The Modern Stream Processor: A View from Apache Flink

Tzu-Li (Gordon) Tai, Senior Software Engineer
Apache Flink PMC / Committer
@tzulitai

ververica

About me

- **PMC / Committer at Apache Flink**
  - Exactly-once streaming connectors (Kafka, Kinesis)
  - State-related things (evolvability of stateful streaming applications, state backends, etc.)

- **Software Engineer at Ververica**
  - Work full-time developing open source for Apache Flink

About Ververica

Original creators of Apache Flink®

Complete Stream Processing Infrastructure

Powered by Apache Flink

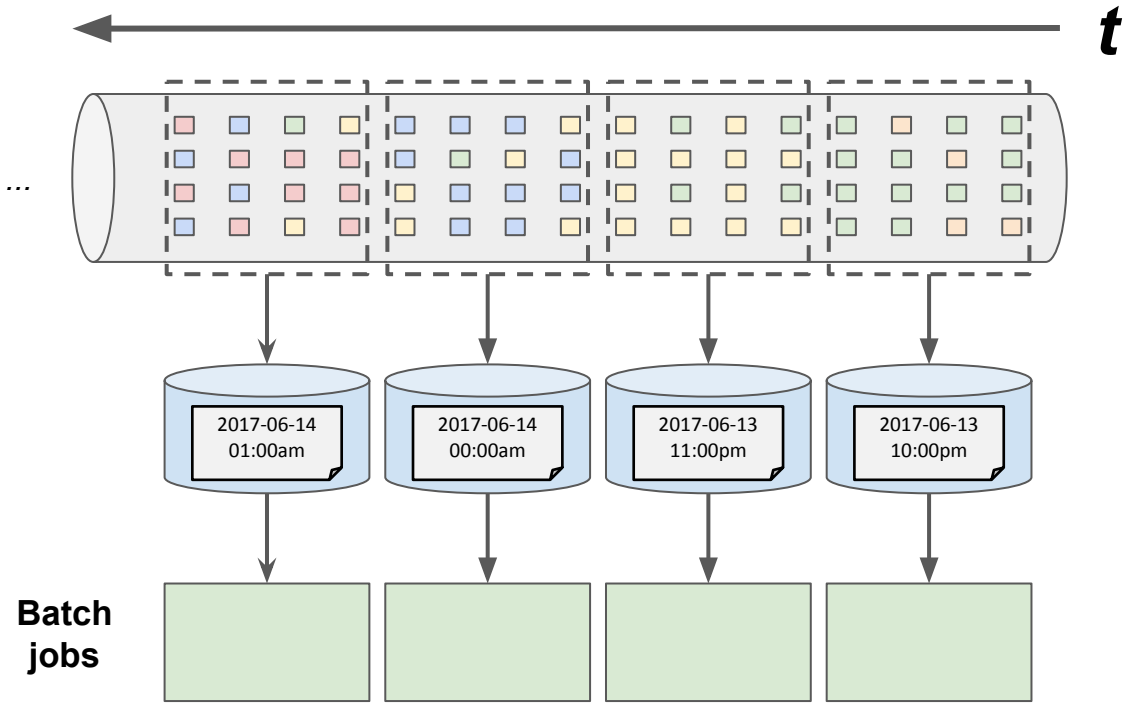
 |

What we will talk about

- What is stateful stream processing and how it has evolved over time
- Apache Flink’s take on stateful stream processing
  - Core features and internal designs that make it work

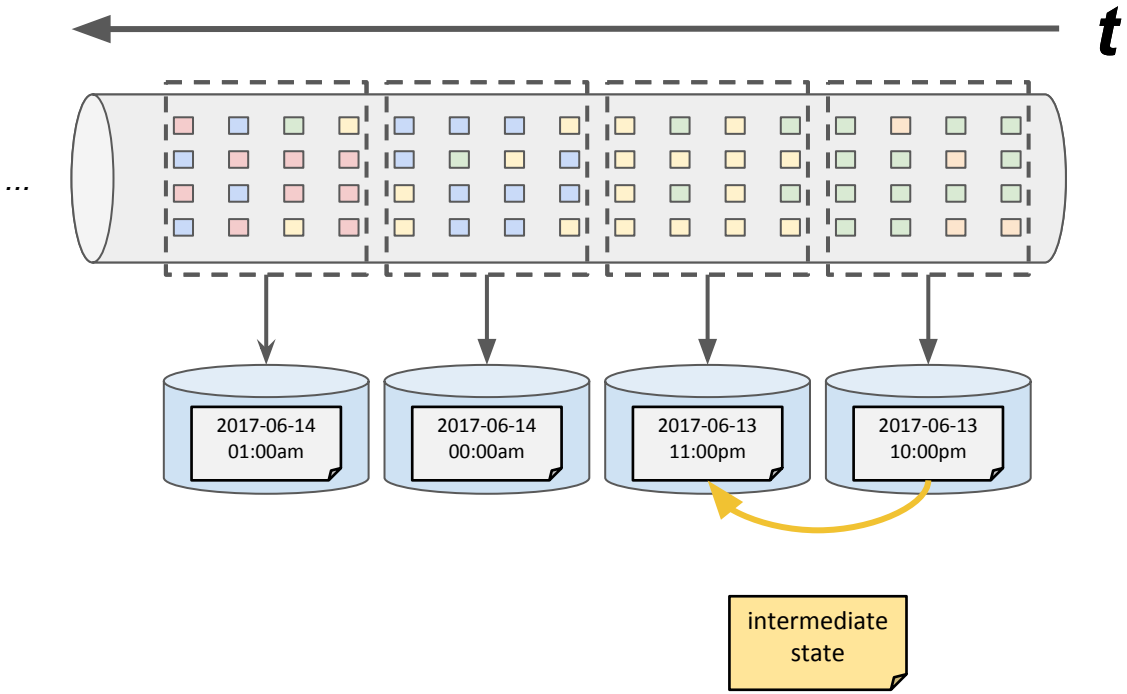
What is Stateful Stream Processing?

The traditional batch way ...



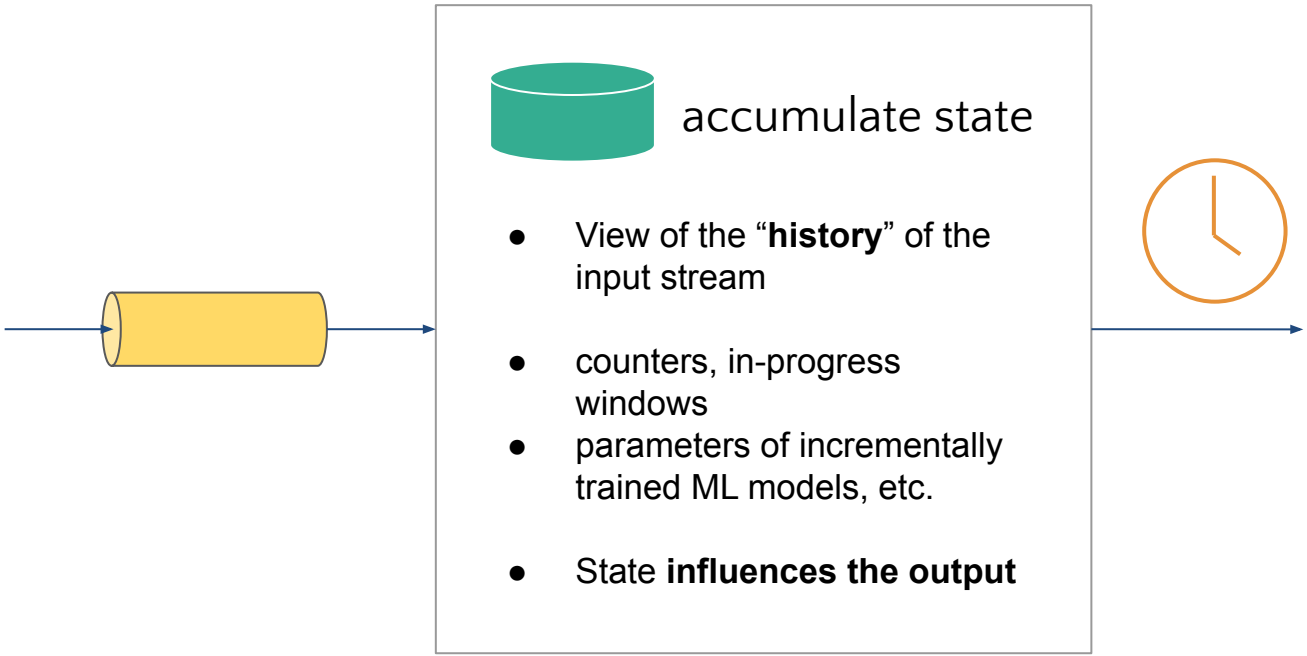
- Continuously ingesting data
- Time-bounded batch files
- Periodic batch jobs

The traditional batch way …



- Consider computing conversion metric **(# of A → B per hour)**
- What if the conversion crossed time boundaries?
  → carry intermediate results to next batch
- What if events come out of order?

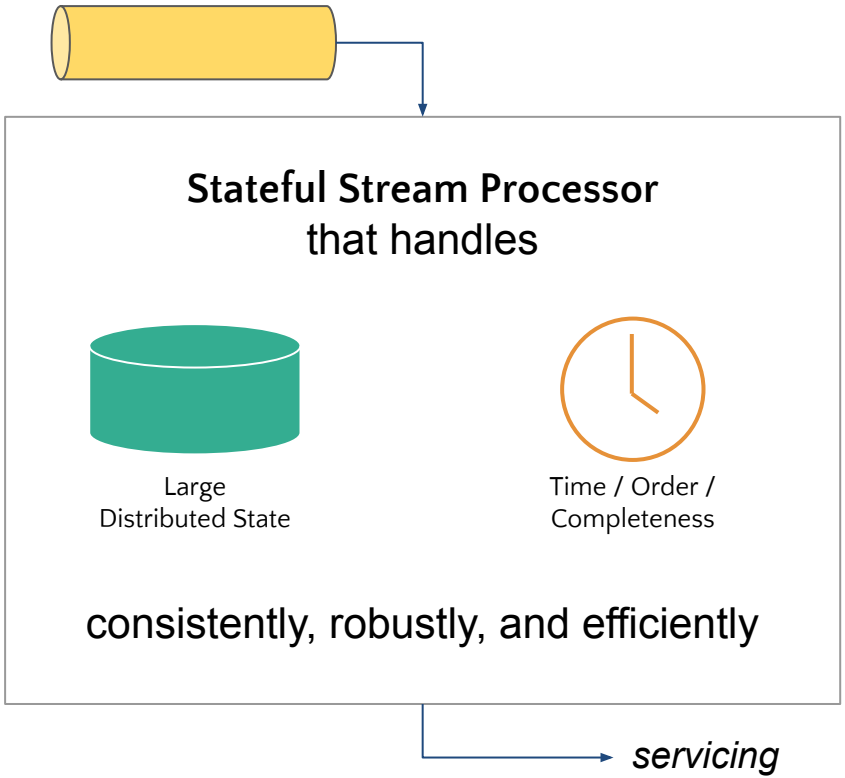
The ideal way



long running computation

- Output depending on **notions of time**
- Outputs **when results are complete**.

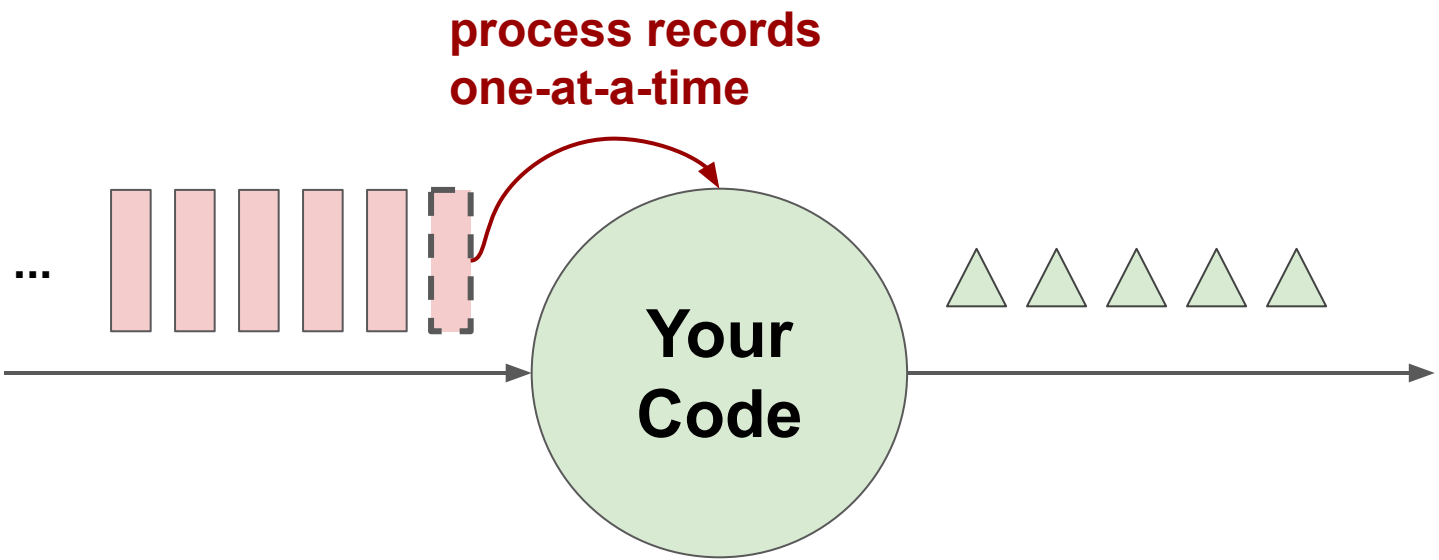
The ideal way



- Stateful stream processing as a **new paradigm** to **continuously process continuous data**
- Produce **accurate** results
- Results available in real-time is only a natural consequence of the model

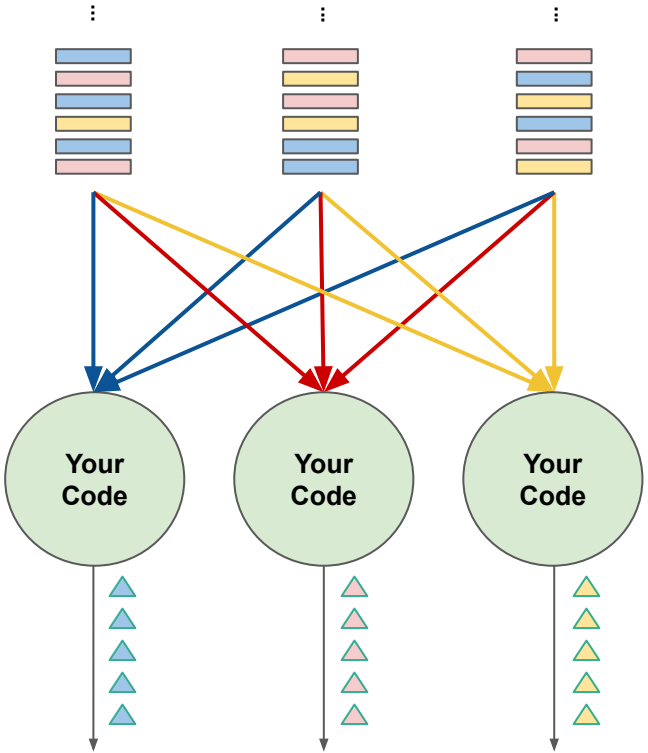
Apache Flink's take on Stateful Stream Processing

Stream Processing



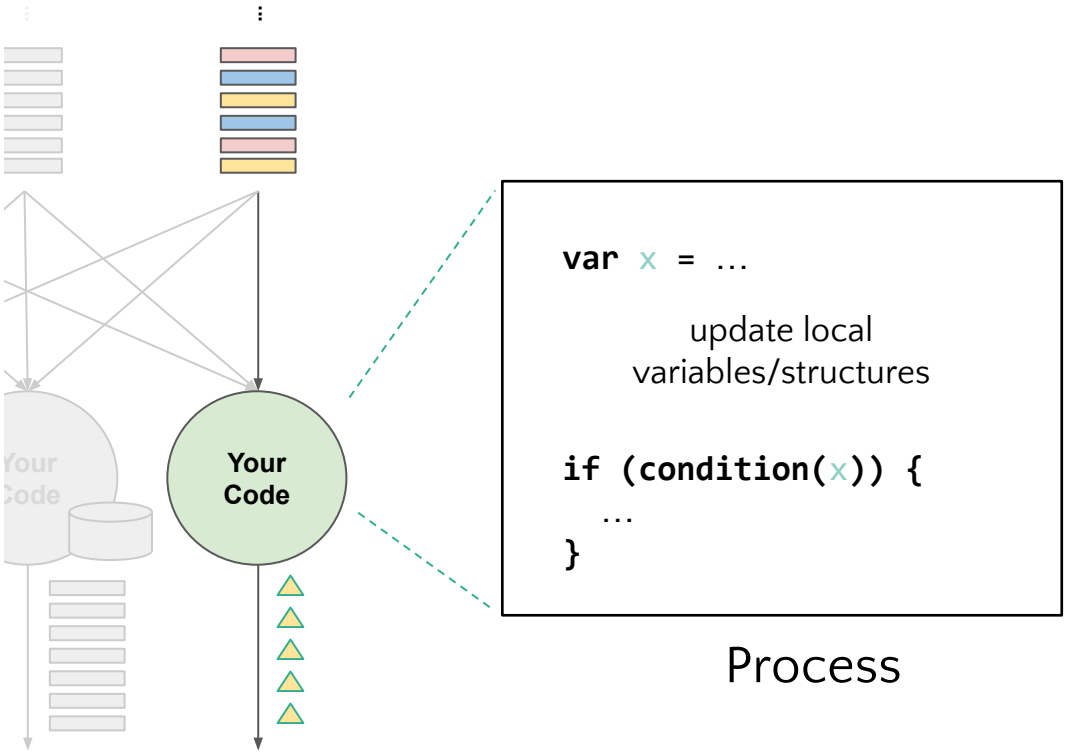
Long running computation, on an endless stream of input

Distributed Stream Processing

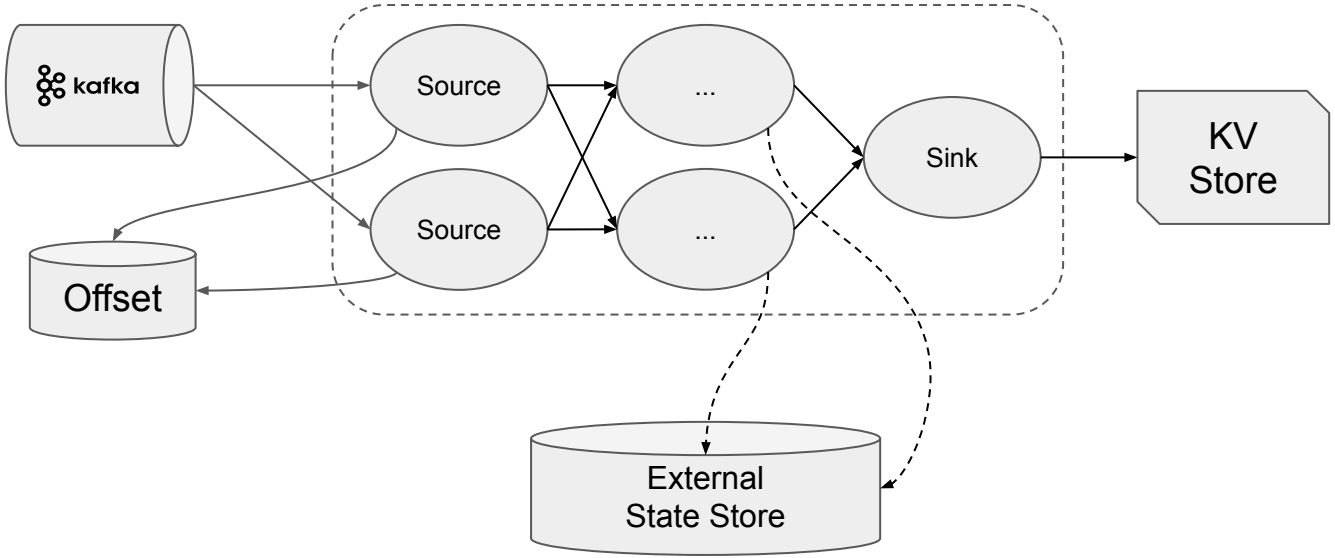


- partitions input streams by some key in the data
- distributes computation across multiple instances
- Each instance is responsible for some key range

Stateful Stream Processing

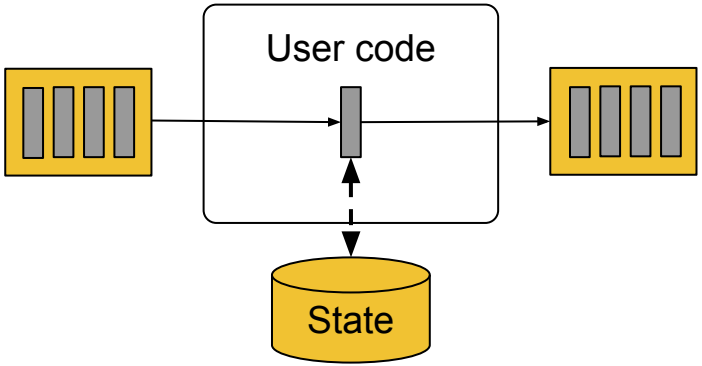


External State (with Stateless Stream Processors)



 |

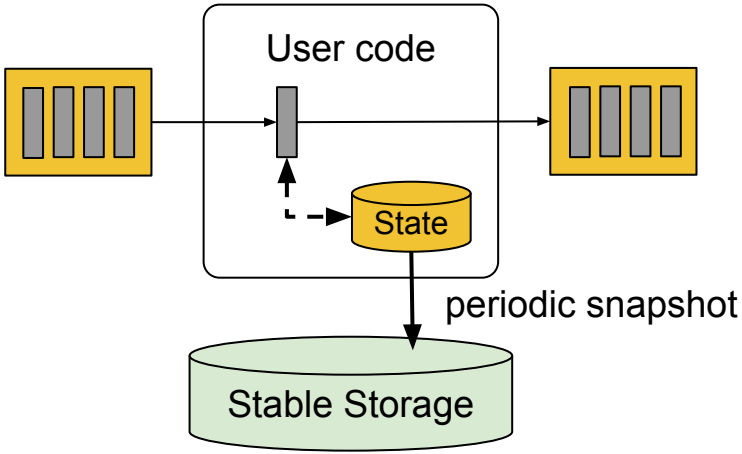
External State v.s. Internal State



External State

- State in a separate data store
- Usually much slower due to remote read / writes
- Fault-tolerance / scalability is responsibility
  of the external storage
- Not that easy to get exactly-once guarantees

External State v.s. Internal State



Internal State

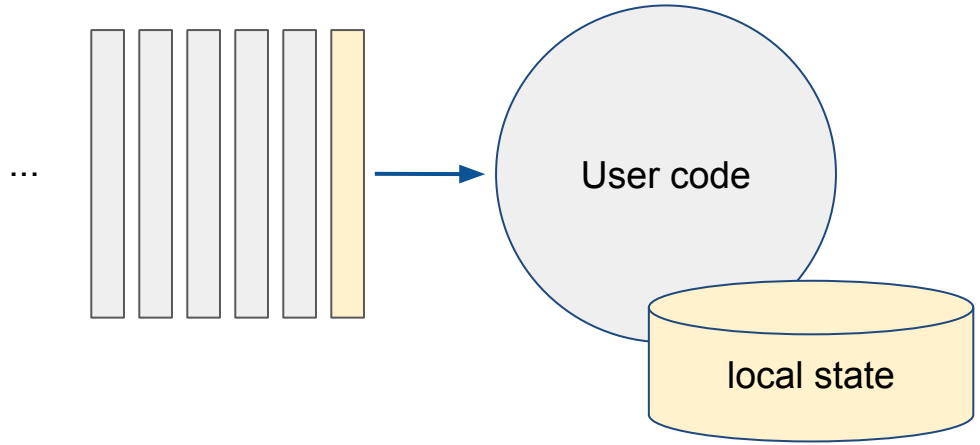
- State within the stream processor
- Local read / writes
- Snapshots to stable store (DFS)
- Always exactly-once consistent
- Stream processor needs to handle large state / rescaling / fault-tolerance

Fault Tolerance for Internal State

Fault tolerance concerns for a stateful stream processor:

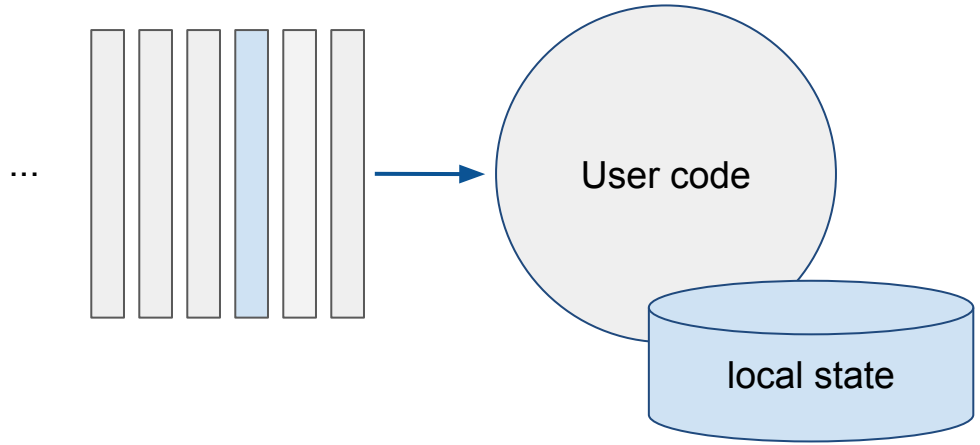
- How to ensure exactly-once semantics for the state?

Fault tolerance: simple case



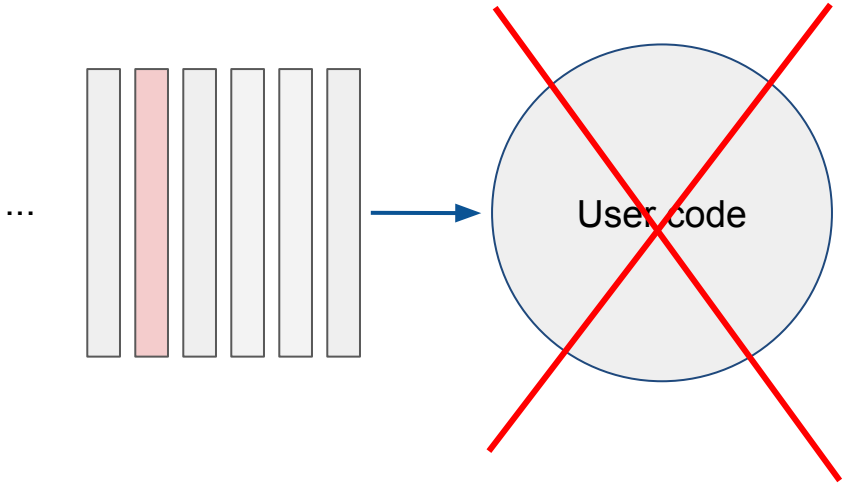
periodically snapshot the state + event log position

Fault tolerance: simple case



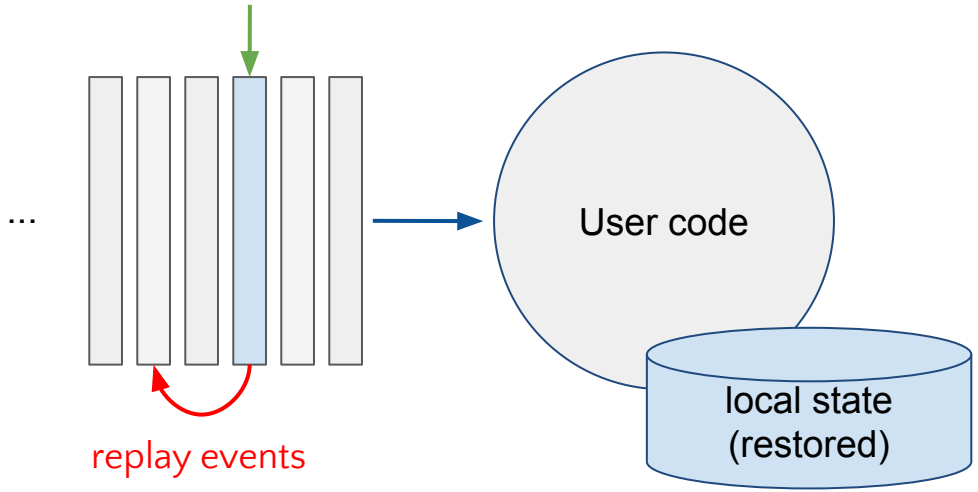
periodically snapshot the state + event log position

Fault tolerance: simple case



 |

Fault tolerance: simple case



Recovery: restore snapshot and replay events since snapshot

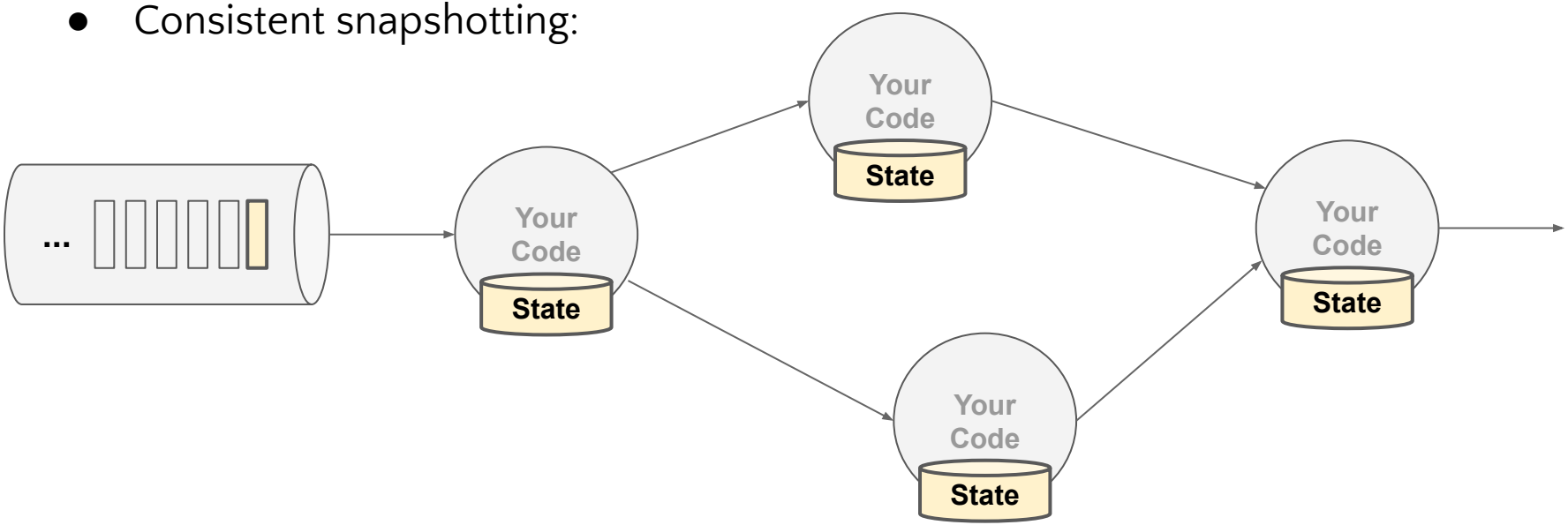
Fault Tolerance for Internal State

Fault tolerance concerns for a stateful stream processor:

- How to ensure exactly-once semantics for the state?
- How to create consistent snapshots of **distributed embedded state**?
- More importantly, how to do it **efficiently without abrupting computation**?

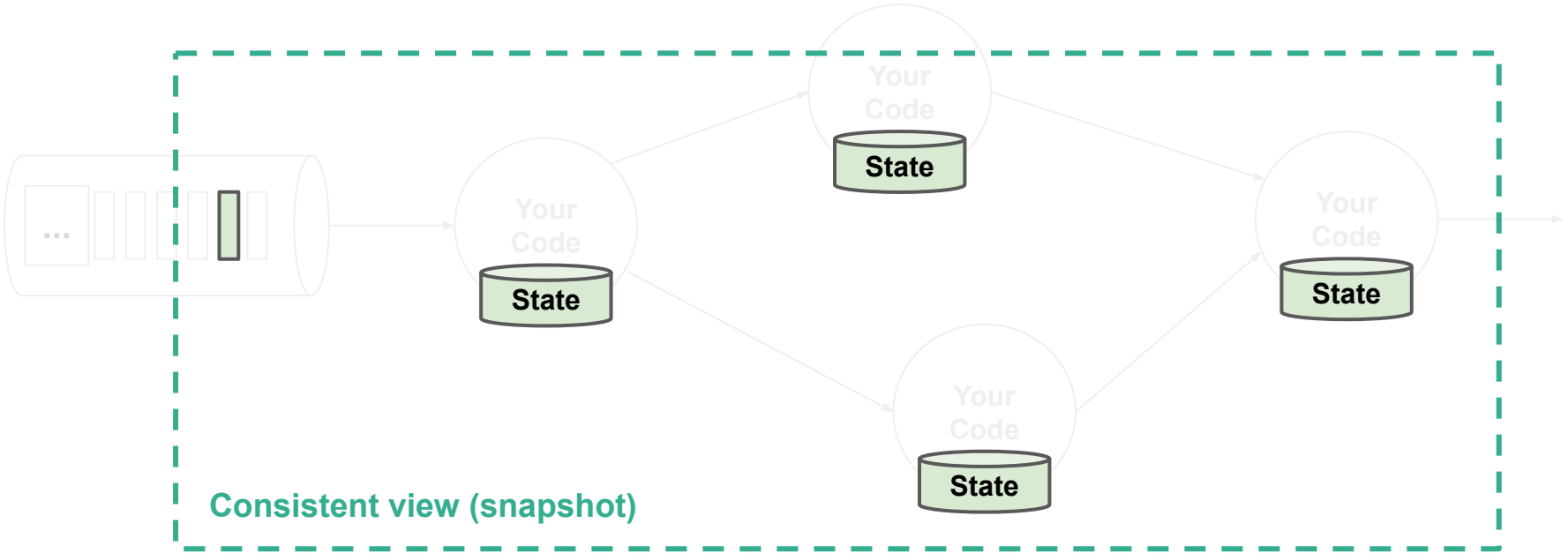
Fault tolerance: distributed state

- Consistent snapshotting:



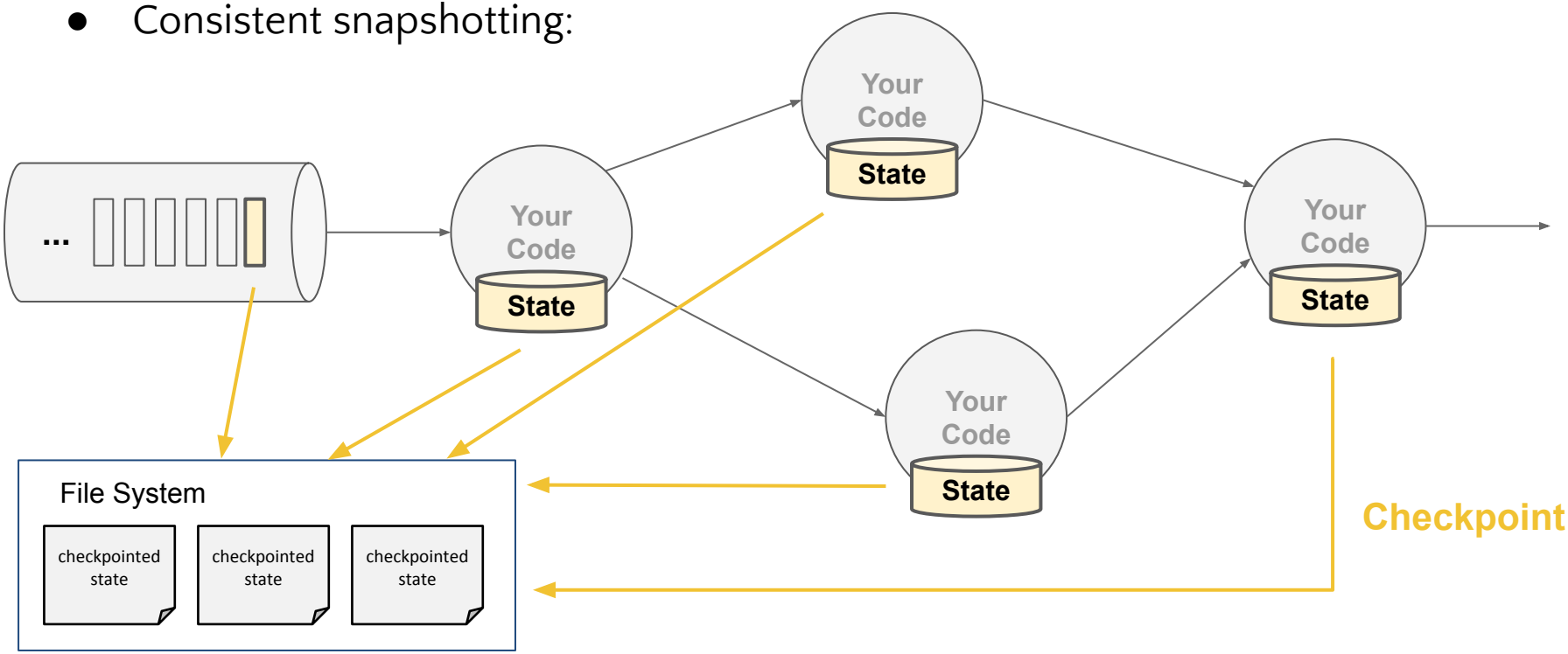
 |

Fault tolerance: distributed state



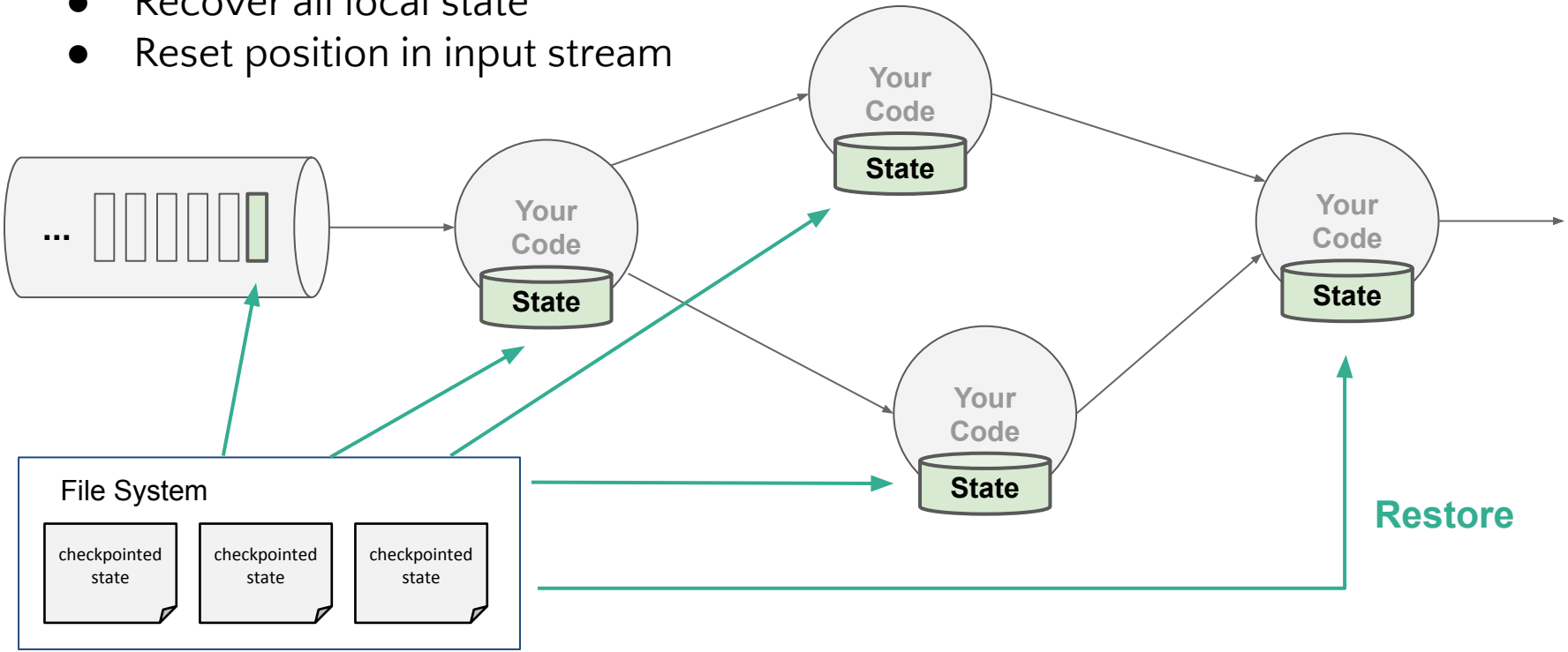
Fault tolerance: distributed state

- Consistent snapshotting:



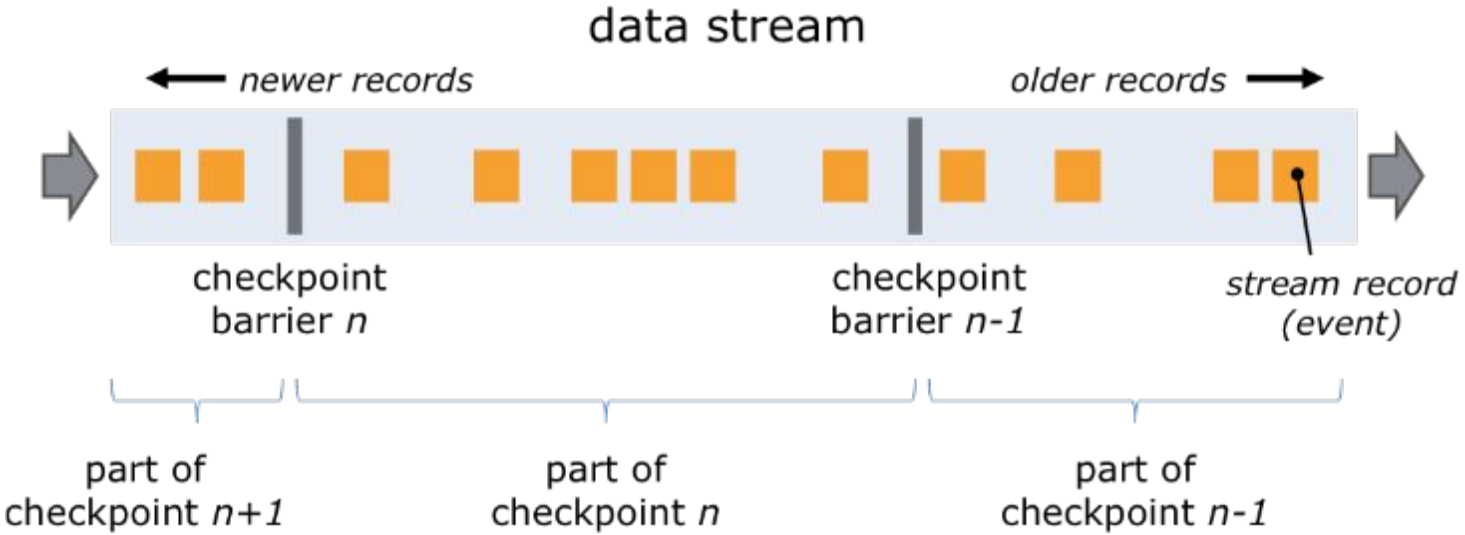
Fault tolerance: distributed state

- Recover all local state
- Reset position in input stream

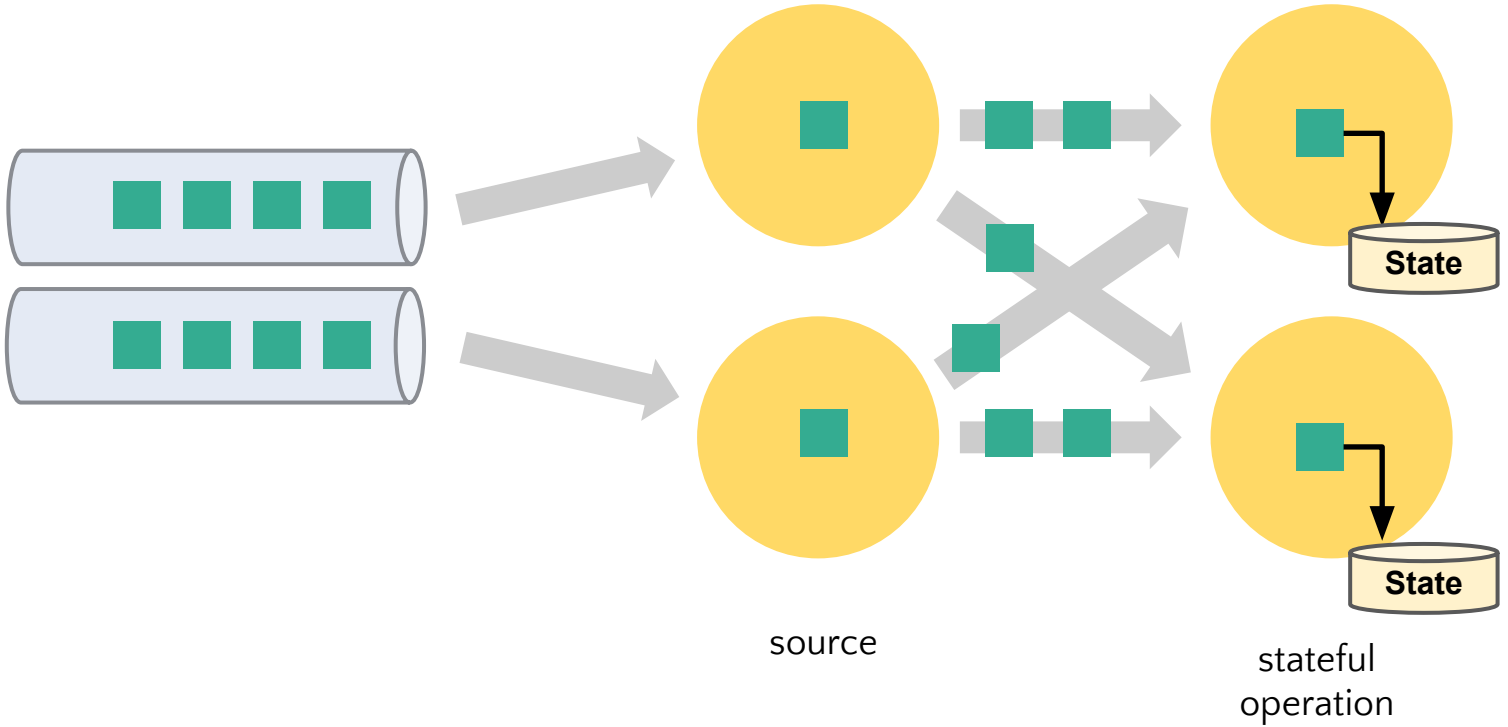


Distributed Snapshots

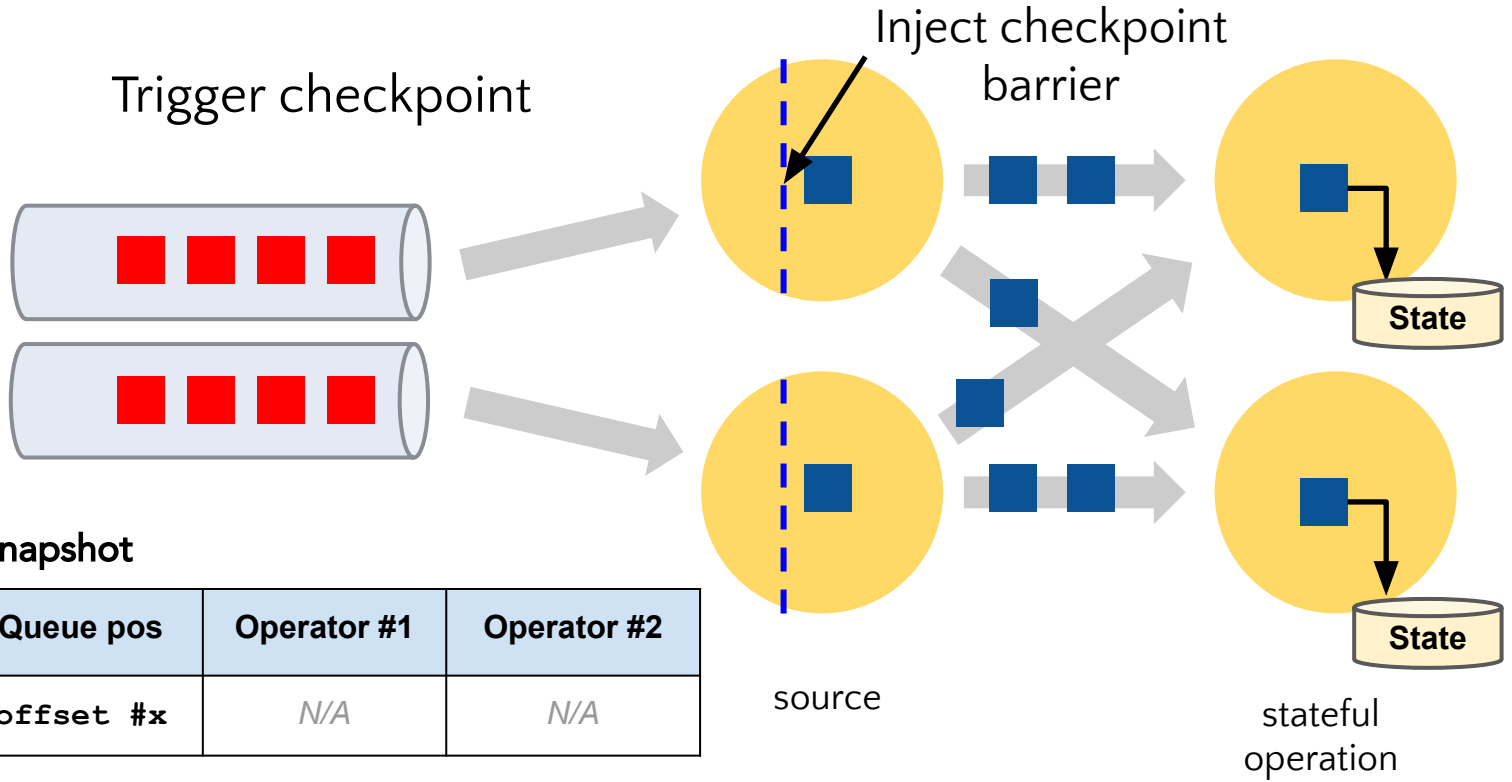
Coordination via “asynchronous checkpoint barriers”, injected into the streams



Distributed Snapshots



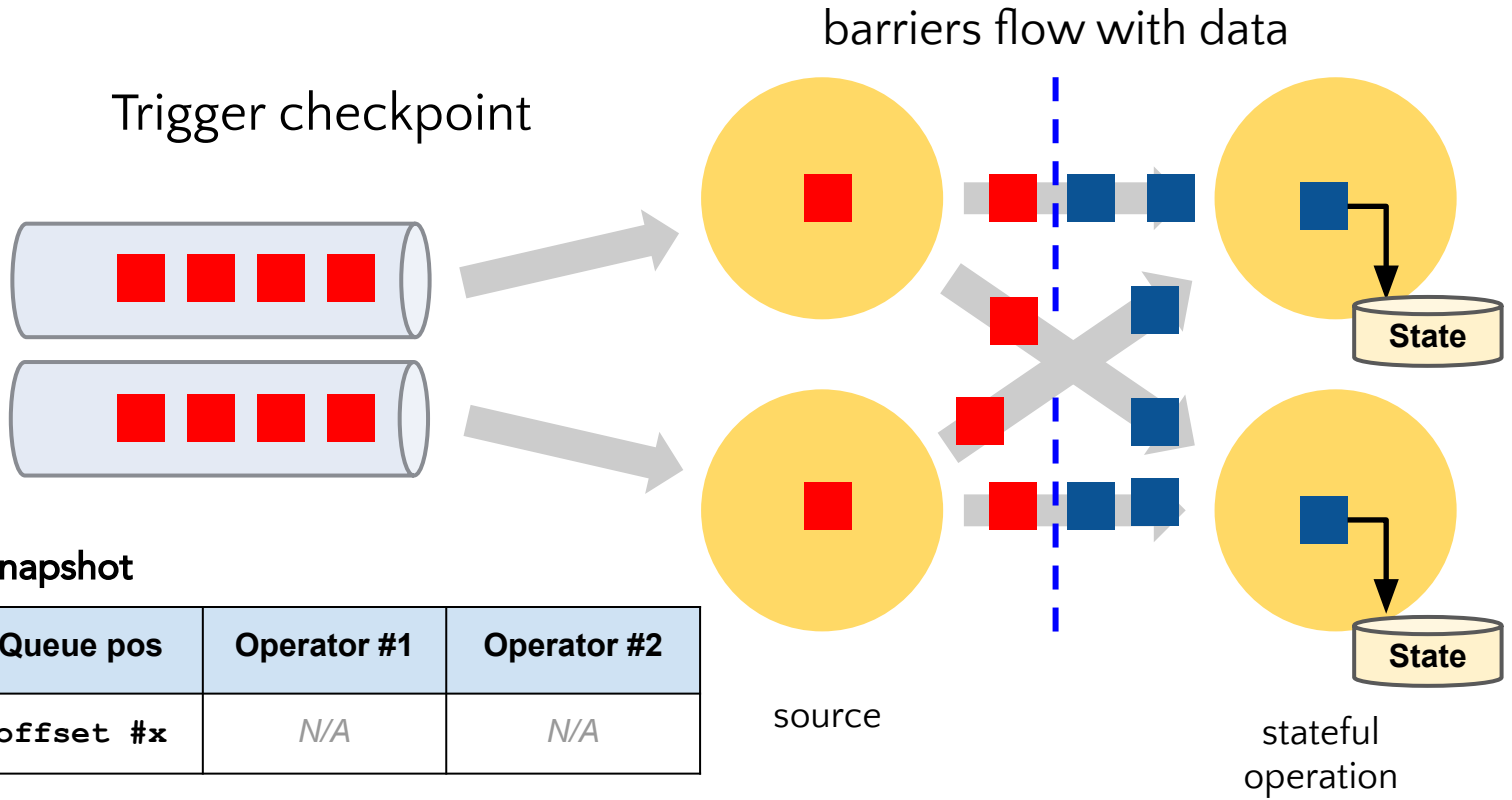
Distributed Snapshots



Snapshot

Queue pos	Operator #1	Operator #2
`offset #x`	*N/A*	*N/A*

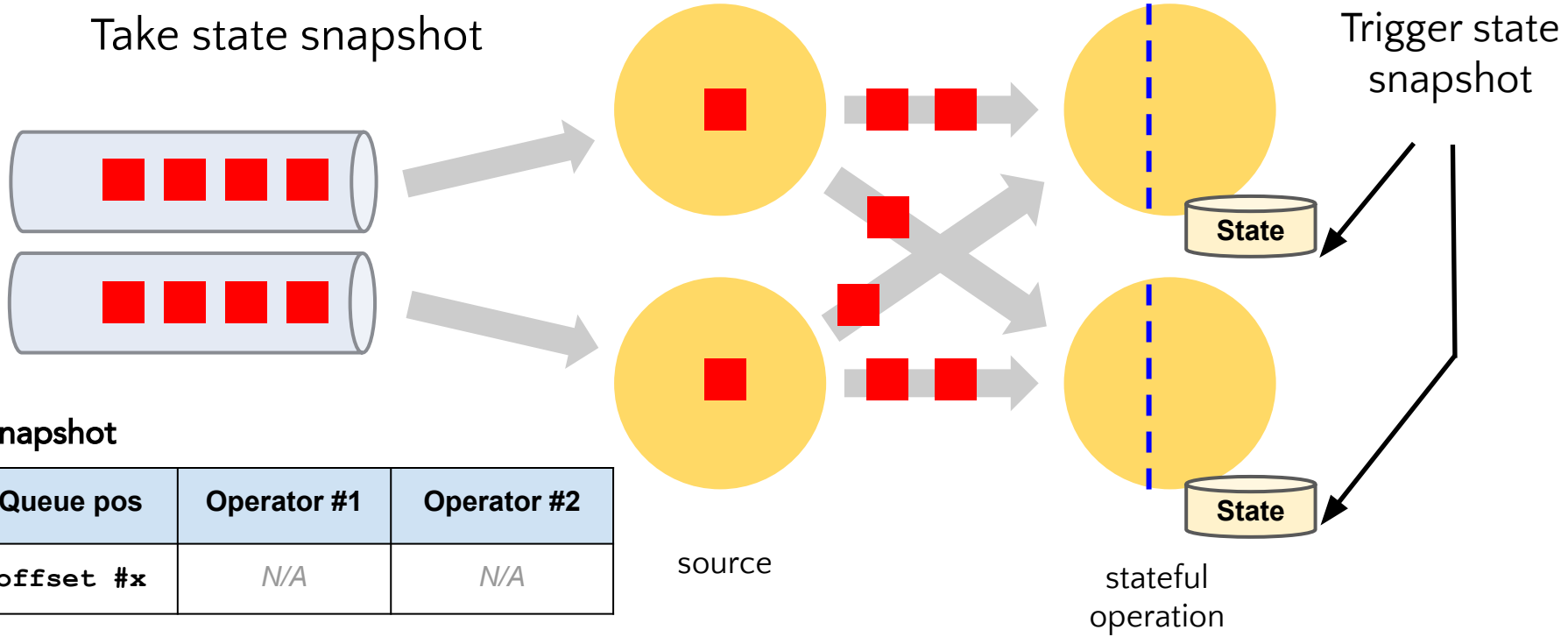
Distributed Snapshots



Snapshot

Queue pos	Operator #1	Operator #2
`offset #x`	*N/A*	*N/A*

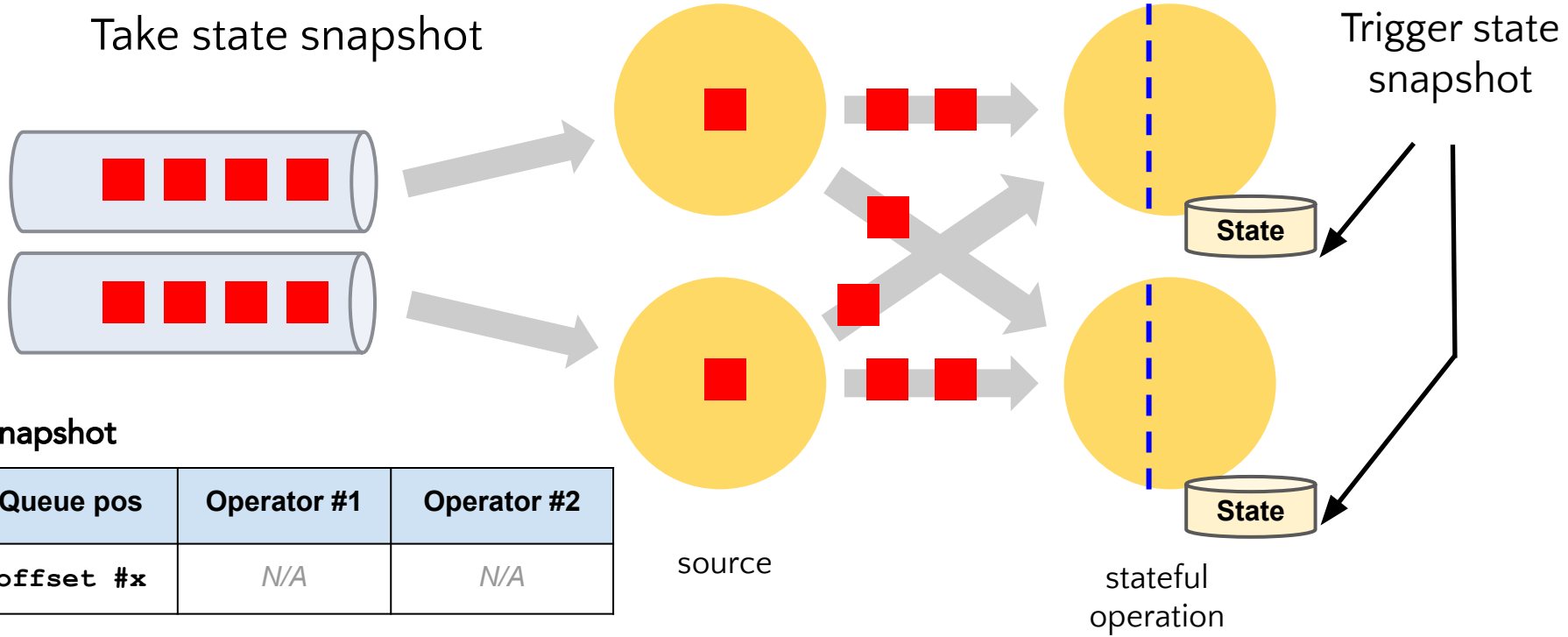
Distributed Snapshots



Snapshot

Queue pos	Operator #1	Operator #2
`offset #x`	*N/A*	*N/A*

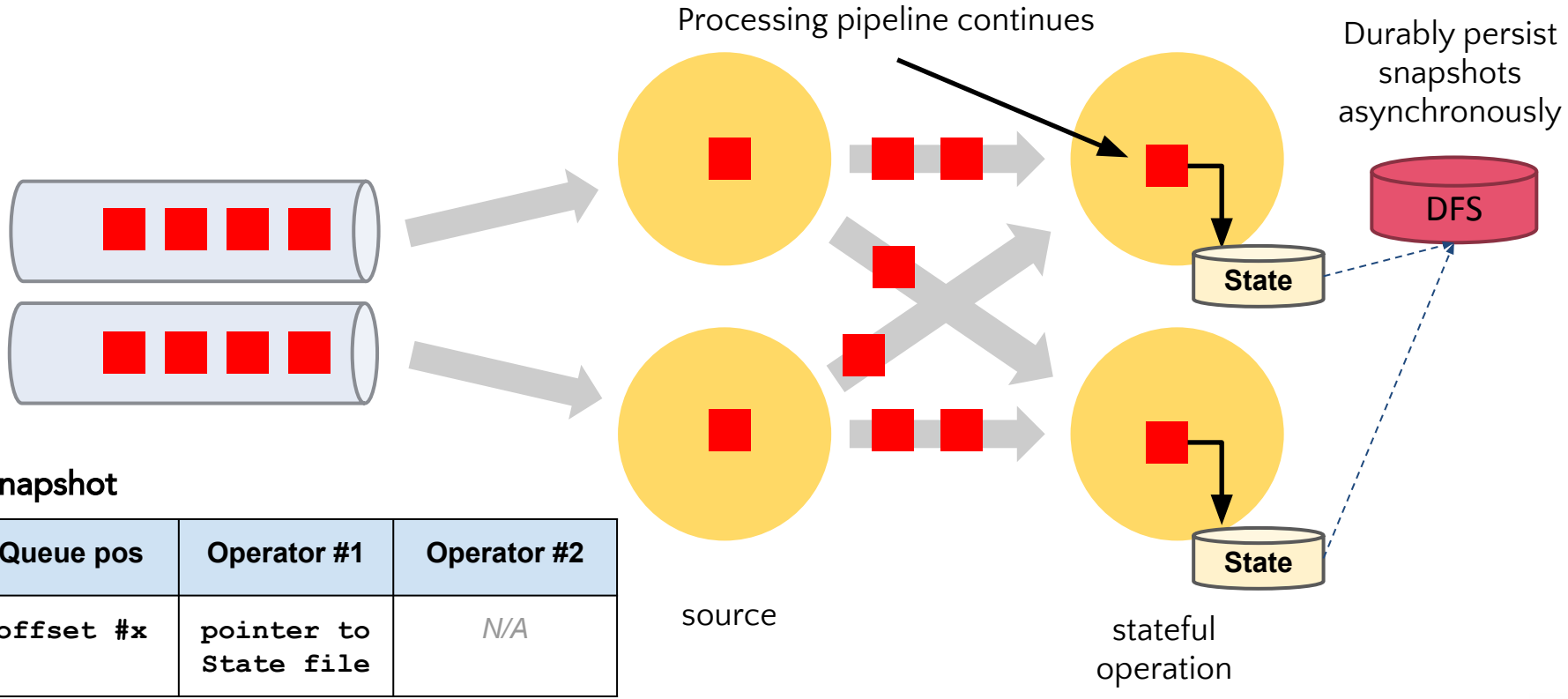
Distributed Snapshots



Snapshot

Queue pos	Operator #1	Operator #2
`offset #x`	*N/A*	*N/A*

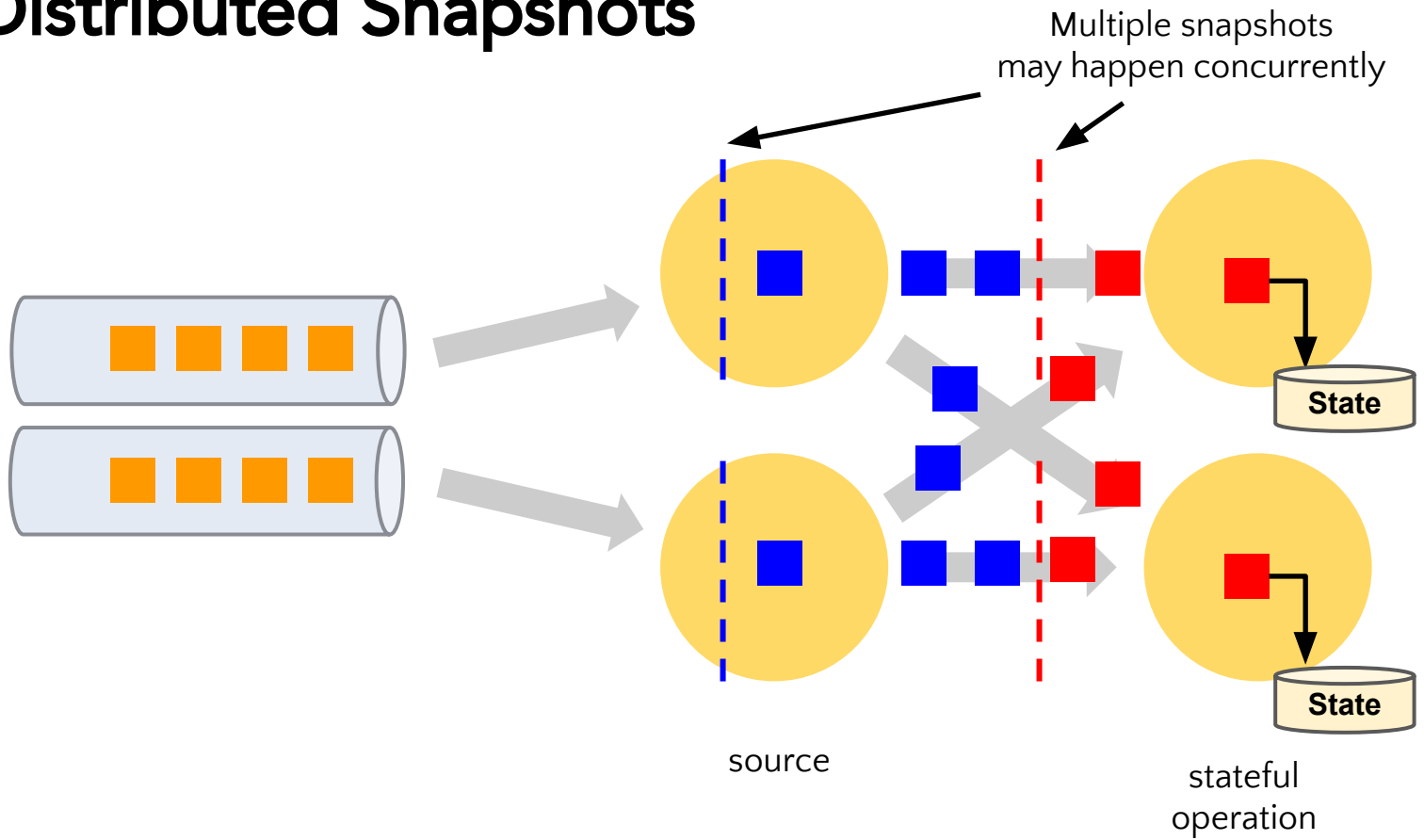
Distributed Snapshots



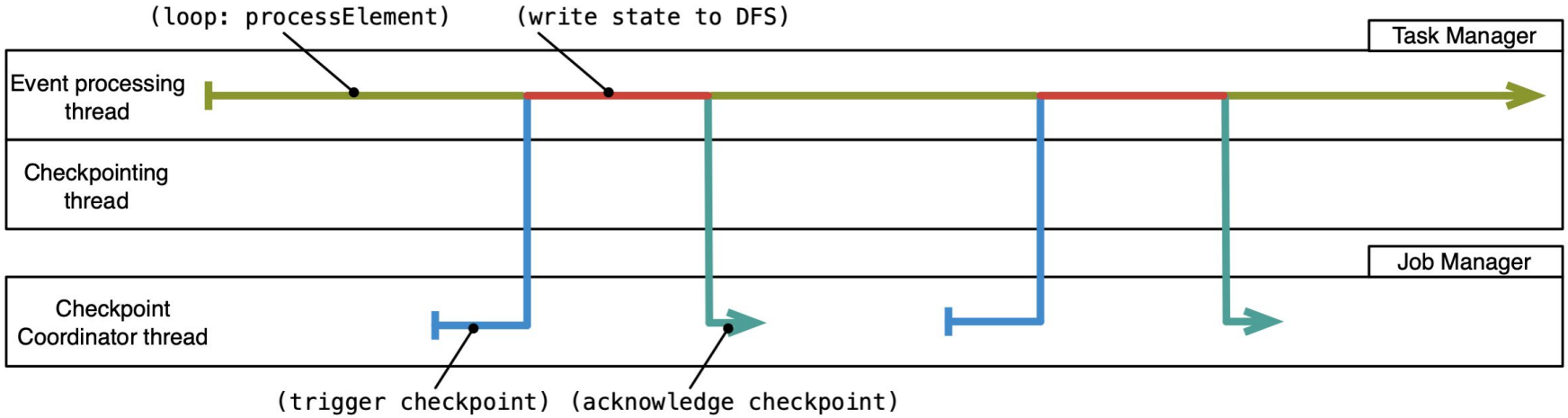
Snapshot

Queue pos	Operator #1	Operator #2
`offset #x`	`pointer to State file`	*N/A*

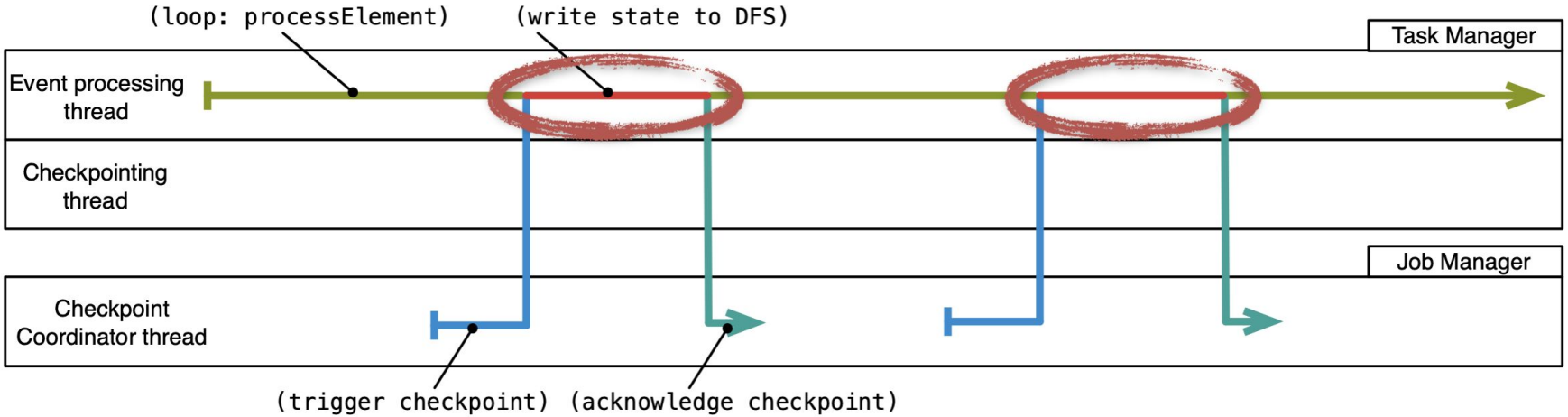
Distributed Snapshots



Synchronous Checkpointing

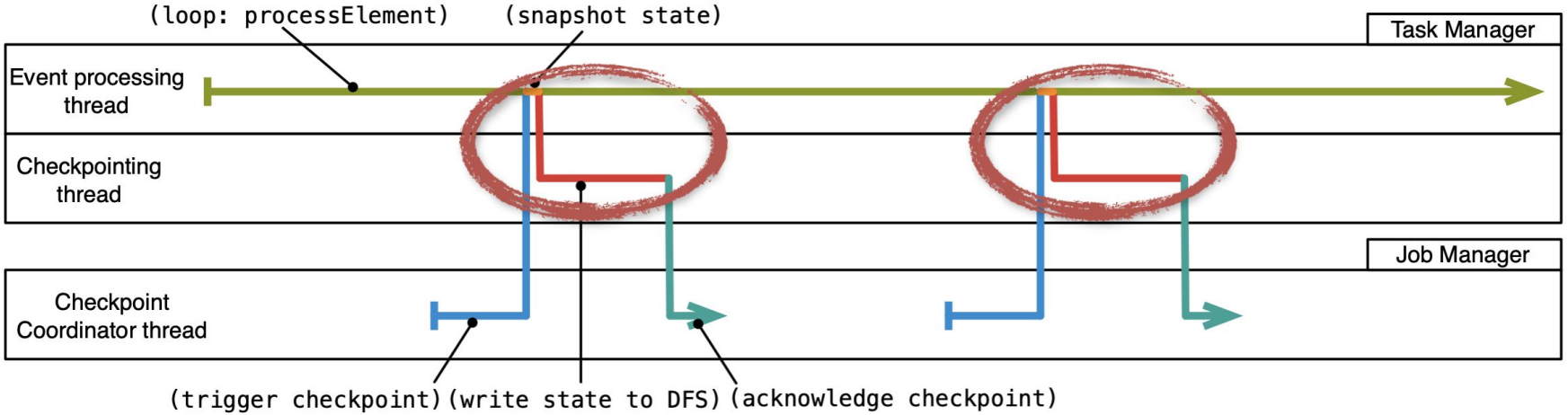


Synchronous Checkpointing



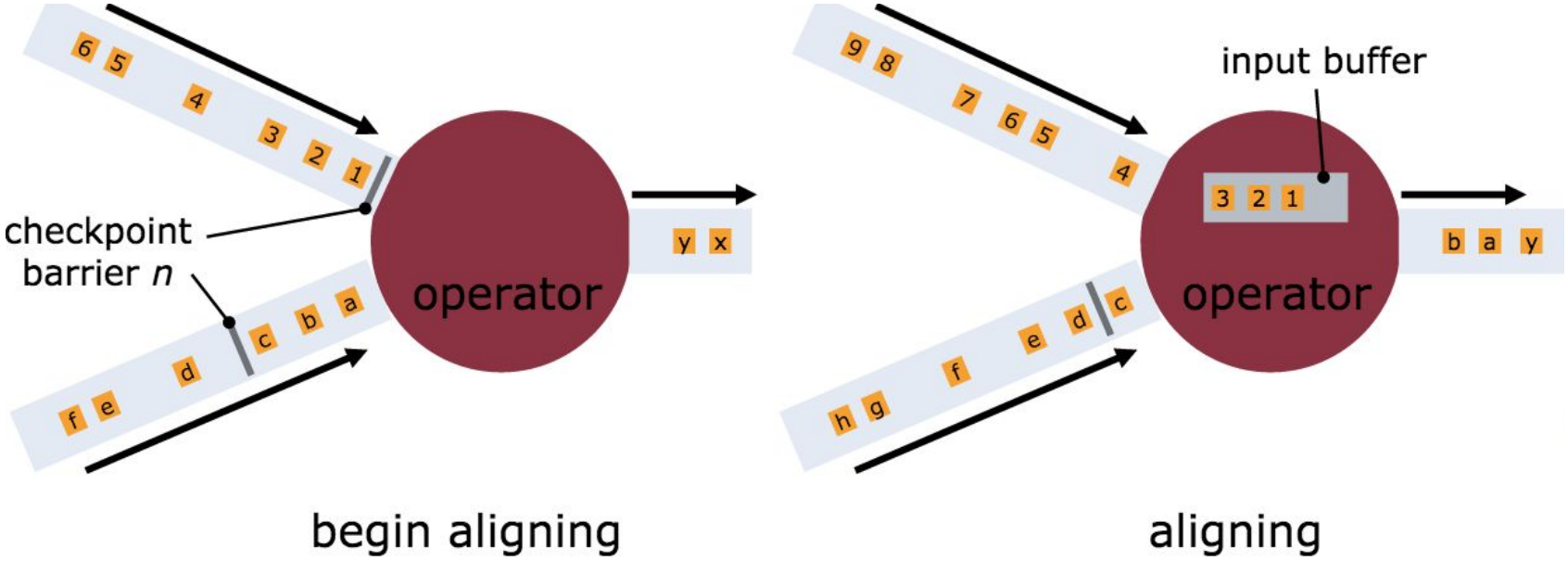
All event processing is on hold to avoid concurrent modifications to state

Asynchronous Checkpointing

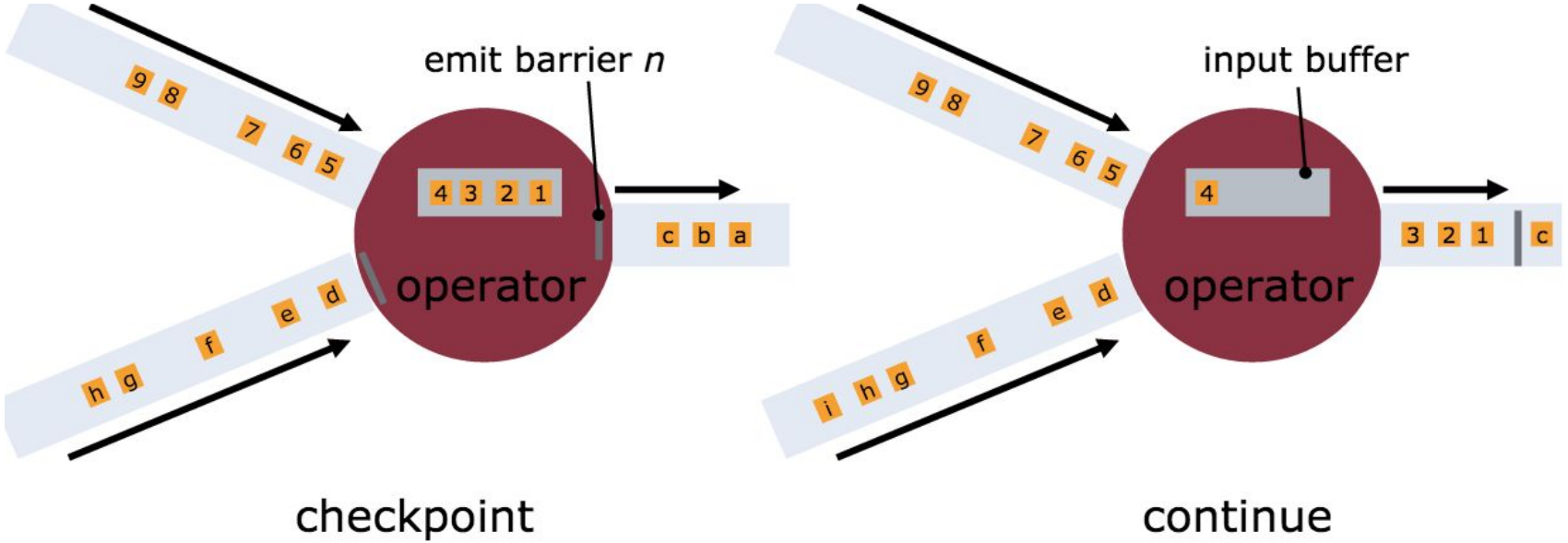


- Minimize pipeline stall time while taking snapshot
- **MVCC (Multi Versioning Concurrency Control)**

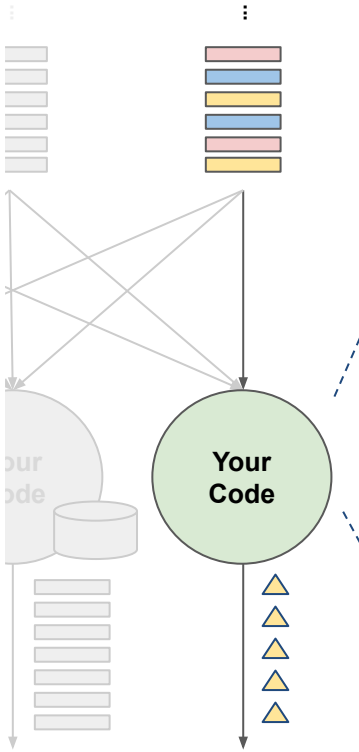
Checkpoint Alignment



Checkpoint Alignment



Handling Large State - Different State Backends

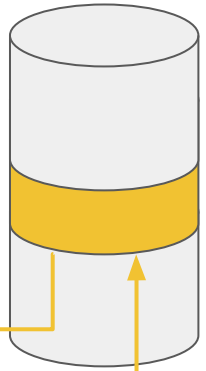



```
val desc =
    new ValueStateDescriptor[MyPojo](
        “my-value-state”,
        createTypeInformation[MyPojo]
    )

val state =
    getRuntimeContext().getState(desc);

val p = state.value();
...

state.update(...)
...
```

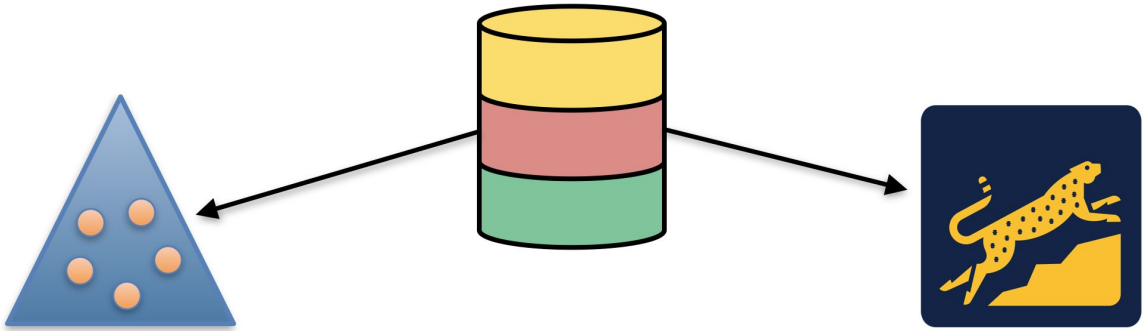
read

write

Local
State Backend



Handling Large State - Different State Backends



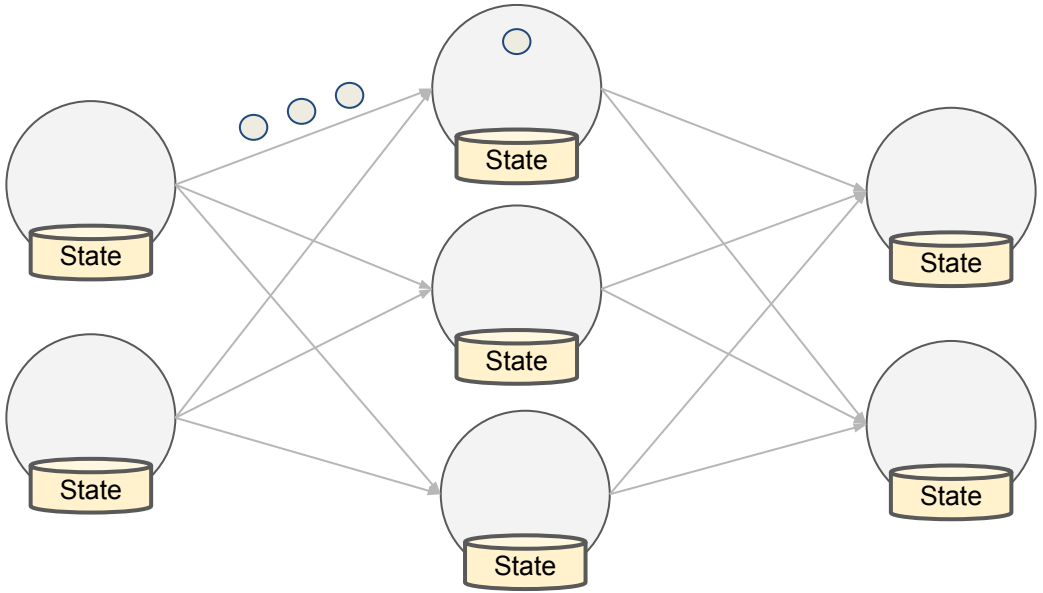
Heap backend

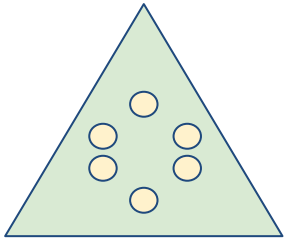
- State lives in memory, on Java heap
- Read / writes operates on Java objects
- State size bounded by memory size

RocksDB backend

- State lives in off-heap and on disk
- Operates on bytes, uses serialization
- State size bounded by disk size

Heap backend

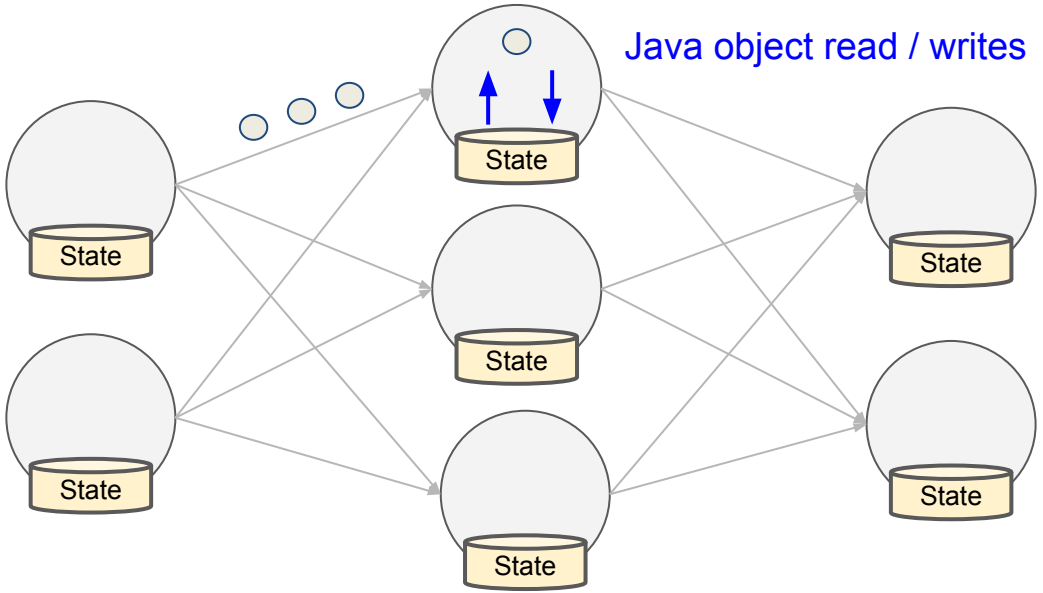


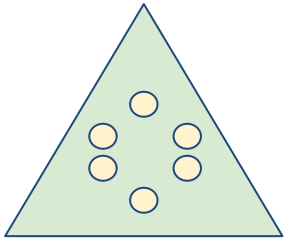


JVM Heap backed state backends (`MemoryStateBackend`, `FsStateBackend`)

⇒ ***lazy*** *serialization +* ***eager*** *deserialization*

Heap backend

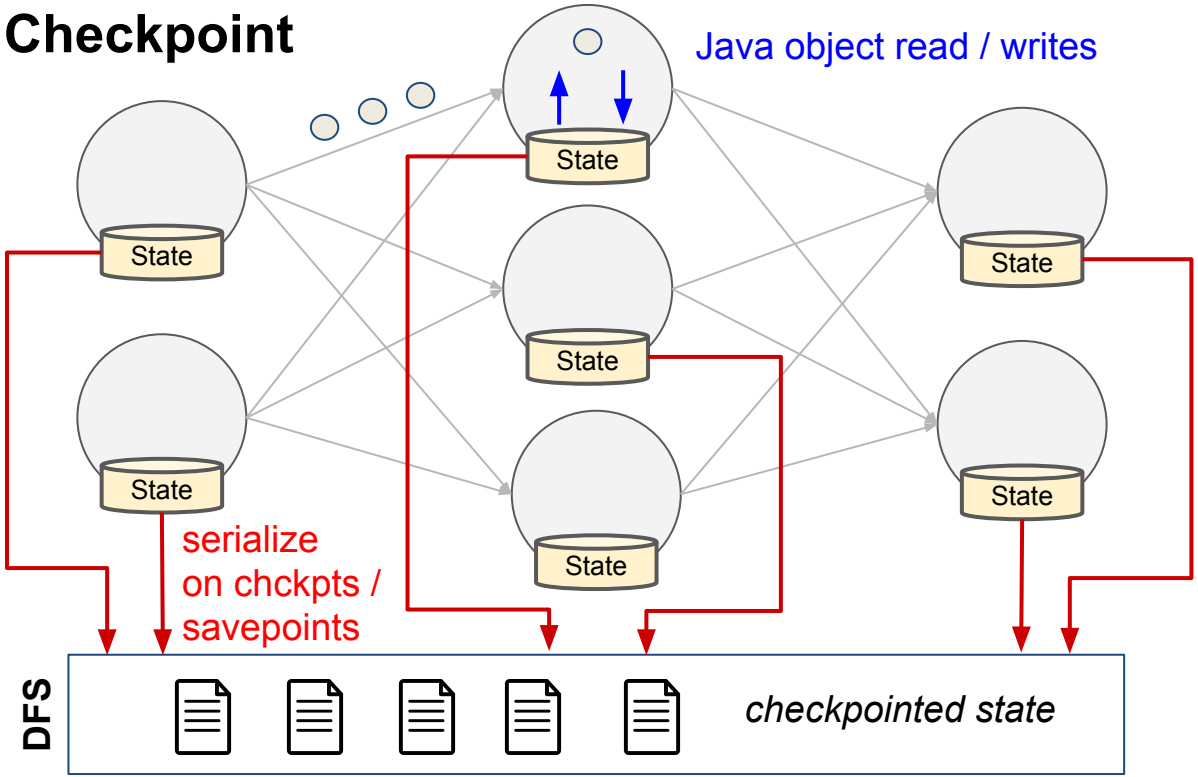


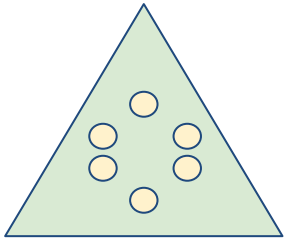


JVM Heap backed state backends (`MemoryStateBackend`, `FsStateBackend`)

⇒ ***lazy*** *serialization +* ***eager*** *deserialization*

Heap backend

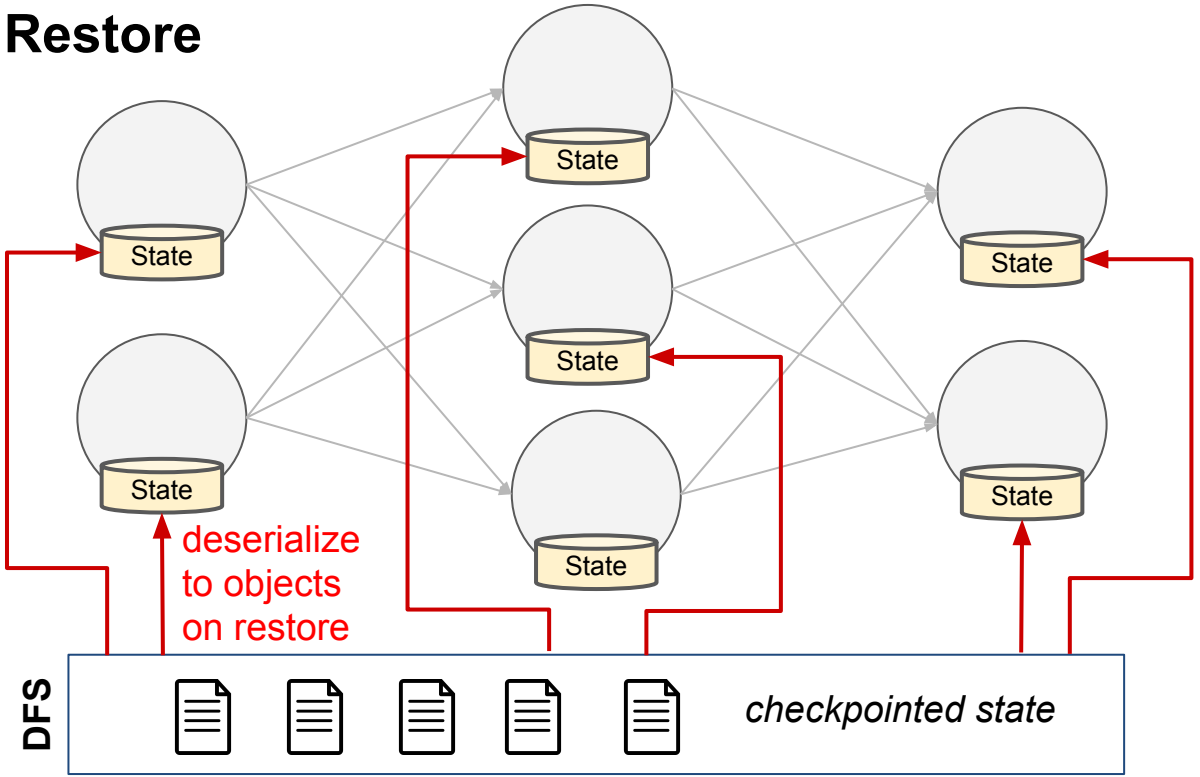


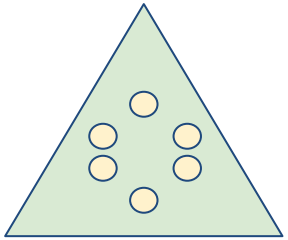


JVM Heap backed state backends (`MemoryStateBackend`, `FsStateBackend`)

⇒ ***lazy*** *serialization +* ***eager*** *deserialization*

Heap backend

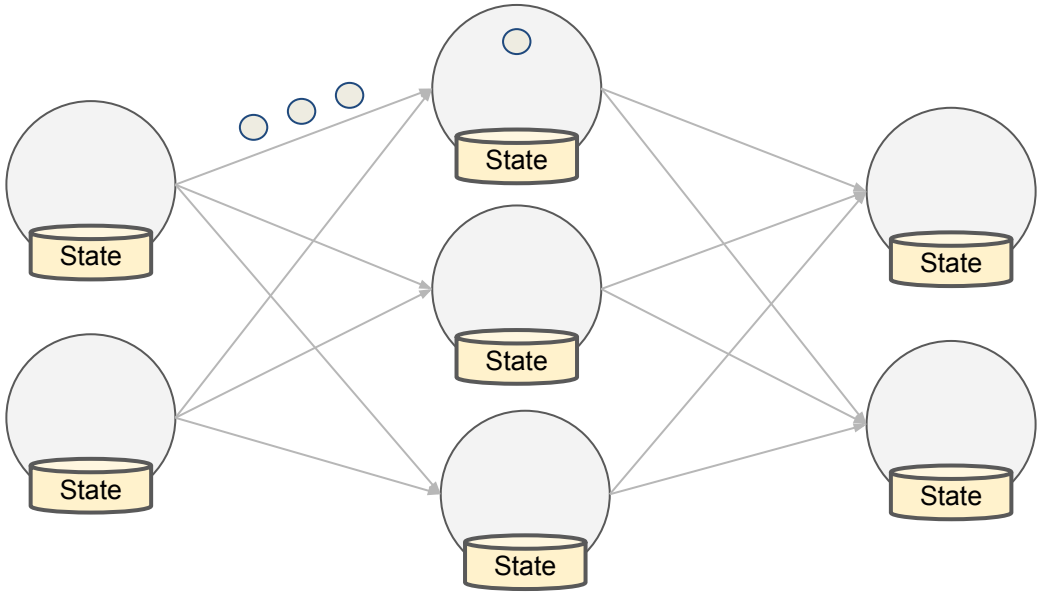




JVM Heap backed state backends (`MemoryStateBackend`, `FsStateBackend`)

⇒ ***lazy*** *serialization +* ***eager*** *deserialization*

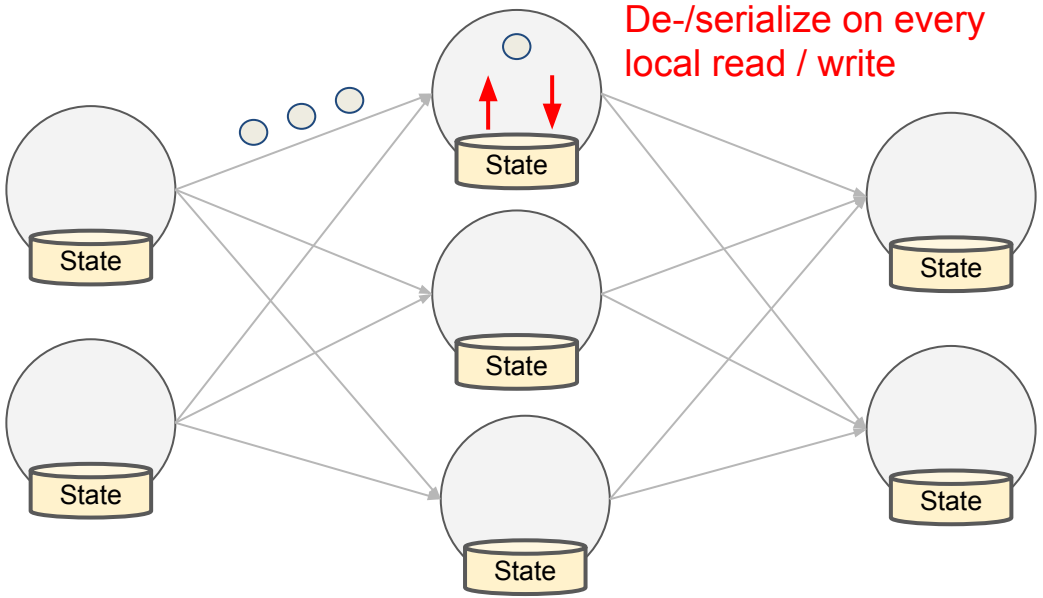
RocksDB backend



Out-of-core state backends
(`RocksDBStateBackend`)

⇒ ***eager*** *serialization +*
lazy *deserialization*

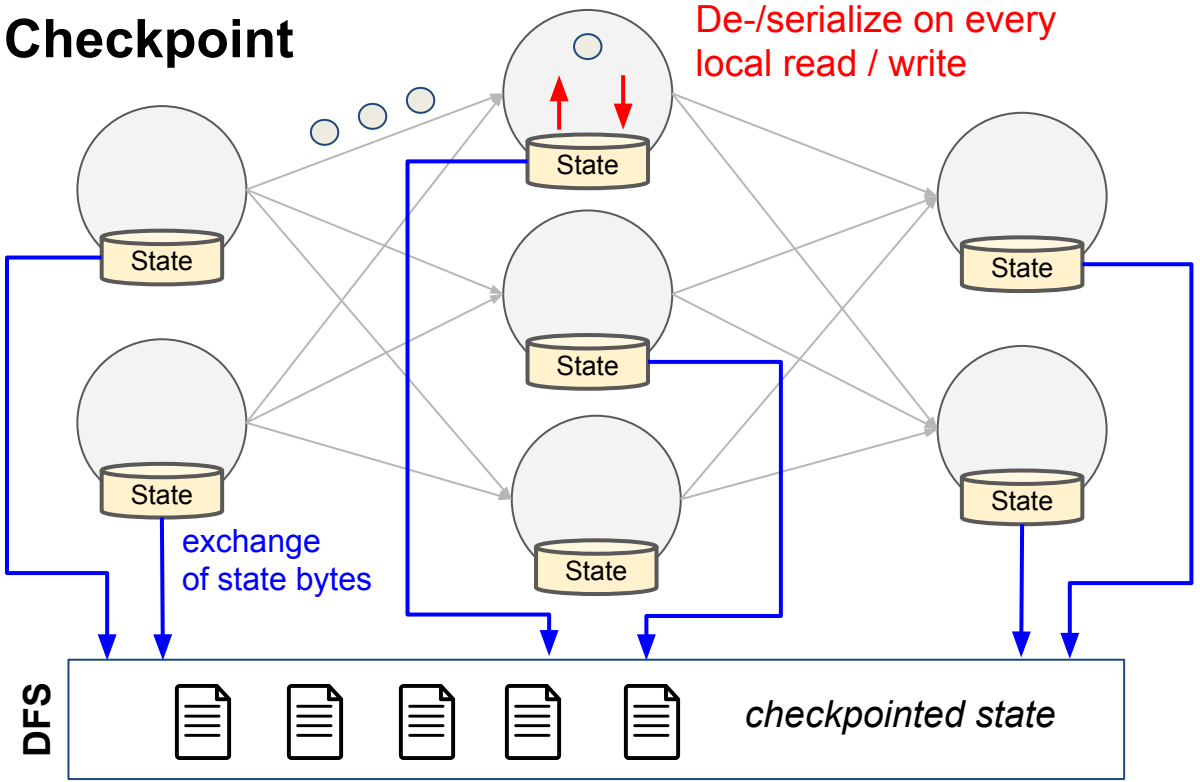
RocksDB backend



Out-of-core state backends
(`RocksDBStateBackend`)

⇒ ***eager*** *serialization +*
lazy *deserialization*

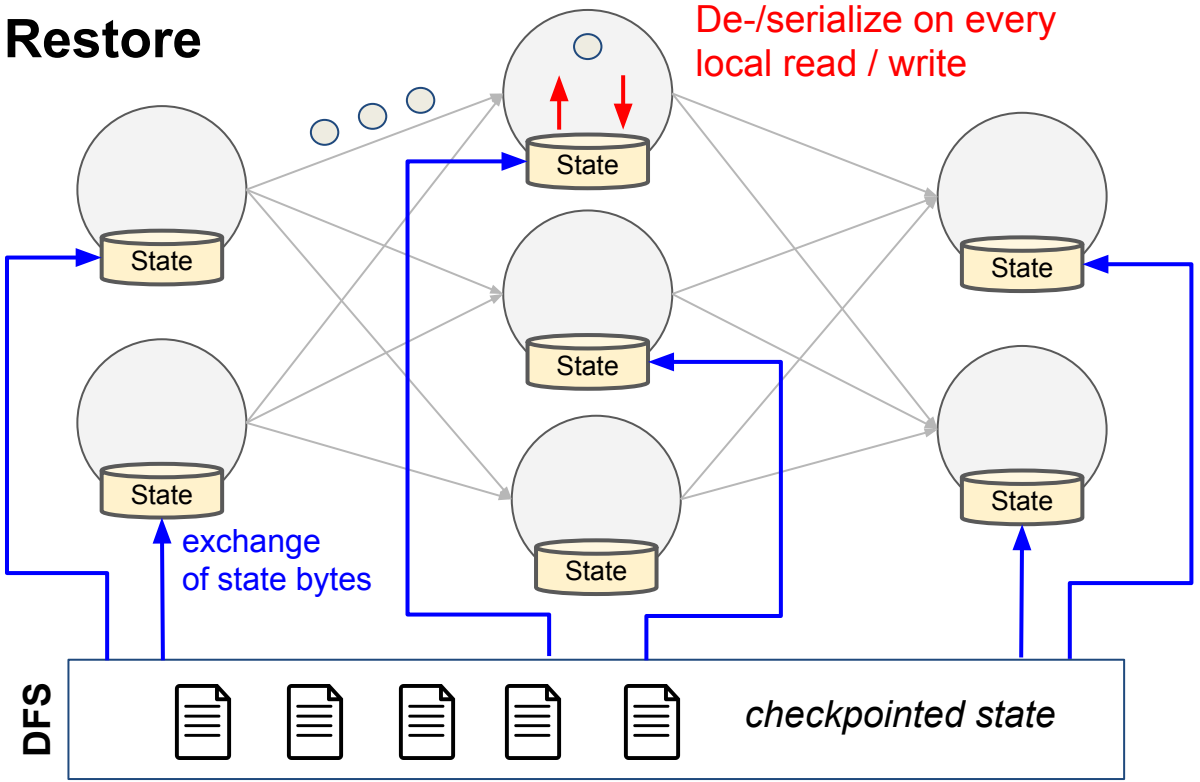
RocksDB backend



Out-of-core state backends
(`RocksDBStateBackend`)

⇒ ***eager*** *serialization +*
lazy *deserialization*

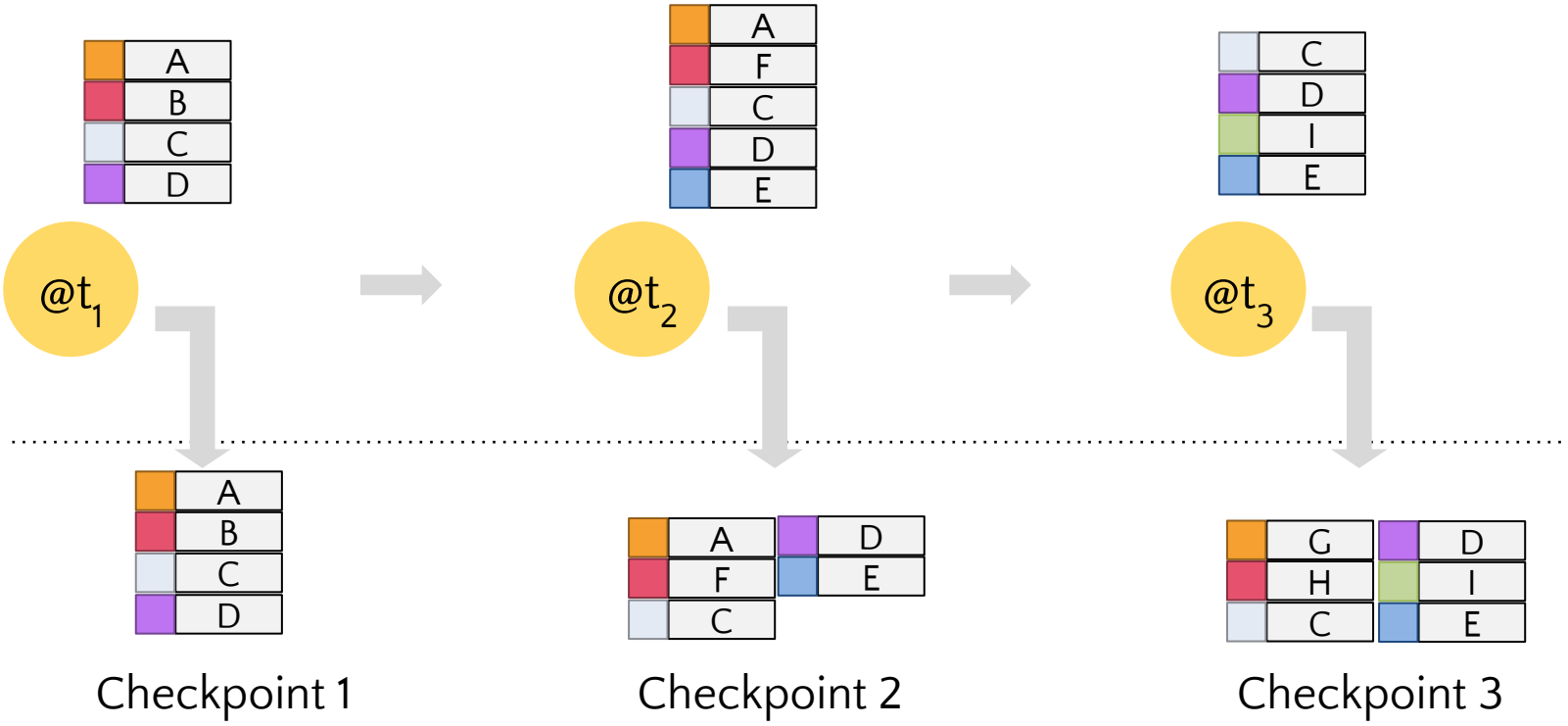
RocksDB backend



Out-of-core state backends
(`RocksDBStateBackend`)

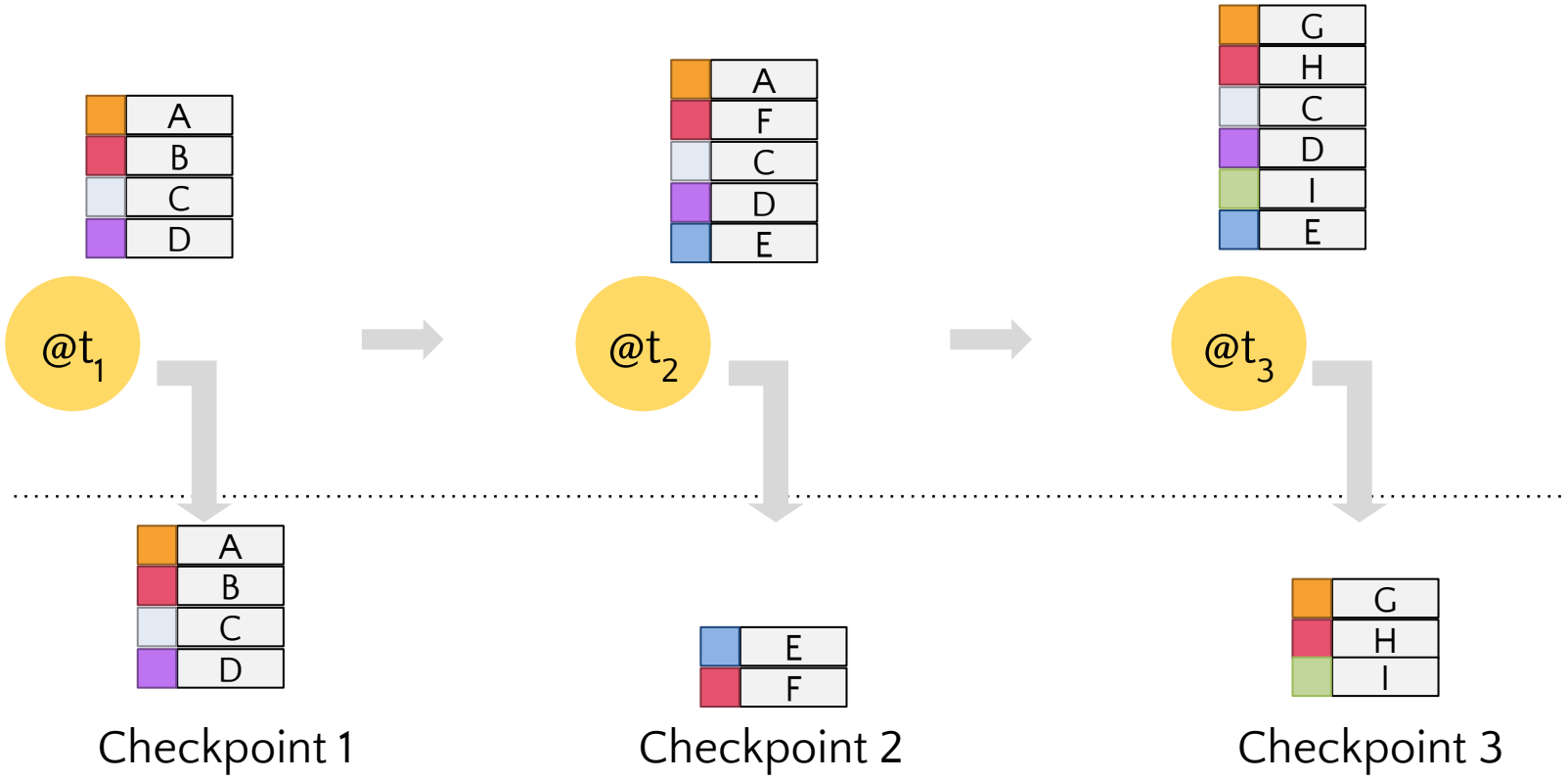
⇒ ***eager*** *serialization +*
lazy *deserialization*

Handling Large State - Incremental Checkpoints



 |

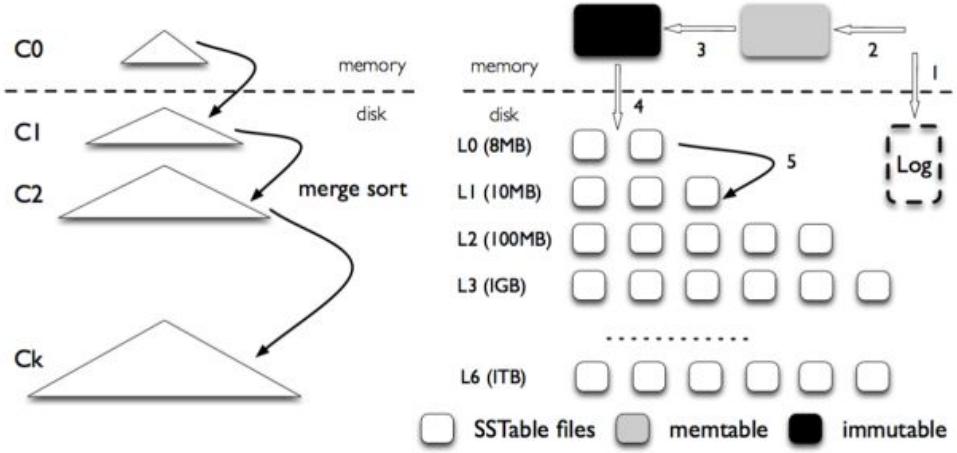
Handling Large State - Incremental Checkpoints



 |

Incremental Checkpoints with RocksDB backend

- RocksDB is inherently well-suited for implementing incremental checkpoints

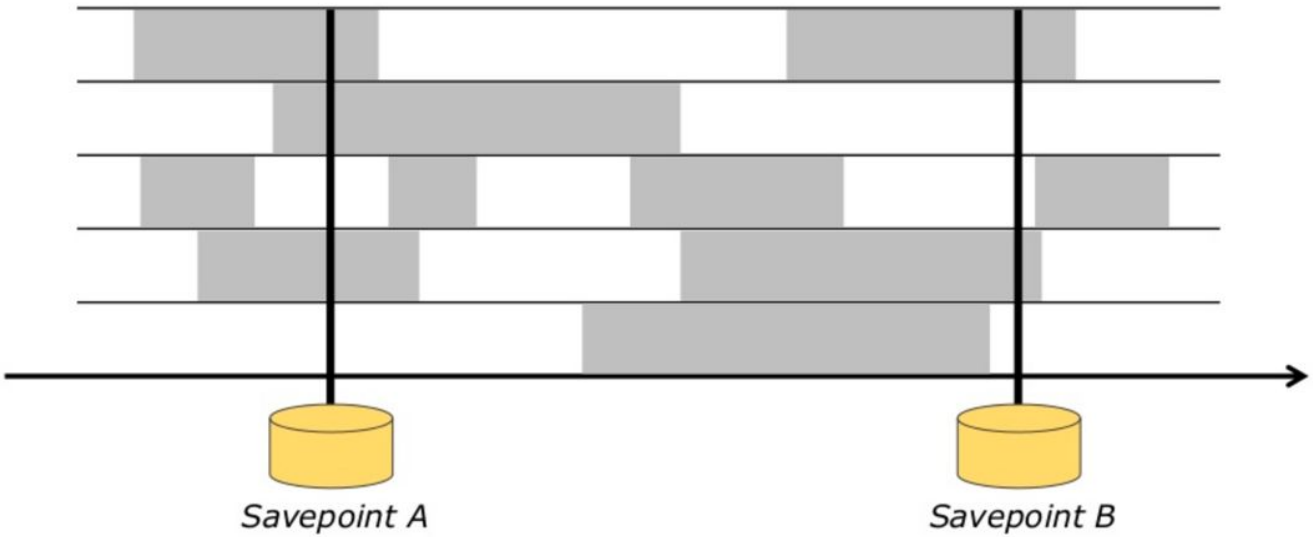


- Reads consider Memtable first, then SSTables
- SSTables are immutable
- **Creation / deletion of SSTables capture the Δ of state**

State Management with Savepoints

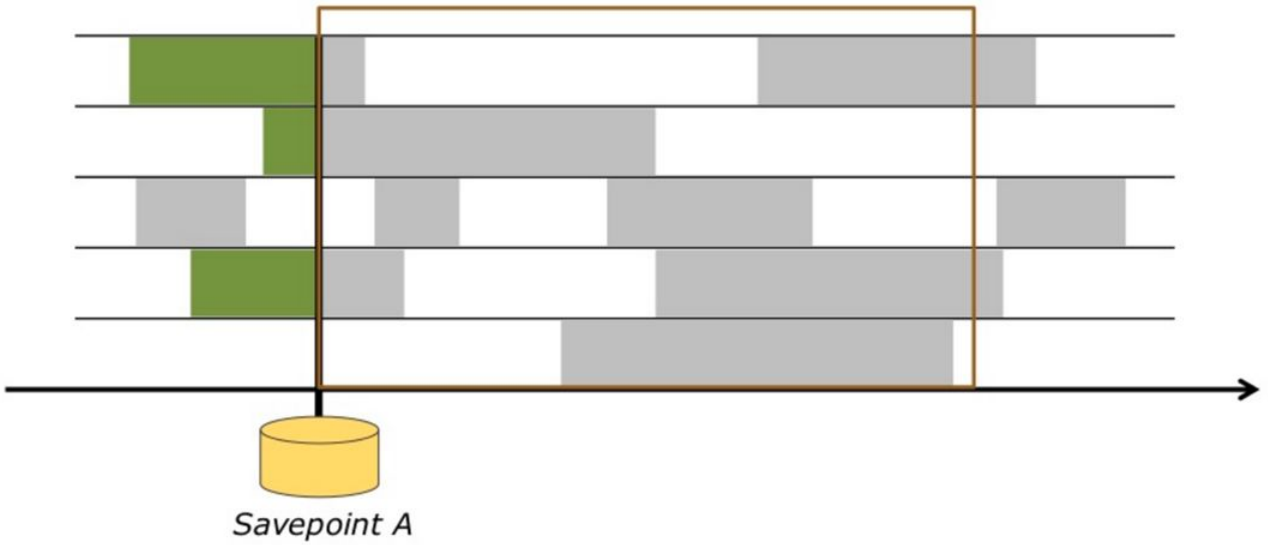
- A persistent snapshot of all state
- When starting an application, state can be initialized from a savepoint
- In-between savepoint and restore we can update Flink version or user code

State Management with Savepoints



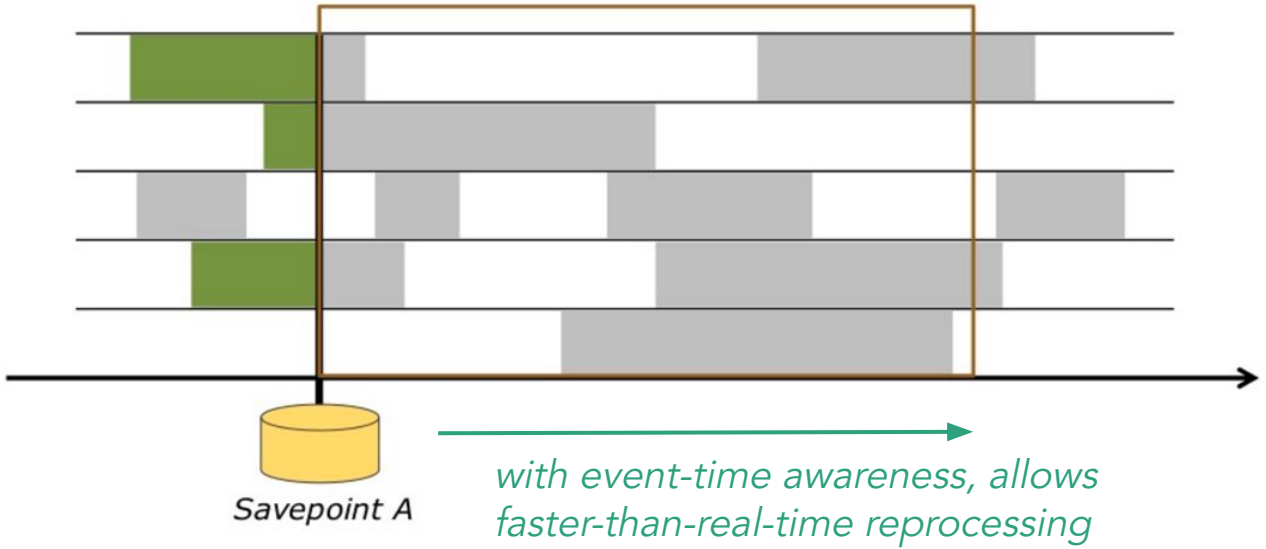
- No stateless point-in-time

State Management with Savepoints



- Before downtime, take a savepoint of the job

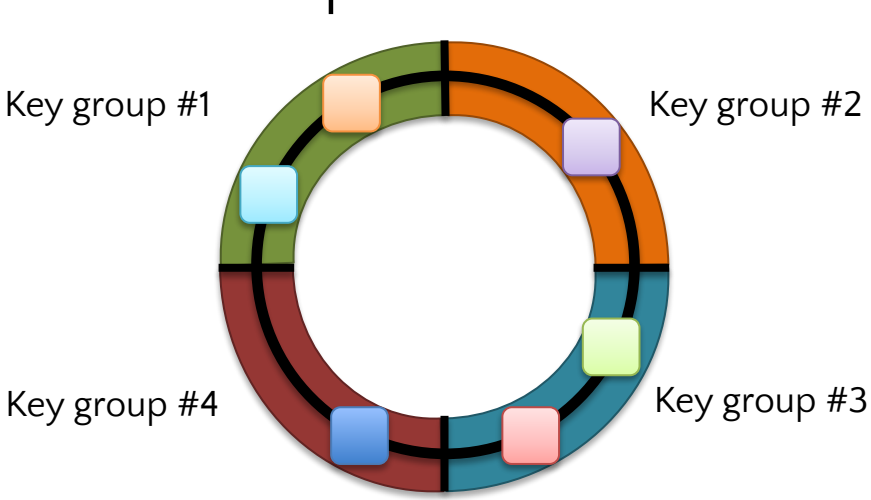
State Management with Savepoints



- Recover job with state, catch up with event-time processing

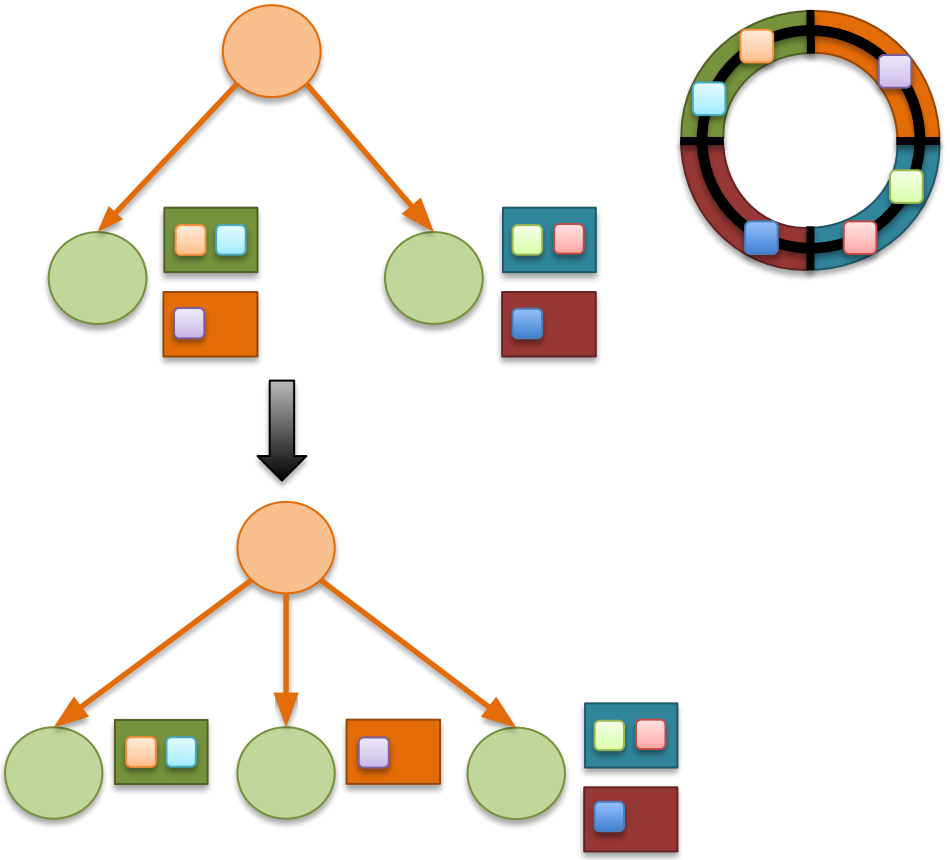
Rescaling State / Elasticity

- Similar to consistent hashing
- Split key space into key groups
- Assign key groups to tasks

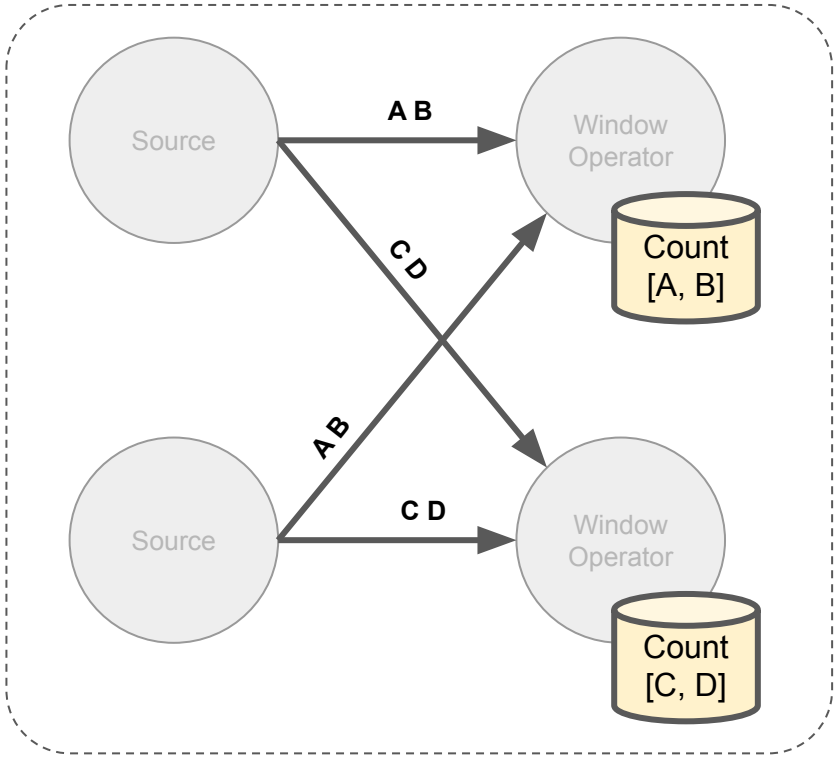


Rescaling State / Elasticity

- Rescaling changes key group assignment
- Maximum parallelism defined by #key groups

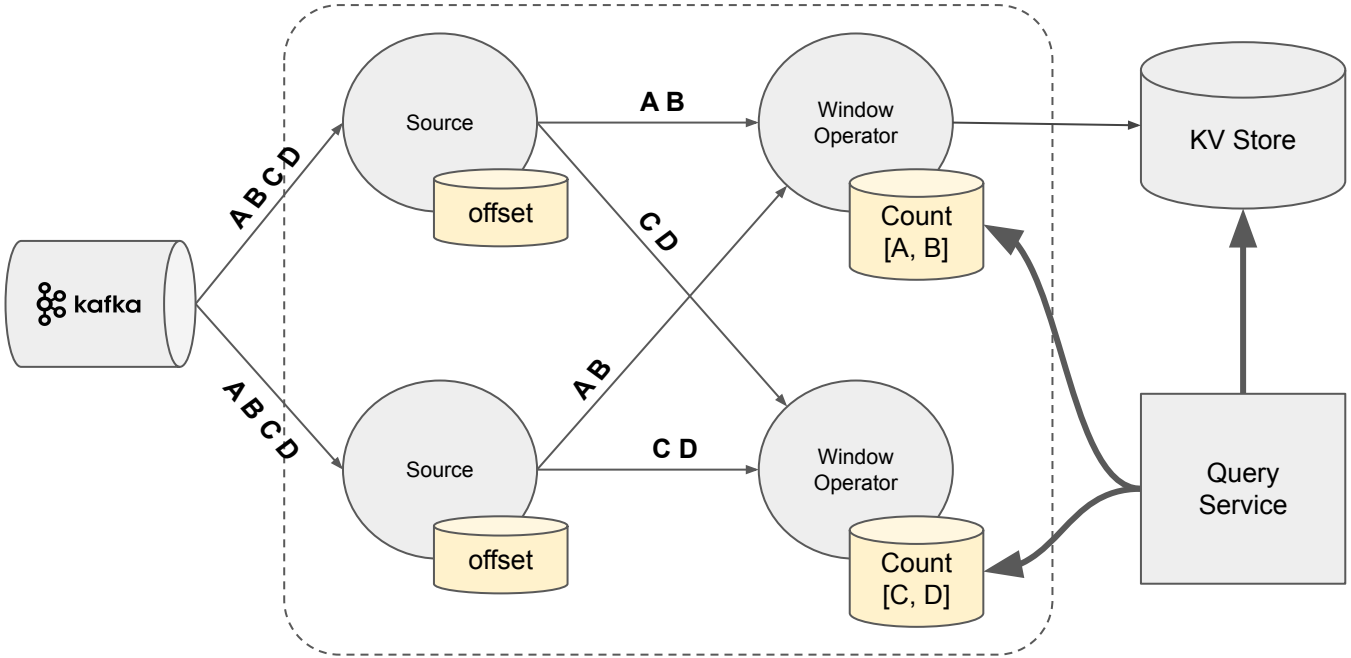


Queryable State



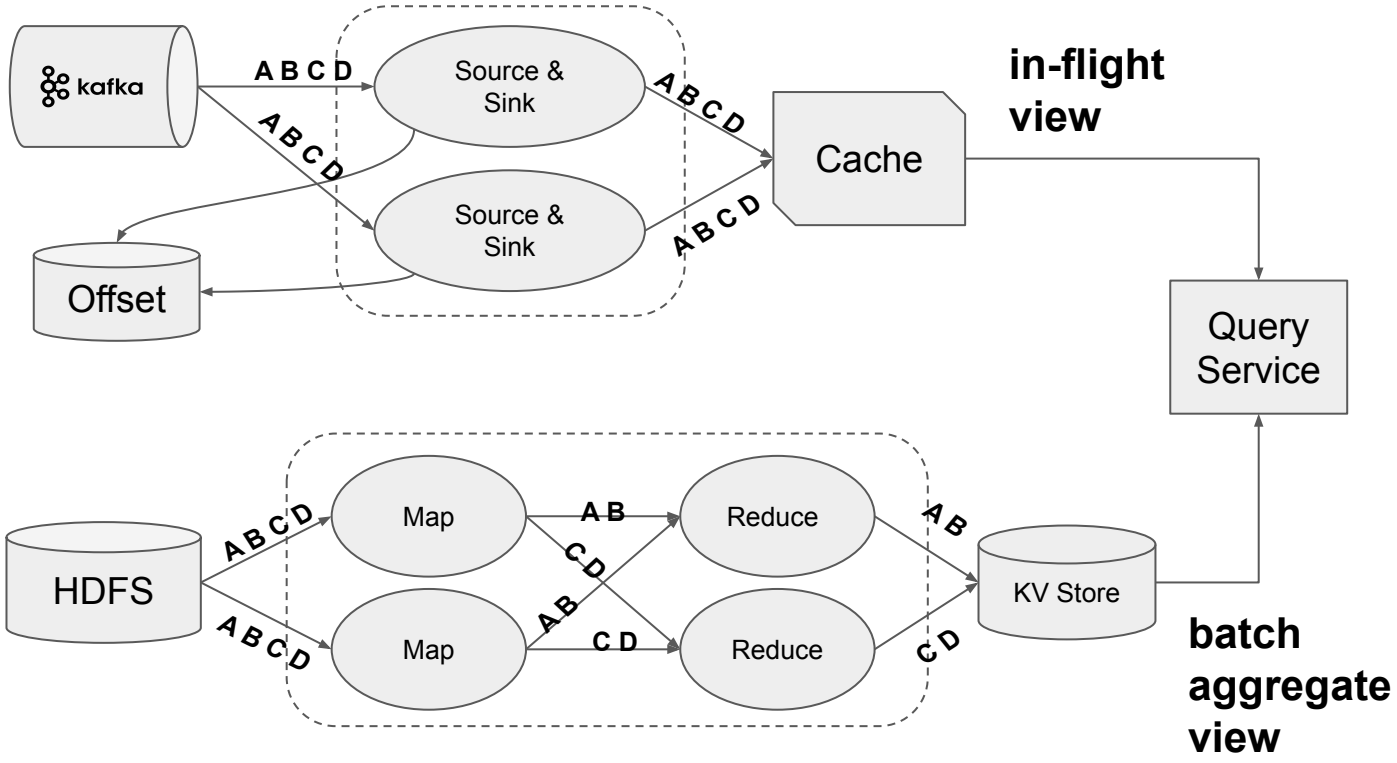
- **Internal state in the stream processor:**
  - Essentially a **key-partitioned**, sharded **KV store**
  - Represents the in-flight aggregations
  - Exactly-once, fault tolerant

A unified approach to replace Lambda Architecture



- Example architecture for supporting a live dashboard with accurate real-time and historic aggregations
- Directly query internal state

Example of a Lambda Architecture



 |

So, in a nutshell …

- **State fault tolerance:** exactly-once semantics, distributed state snapshotting
- **Extremely large state (TB+ scale):** out-of-core state, efficiently snapshotting large state
- **State management:** handling state w.r.t. job changes & rescales
- **Event-time aware:** controlling when results are complete and emitted w.r.t. event-time

Apache Flink, as a stateful stream processor,
is fundamentally designed to address and provide
exactly these ;)

 |

Tzu-Li (Gordon) Tai,
Senior Software Engineer

@tzulitai