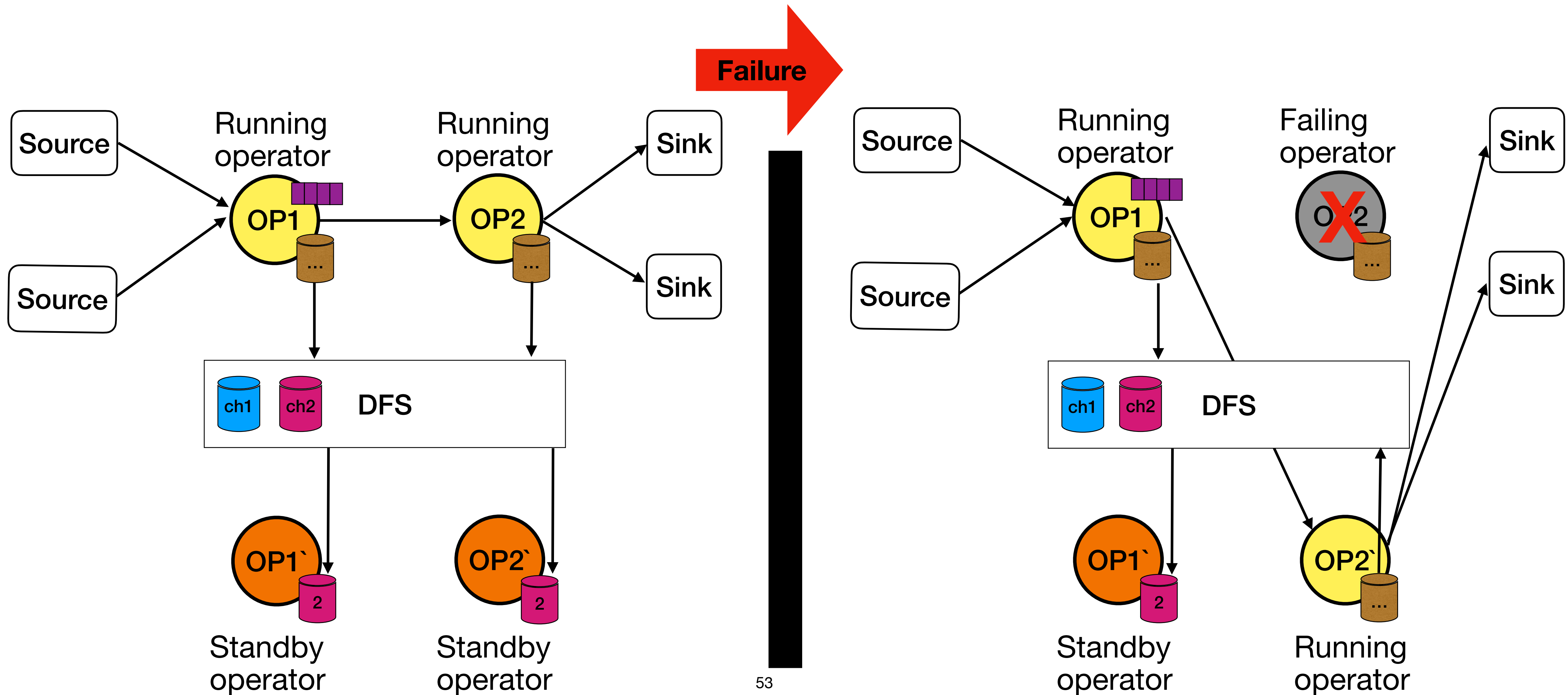
Cloud-optimized architecture



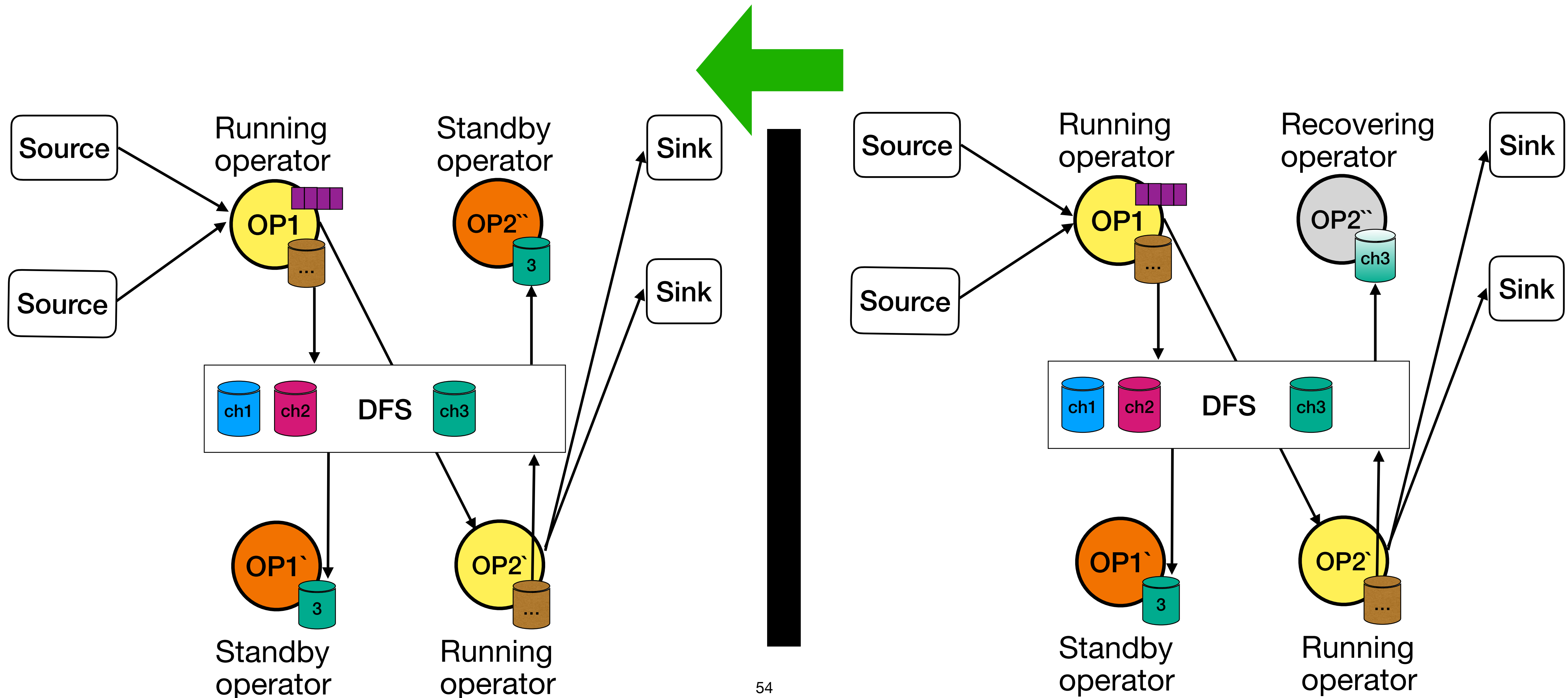
Availability-optimized architecture



High availability-optimized architecture



High availability-optimized architecture

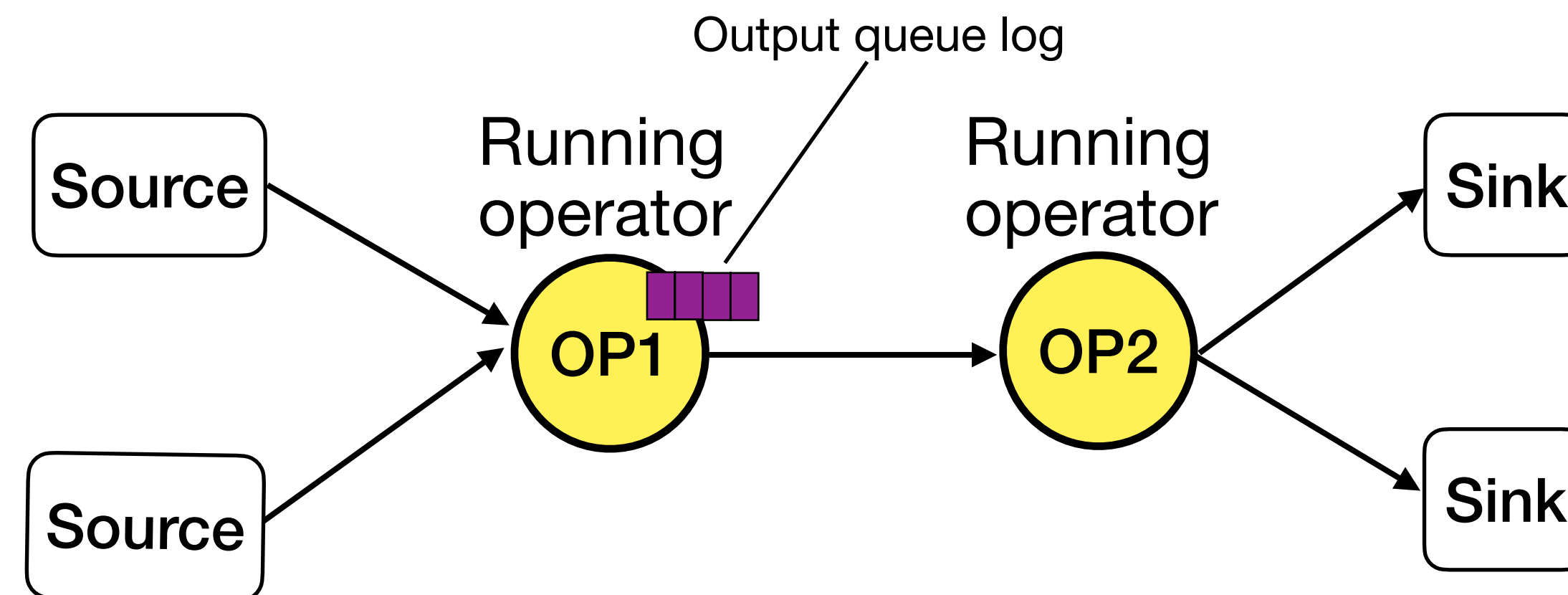


Challenges in passive standby

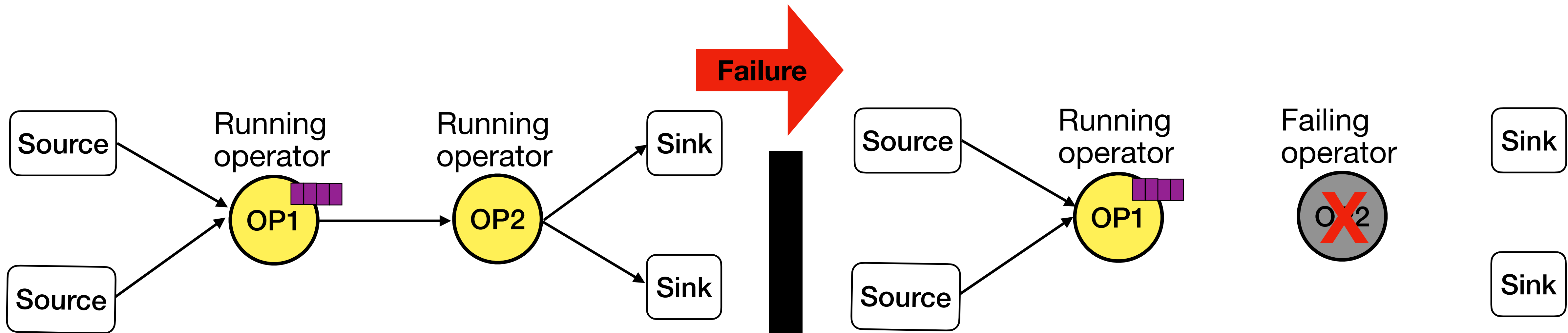
- State migration in cloud-based (on-demand) recovery
- Dynamic reconfiguration on-the-fly especially for the availability-optimised architecture
- Stay tuned for the “**Load management & Elasticity**” part with Vasia

Upstream backup

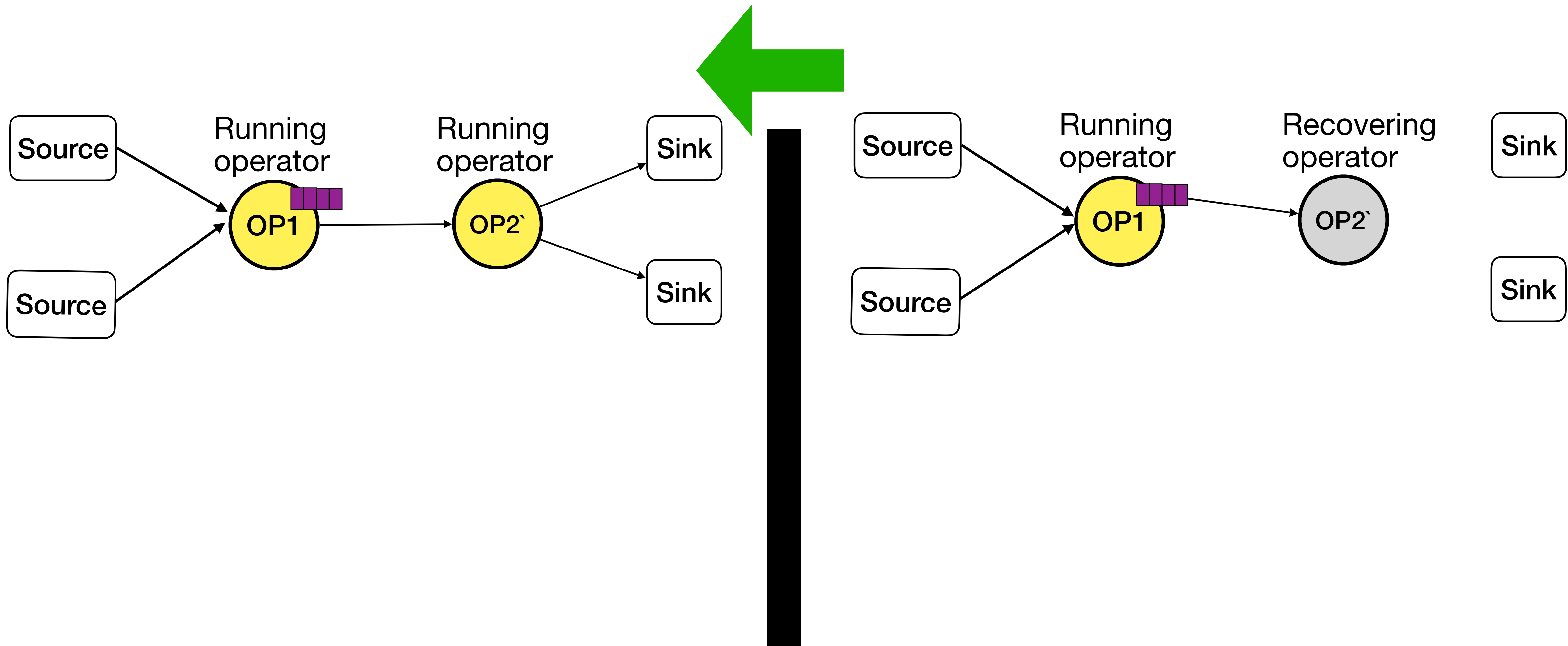
Replicating streams



Upstream backup



Upstream backup



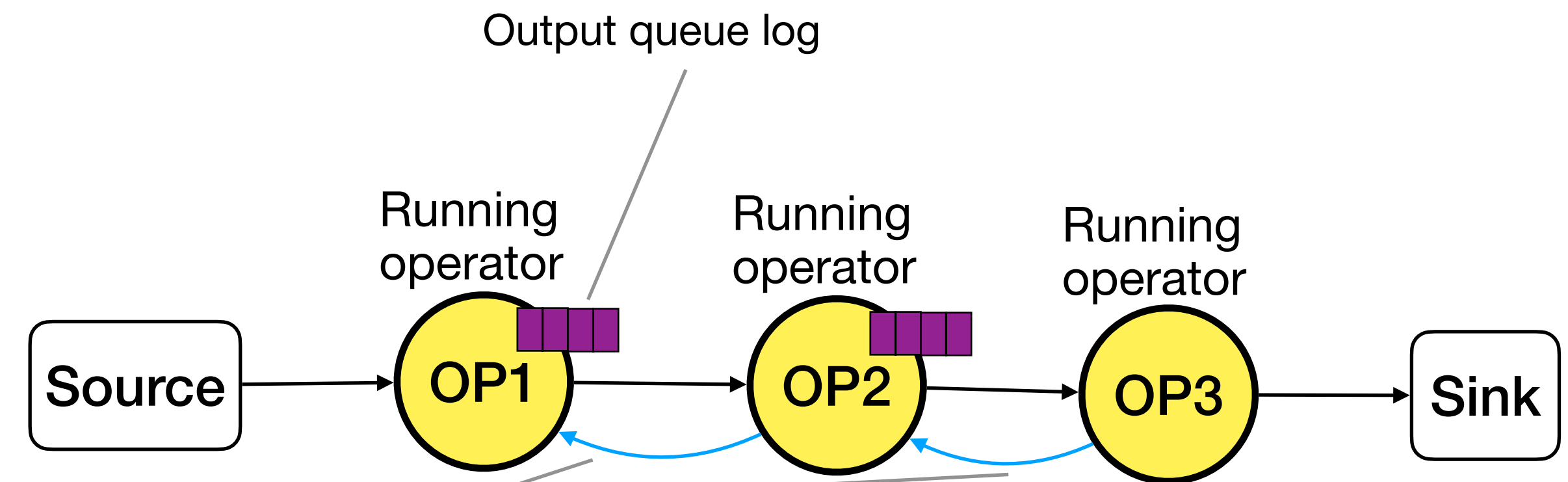
Challenges in upstream backup

- Discarding tuples in the output buffer queue of each upstream
 - Associating tuples with their effects

Discarding tuples in output buffer queues

- Discarding via **acknowledgements**

- Downstream operator *OP3* sends acknowledgement to upstream operator *OP2* on tuple receipt
- OP2* associates the acknowledged tuple to the tuple(s) that produced it and sends acknowledgements for those tuples to *OP1* by including input tuple ids to each output tuple. *OP1* removes the tuples from its output buffer queue



What is availability in data stream processing

Overview of current landscape

- Existing research relates system availability merely to failures in the spirit of what availability means for computer systems in general
- General definition
 - A system is available when it responds to requests with correct results (Gray et al)
- But streaming systems perform computations on potentially unbounded input
 - What does “respond with correct results” mean?

What is availability in data stream processing

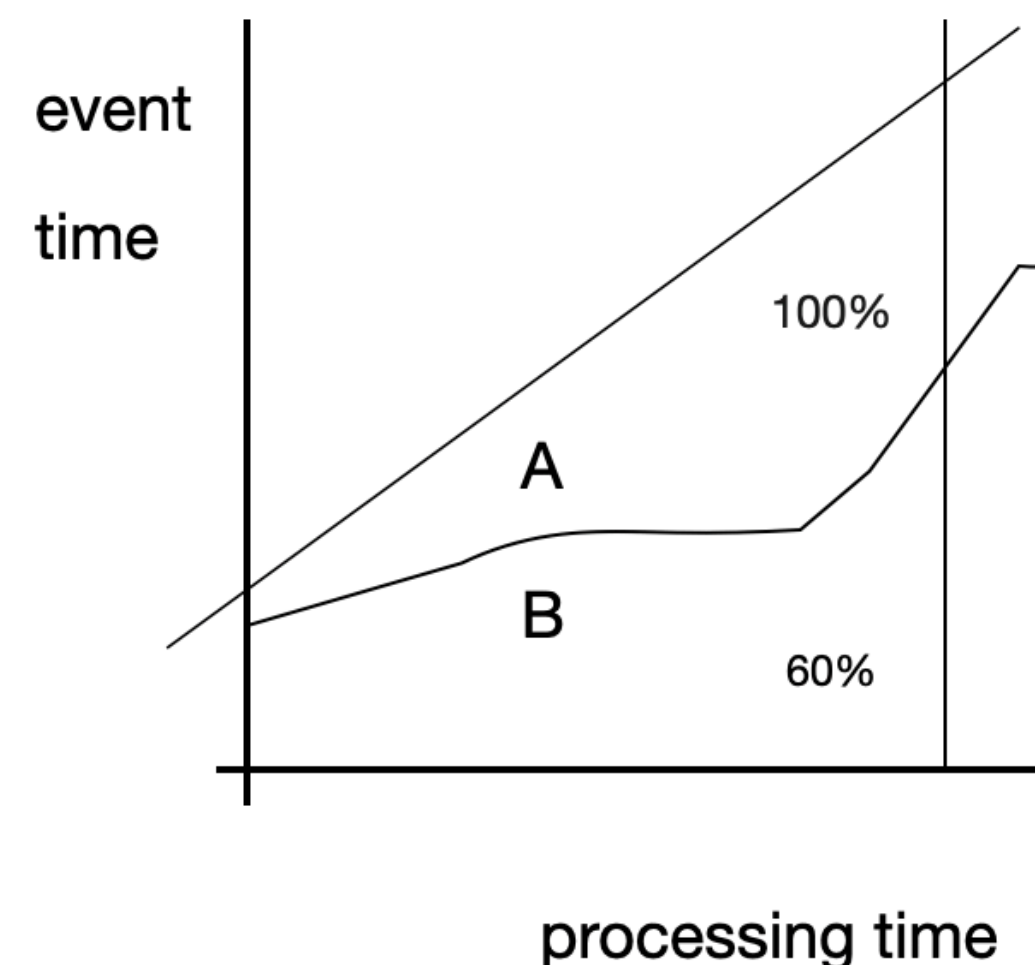
Causes of unavailability in stream processing

- Software and hardware failures
- Overload, which causes load shedding or back pressure
- Other types of processing interruption that makes the system fall behind input
 - Checkpoints and state migration
 - Garbage collection
 - Calls to external systems

What is availability in data stream processing

Proposed definition and measurement of availability in streaming

- **A streaming system is available when it can provide output based on the processing of its current input**
- Availability in streaming can be measured via **progress tracking mechanisms** (e.g. by the slack between processing time and event time), which show the system's processing progress with respect to the input



Slack over time (area in plot) = Availability SLA

Example: **A: 100% availability, B: 60%**

References

- J.Gray and D. P. Siewiorek. High-availability computer systems. *Computer*, 24(9):39–48, Sept. 1991.
- E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, Sept. 2002.
- M. A. Shah, J. M. Hellerstein, and E. Brewer. Highly available, fault-tolerant, parallel dataflows. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 827–838, New York, NY, USA, 2004. ACM.
- J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik. High-availability algorithms for distributed stream processing. In *ICDE*, 2005.
- T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. Millwheel: fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6(11):1033–1044, 2013.
- Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang. Timestream: Reliable stream computation in the cloud. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 1–14, New York, NY, USA, 2013. ACM.
- R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 725–736, New York, NY, USA, 2013. ACM.
- Matei Zaharia, Tathagata Das, Haoyuan Li, Tim Hunter, Scott Shenker, and Ion Stoica. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In *SOSP*, 2013.
- R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Making state explicit for imperative big data processing. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, page 49–60, USA, 2014. USENIX Association.
- P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache Flink: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38:28–38, 2015.
- W. Lin, H. Fan, Z. Qian, J. Xu, S. Yang, J. Zhou, and L. Zhou. Streamscope: Continuous reliable distributed processing of big data streams. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, pages 439–453, Berkeley, CA, USA, 2016. USENIX Association.
- Gabriela Jacques-Silva, Fang Zheng, Daniel Debrunner, Kun-Lung Wu, Victor Dogaru, Eric Johnson, Michael Spicer, and Ahmet Erdem Sariyüce. 2016. Consistent regions: guaranteed tuple processing in IBM streams. *Proc. VLDB Endow.* 9(13): 1341–1352, Sep. 2016.
- P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas. State management in Apache Flink: Consistent stateful distributed stream processing. In *VLDB*, 2017.
- M. Armbrust, T. Das, J. Torres, B. Yavuz, S. Zhu, R. Xin, A. Ghodsi, I. Stoica, and M. Zaharia. Structured streaming: A declarative API for real-time applications in Apache Spark. In *SIGMOD*, 2018.
- S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringham, I. Gupta, and R. H. Campbell. Samza: Stateful scalable stream processing at Linkedin. In *VLDB*, 2017.
- B.Chandramouli and J. Goldstein. Shrink: Prescribing resiliency solutions for streaming. *Proc. VLDB Endow.*, 10(5):505–516, Jan. 2017.

Thanks for viewing! And don't forget:

1. Fault recovery largely decides the consistency of a streaming system
2. There are multiple levels of consistency; the strictest is described by the output commit problem and there are a few ways to solve it
3. High availability relies on replication
4. Availability in streaming should relate to the processing progress with respect to the input