DataStream API

State & Failure Recovery

Apache Flink® Training

dataArtisans

Flink v1.3 – 20.06.2017

Working with (Rescalable) State

Stateful Functions

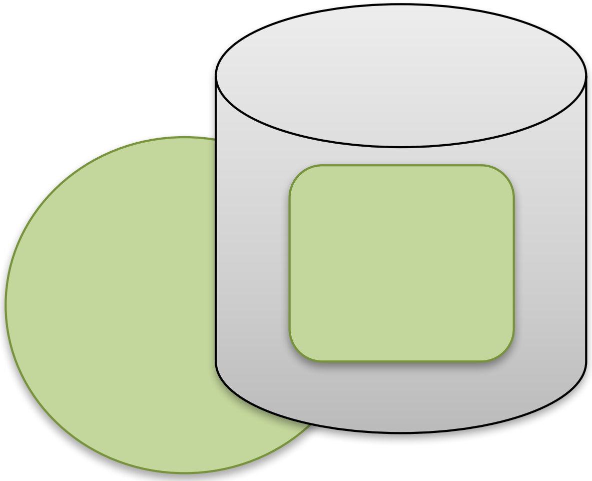
- All DataStream functions can be stateful
  - Flink manages state so that it can be redistributed/rescaled
  - State is checkpointed and restored in case of a failure (if checkpointing is enabled)

- Flink manages two types of state:
  - Operator (non-keyed) state
  - Keyed state

- Flink supports rescaling the state it manages

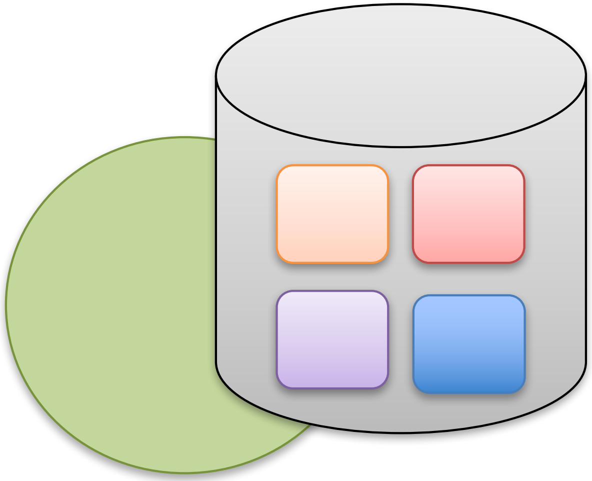
Operator vs Keyed State

Operator (non-keyed)



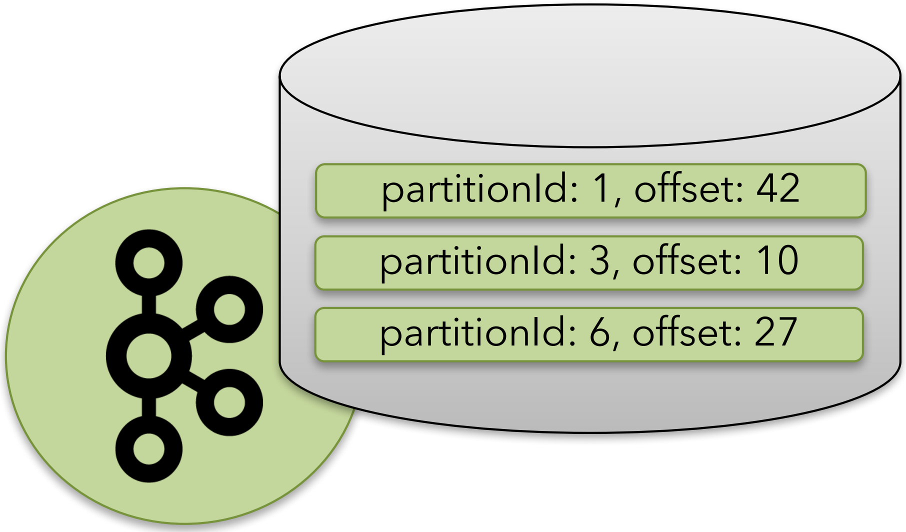
- State bound only to operator
- E.g. source state

Keyed



- State bound to an operator + key
- E.g. Keyed UDF and window state
- `"SELECT count(*) FROM t GROUP BY t.key"`

Repartitioning Operator State



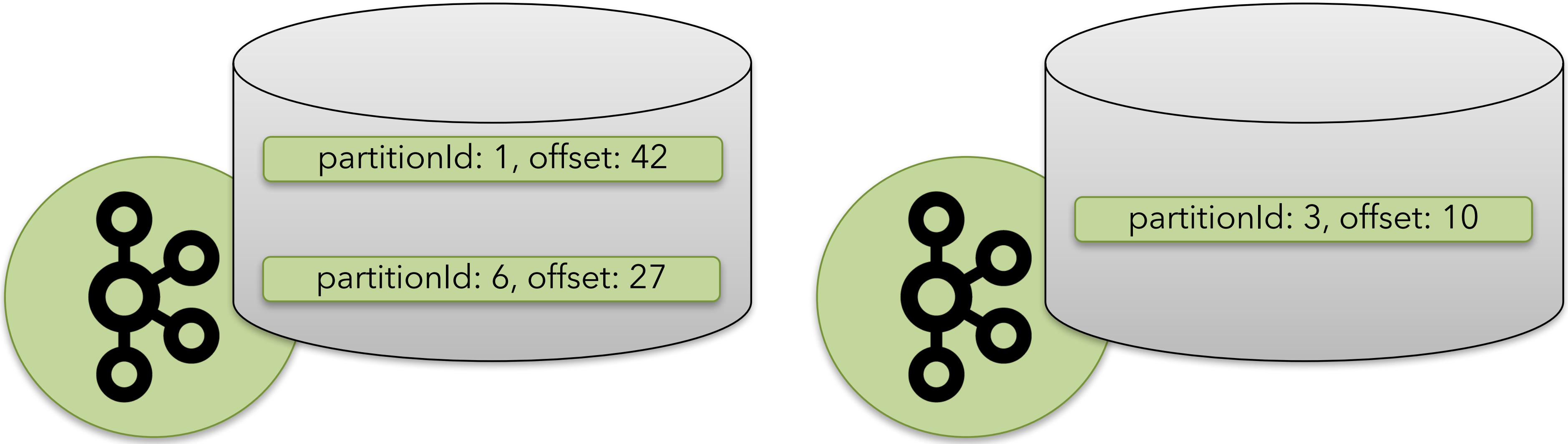
partitionId: 1, offset: 42

partitionId: 3, offset: 10

partitionId: 6, offset: 27

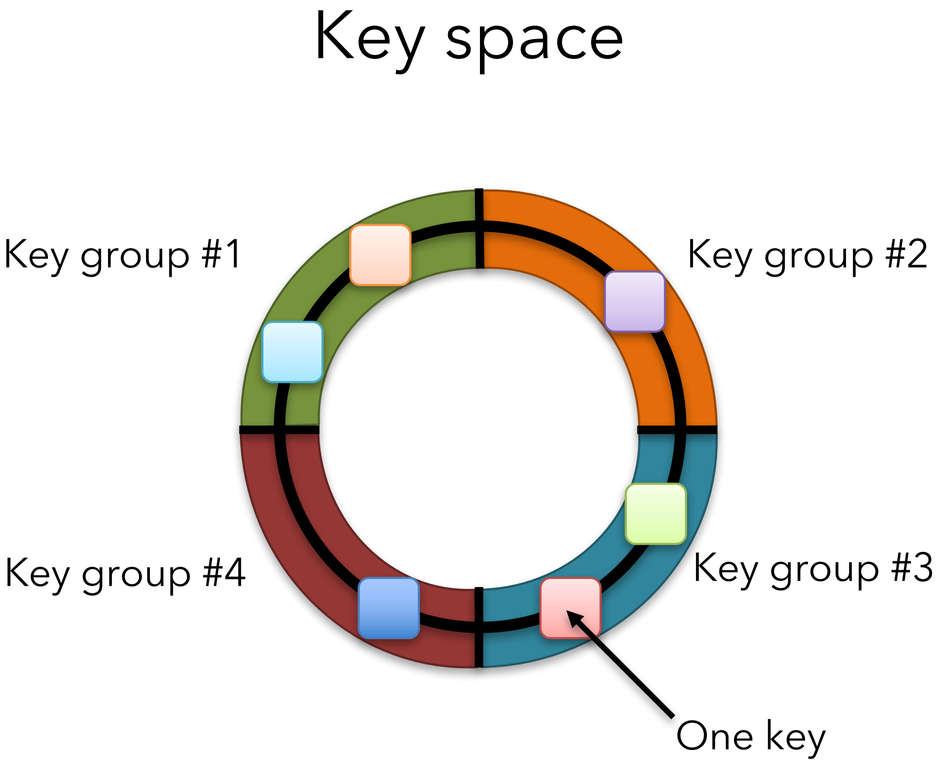
Operator state: a list of state elements which can be freely repartitioned

Scaling out

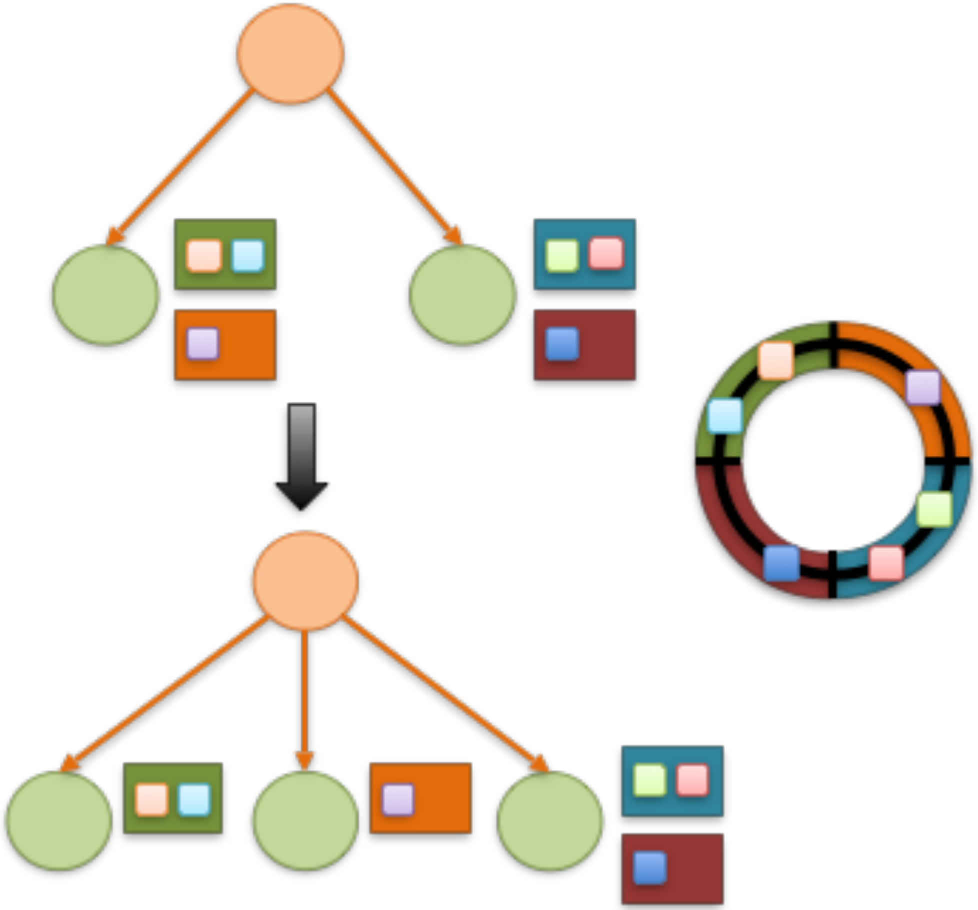

partitionId: 1, offset: 42
partitionId: 6, offset: 27
partitionId: 3, offset: 10

Repartitioning Keyed State

- Split key space into key groups
- Every key falls into exactly one key group
- Assign key groups to tasks
- Maximum parallelism defined by #key groups



Rescaling changes key group assignment



Types of Keyed State

- ValueState<T>
- ListState<T>
- ReducingState<T>
- MapState<UK, UV> *(new in 1.3)*
- ~~FoldingState~~<T> *(deprecated in 1.3)*
    - AggregatingState<IN, OUT>

Using Key-Partitioned State

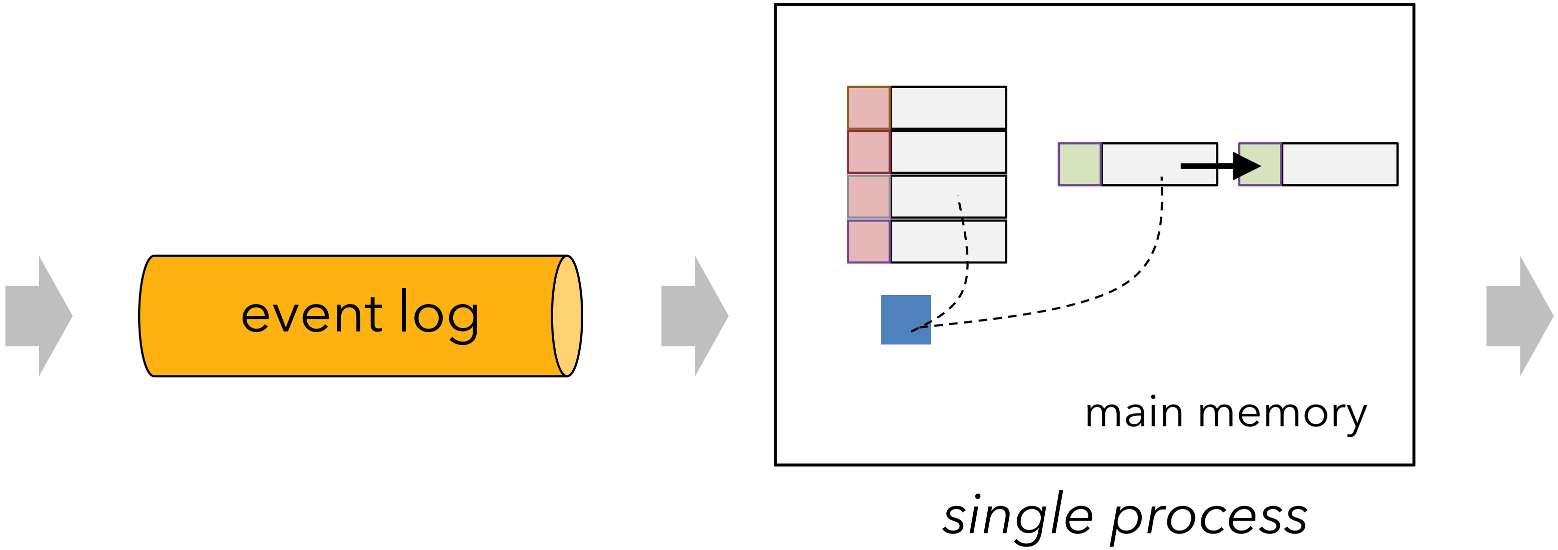
```
DataStream<Tuple2<String, String>> strings = …
DataStream<Long> lengths = strings
  .keyBy(0)
  .map(new MapWithCounter());

public static class MapWithCounter extends RichMapFunction<Tuple2<String, String>, Long> {
    // state object
    private ValueState<Long> totalLengthByKey;

    @Override
    public void open (Configuration conf) {
        // obtain state object
        ValueStateDescriptor<Long> descriptor = new ValueStateDescriptor<>(
            "totalLengthByKey", Long.class);
        totalLengthByKey = getRuntimeContext().getState(descriptor);
    }

    @Override
    public Long map (Tuple2<String, String> value) throws Exception {
        long length = totalLengthByKey.value();    // fetch state for current key
        if (length == null) length = 0;
        long newTotalLength = length + value.f1.length();
        totalLengthByKey.update(newTotalLength);   // update state of current key
        return totalLengthByKey.value();
    }
}
```

Fault Tolerance via Snapshotting

Fault tolerance: simple case

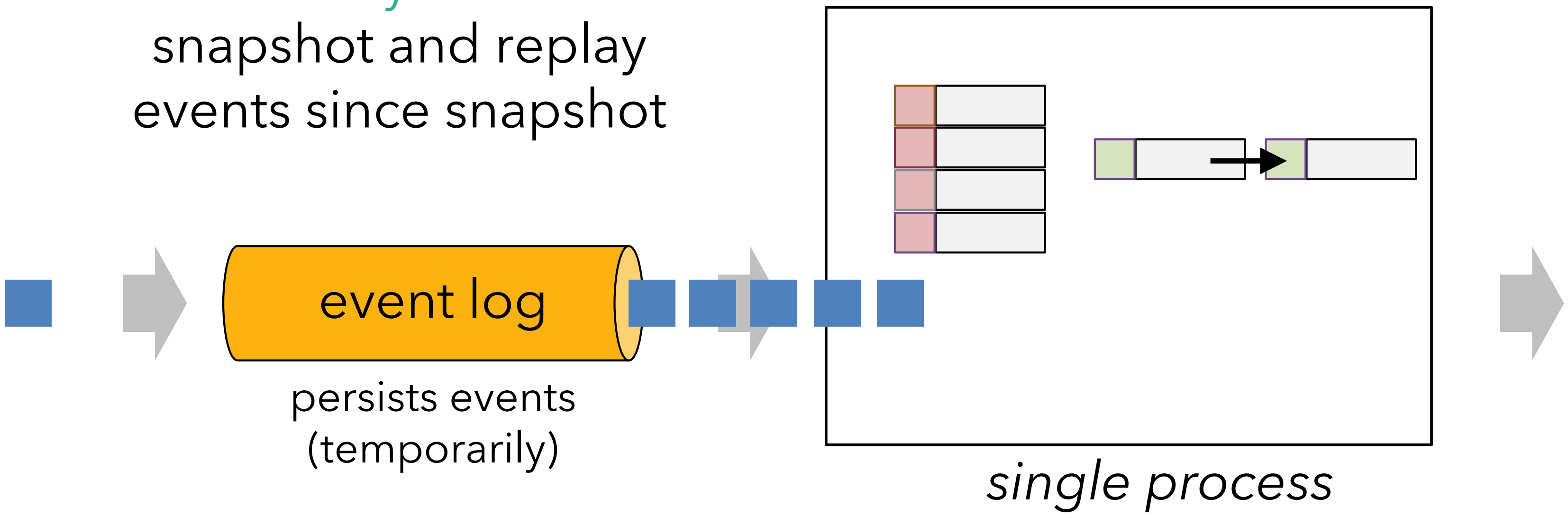
How to ensure exactly-once semantics for the state?

periodically snapshot the memory



Fault tolerance: simple case

Recovery: restore snapshot and replay events since snapshot



event log

persists events (temporarily)

single process

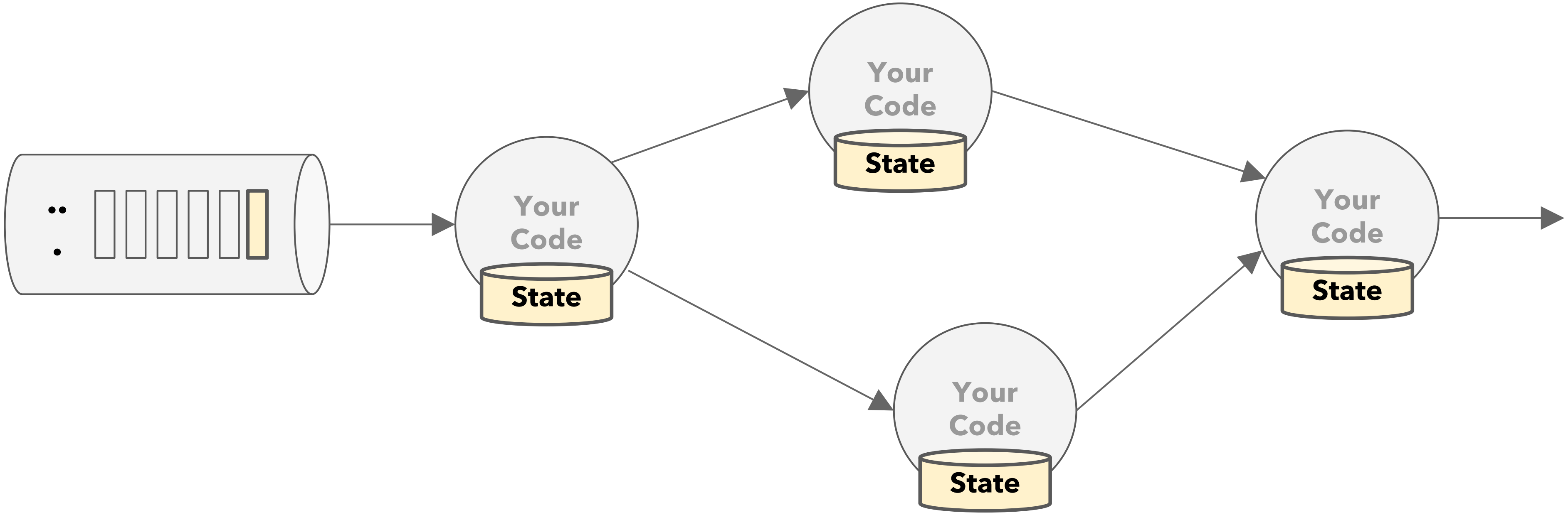
State fault tolerance

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 interrupting computation**?

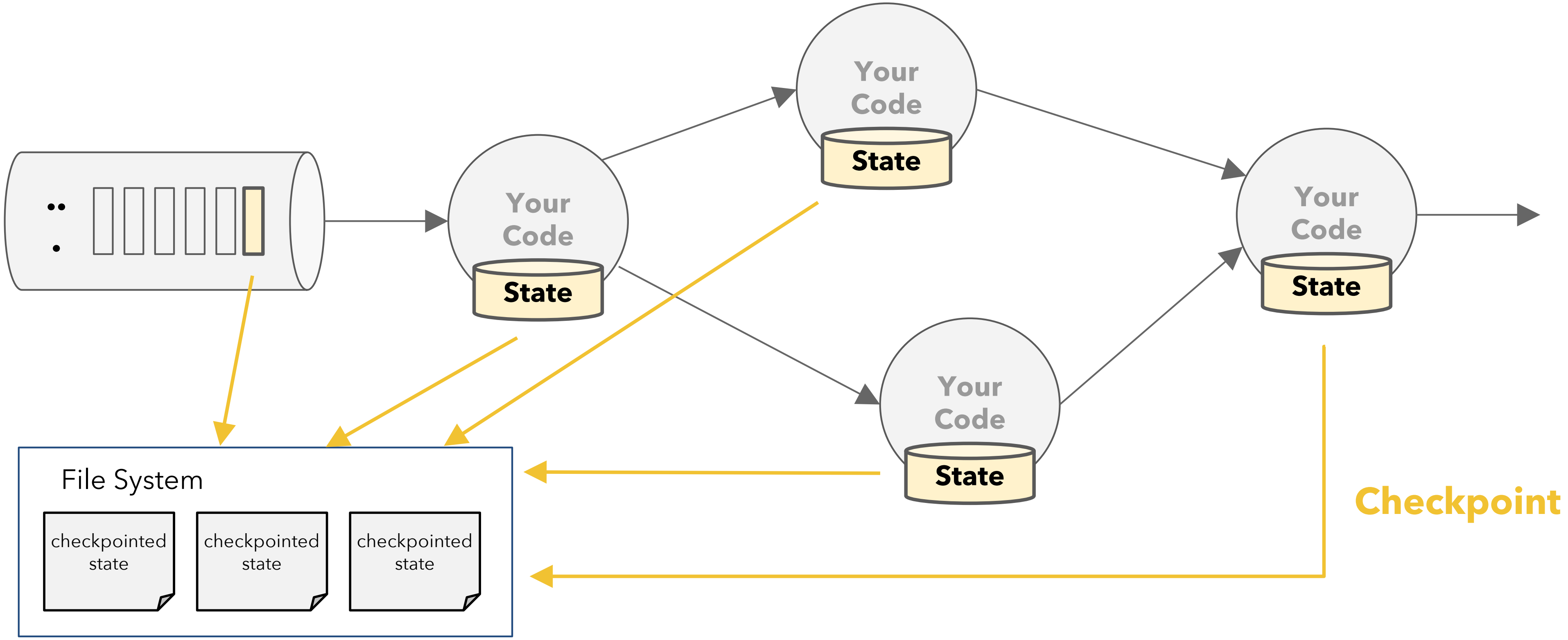
State fault tolerance (II)

Consistent snapshotting:



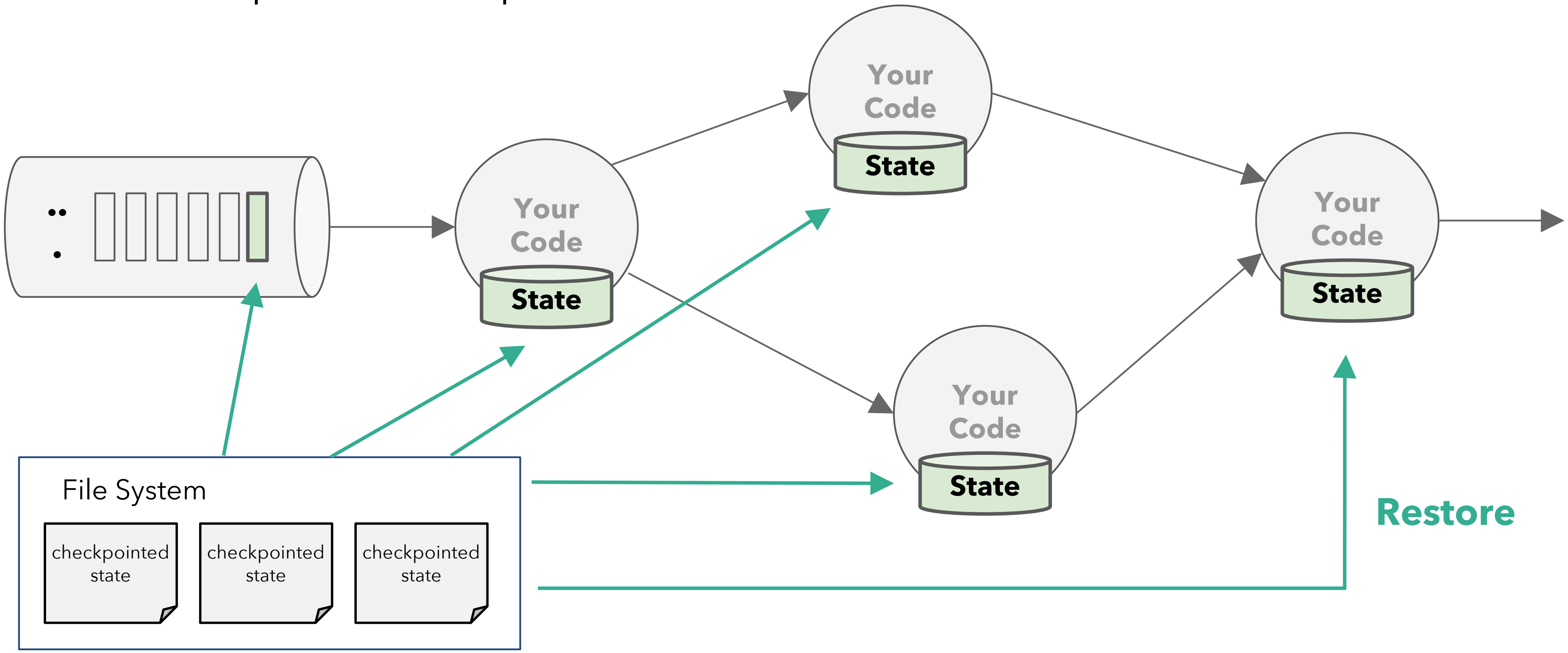
State fault tolerance (II)

Consistent snapshotting:



State fault tolerance (III)

- Recover all embedded state
- Reset position in input stream



Checkpoints

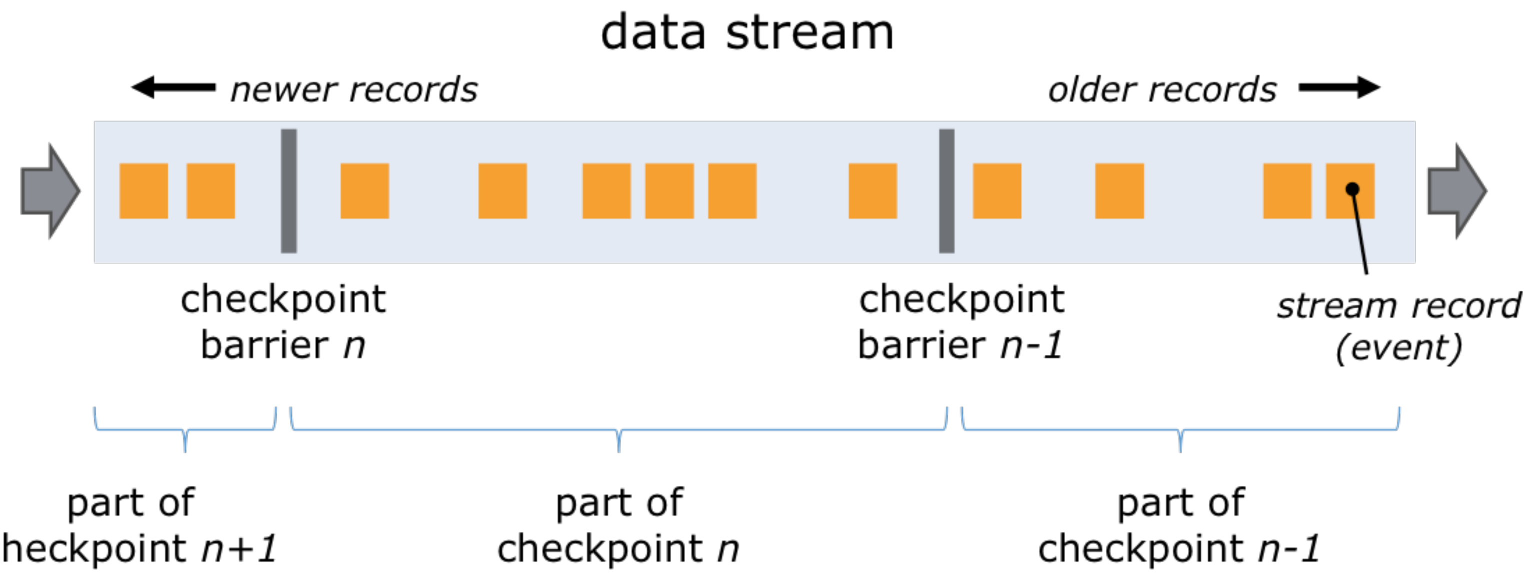
Checkpointing in Flink

- Asynchronous Barrier Snapshotting
  - checkpoint barriers are inserted into the stream and flow through the graph along with the data
  - this avoids a "global pause" during checkpointing

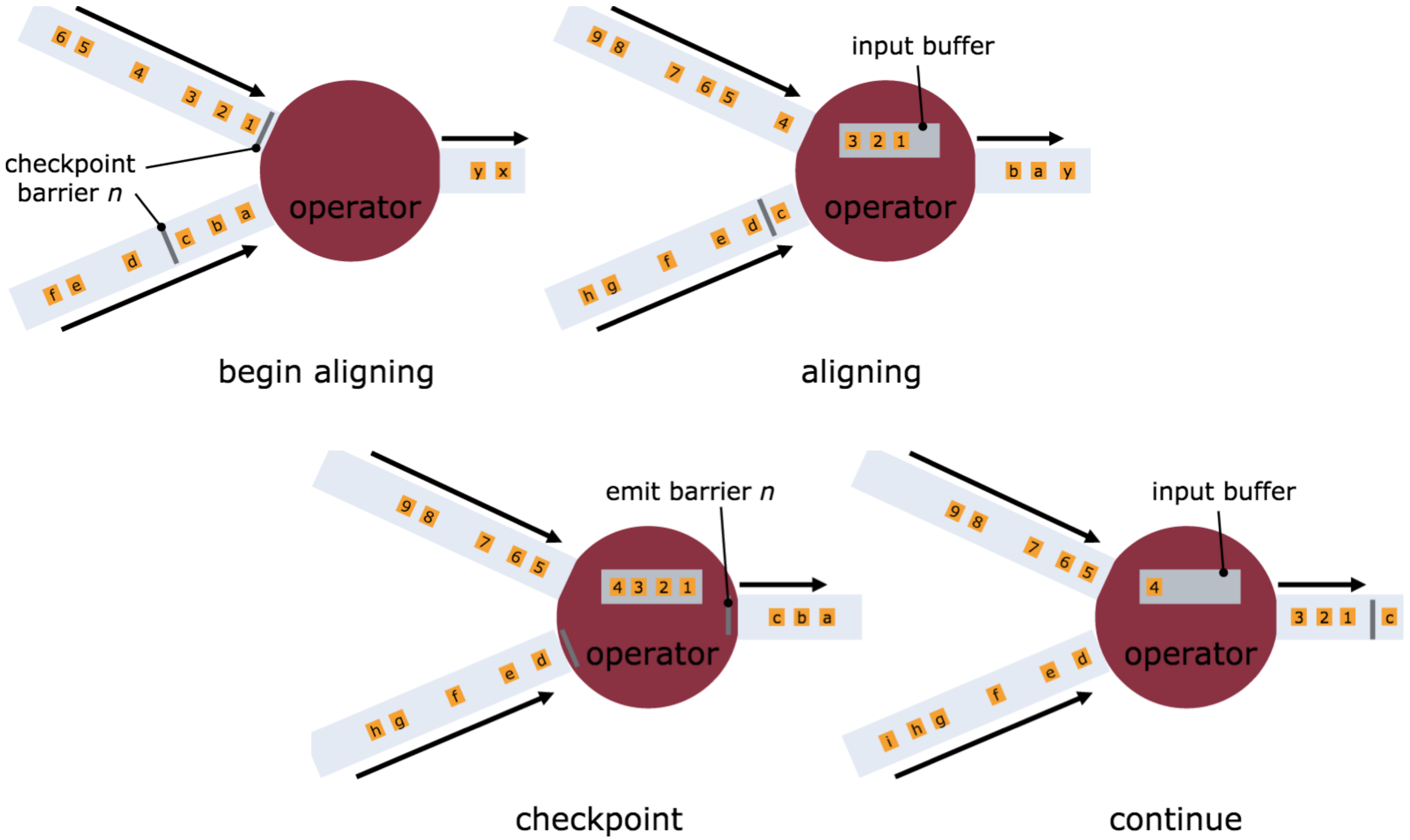
- Checkpoint barriers cause ...
  - replayable sources to checkpoint their offsets
  - operators to checkpoint their state
  - sinks to commit open transactions

- The program is rolled back to the latest completed checkpoint in case of a failure.

Checkpoint Barriers



Asynchronous Barrier Snapshotting



begin aligning

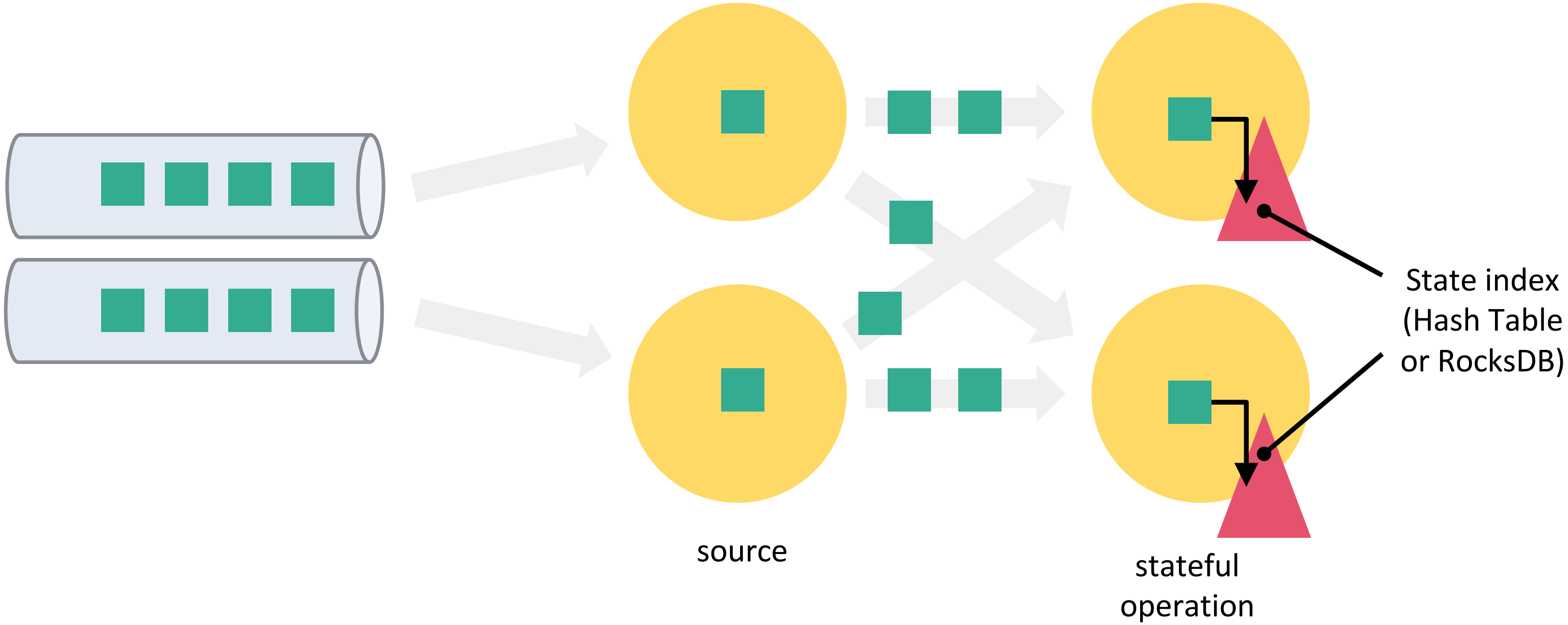
aligning

checkpoint

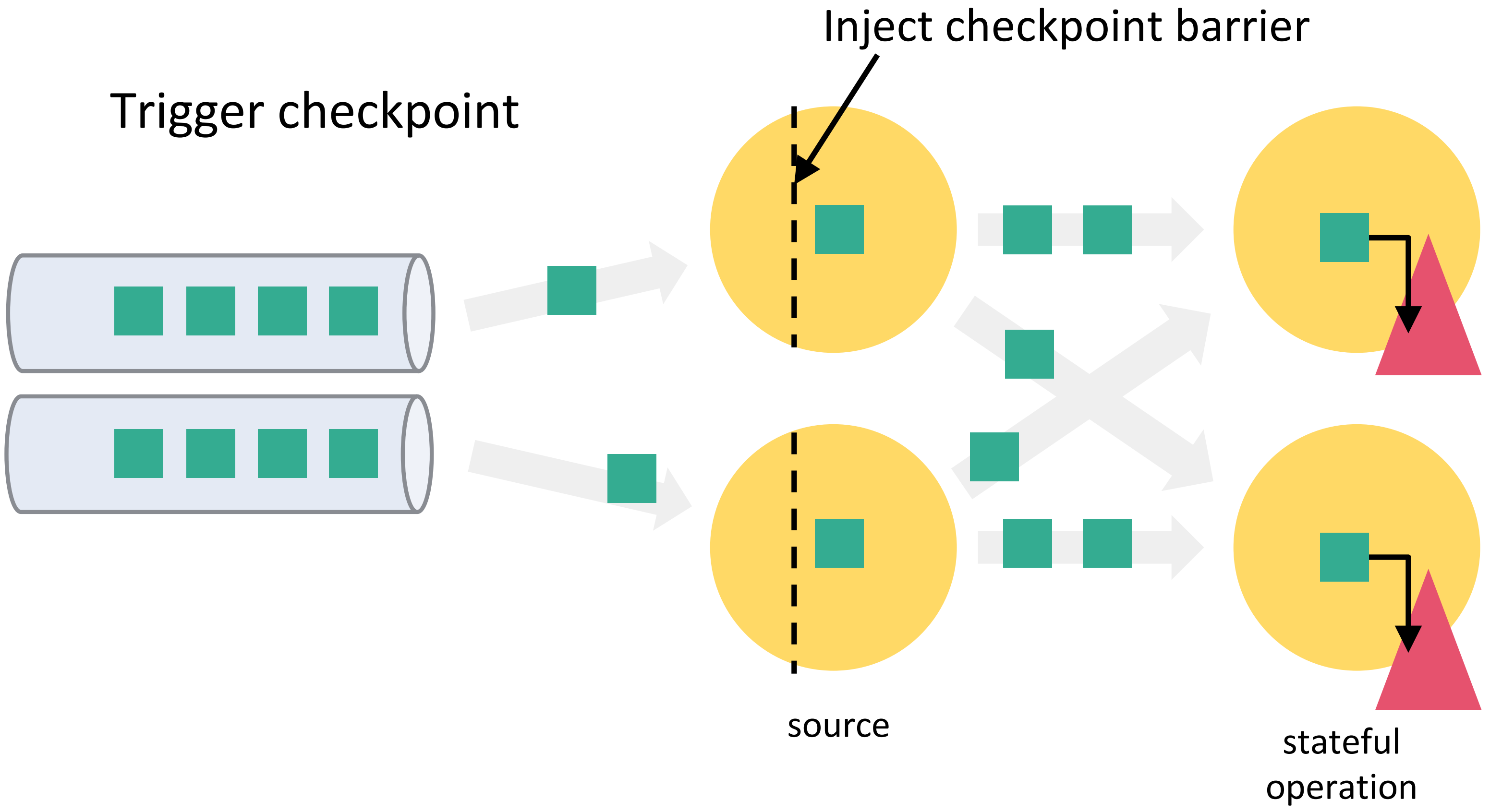
continue

Distributed snapshots

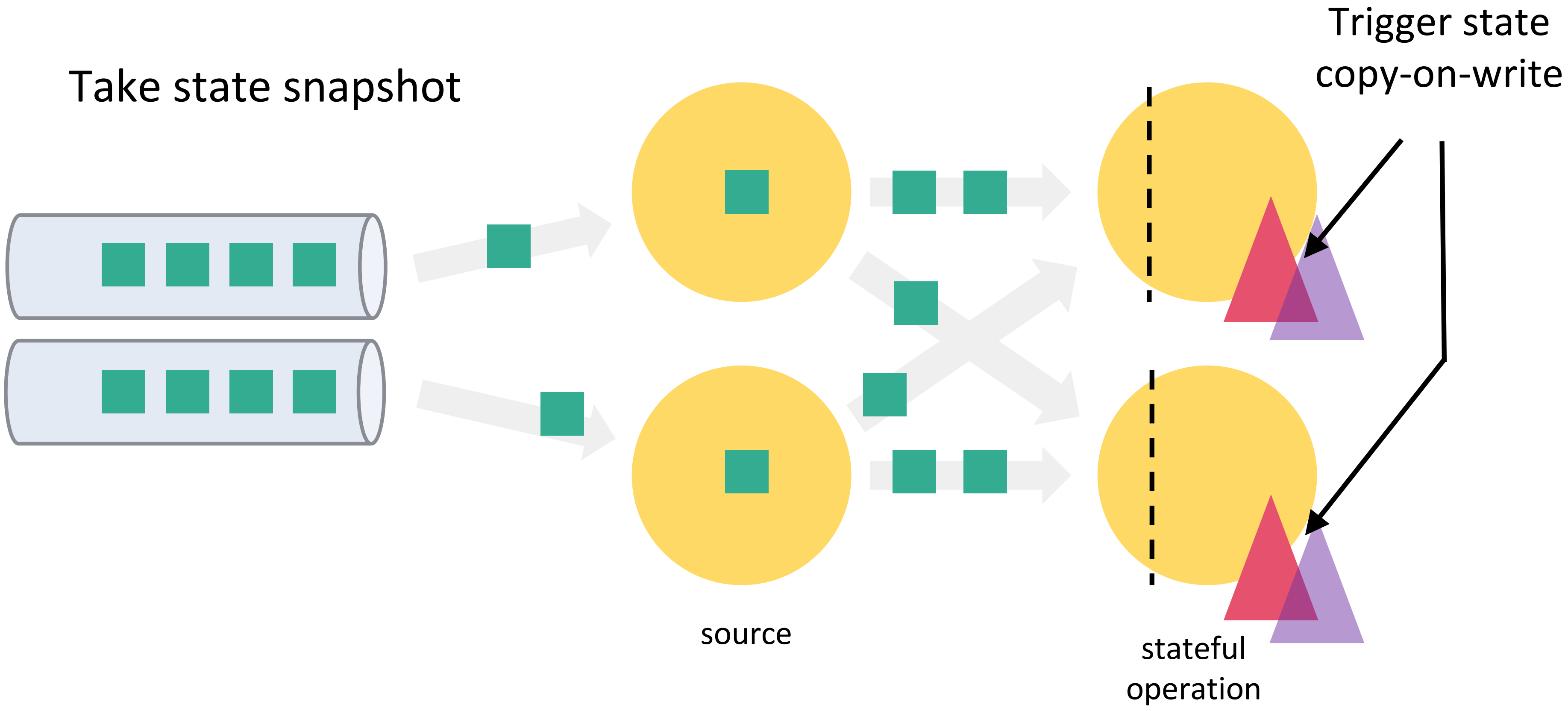
Events flow without replication or synchronous writes



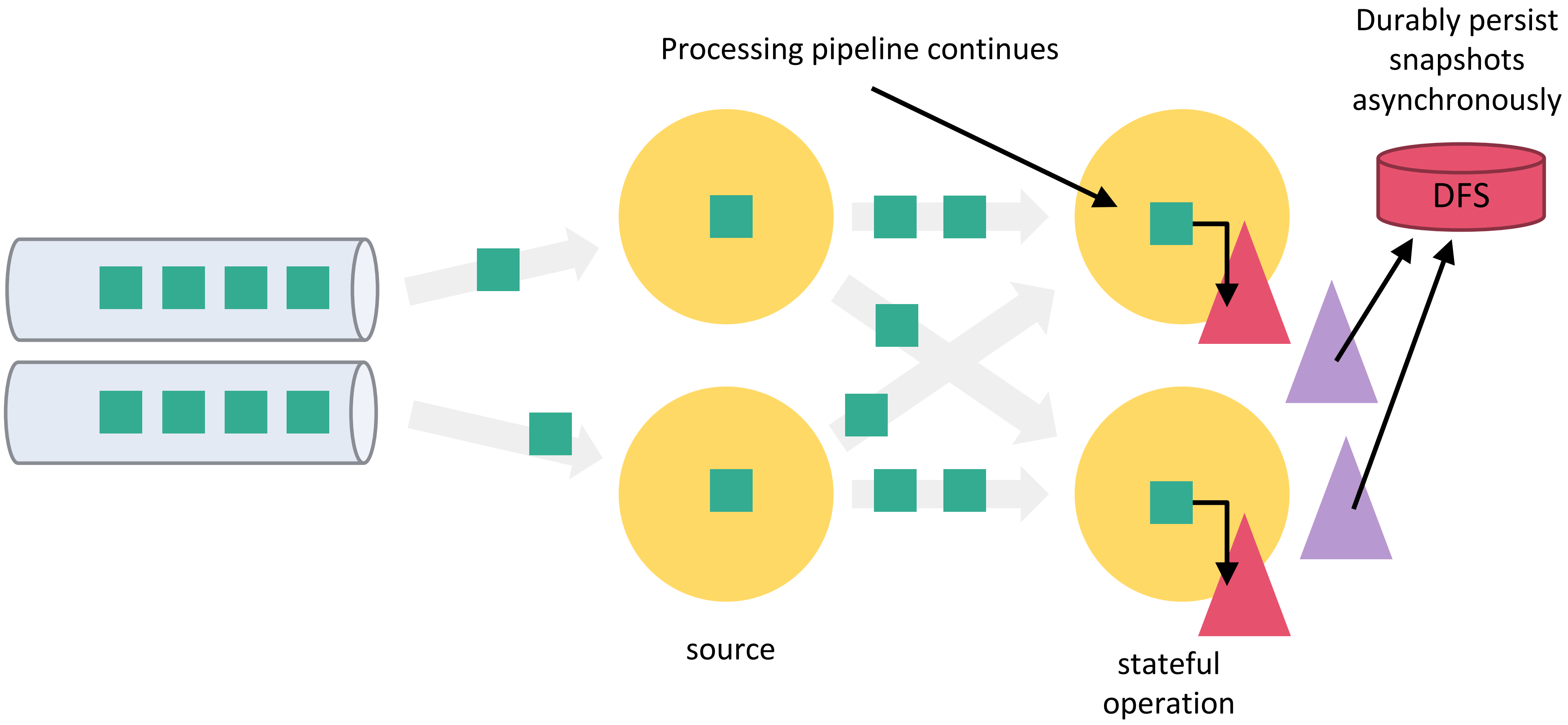
Distributed snapshots



Distributed snapshots



Distributed snapshots


Processing pipeline continues
Durably persist snapshots asynchronously
DFS
source
stateful operation

Enabling Checkpointing

- Checkpointing is disabled by default.
- Enable checkpointing with exactly once consistency:

```
// checkpoint every 5 seconds
env.enableCheckpointing(5000)
```

- Configure at least once consistency (for lower latency):

```
env.getCheckpointConfig()
   .setCheckpointingMode(CheckpointingMode.AT_LEAST_ONCE);
```

- Most applications perform well with a few seconds checkpointing interval.

Restart Strategies

- How often and fast does a job try to restart?

- Available strategies
  - No restart (default)
  - Fixed delay
  - Failure rate

```
// Fixed Delay restart strategy
env.setRestartStrategy(
  RestartStrategies.fixedDelayRestart(
    3,                             // no of restart attempts
    Time.of(10, TimeUnit.SECONDS) // restart interval
));
```

State Backends

State in Flink

- There are several sources of state in Flink
  - Windows
  - User functions
  - Sources and Sinks
  - Timers

- State is persisted during checkpoints, if checkpointing is enabled

- Internal representation and storage location depend on the configured State Backend

Choosing a State Backend

Name	Working state	State backup	Snapshotting
RocksDBStateBackend	Local disk (tmp directory)	Distributed file system	Asynchronously
• Good for state larger than available memory • 10x slower than memory-based backends			
FsStateBackend	JVM Heap	Distributed file system	Synchronous / Async option in Flink 1.3
• Fast, requires large heap			
MemoryStateBackend	JVM Heap	JobManager JVM Heap	Synchronous / Async option in Flink 1.3
• Good for testing and experimentation with small state (locally)			

State Backend Configuration

- Configuration of default state backend in

```
./conf/flink-conf.yaml
```

- State backend configuration in job

```
env.setStateBackend(
    new FsStateBackend(
      "hdfs://namenode:40010/flink/checkpoints”
  ));
```

Savepoints

State management

Two important management concerns for a long-running job:

- Can I change / fix bugs in my streaming pipeline? How do I handle job downtime?

- Can I rescale (change parallelism of) my computation?

State management: 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

Savepoints

- A *Checkpoint* is a globally consistent point-in-time snapshot of a streaming application *(point in stream, state)*

- *Savepoints* are user-triggered, retained checkpoints

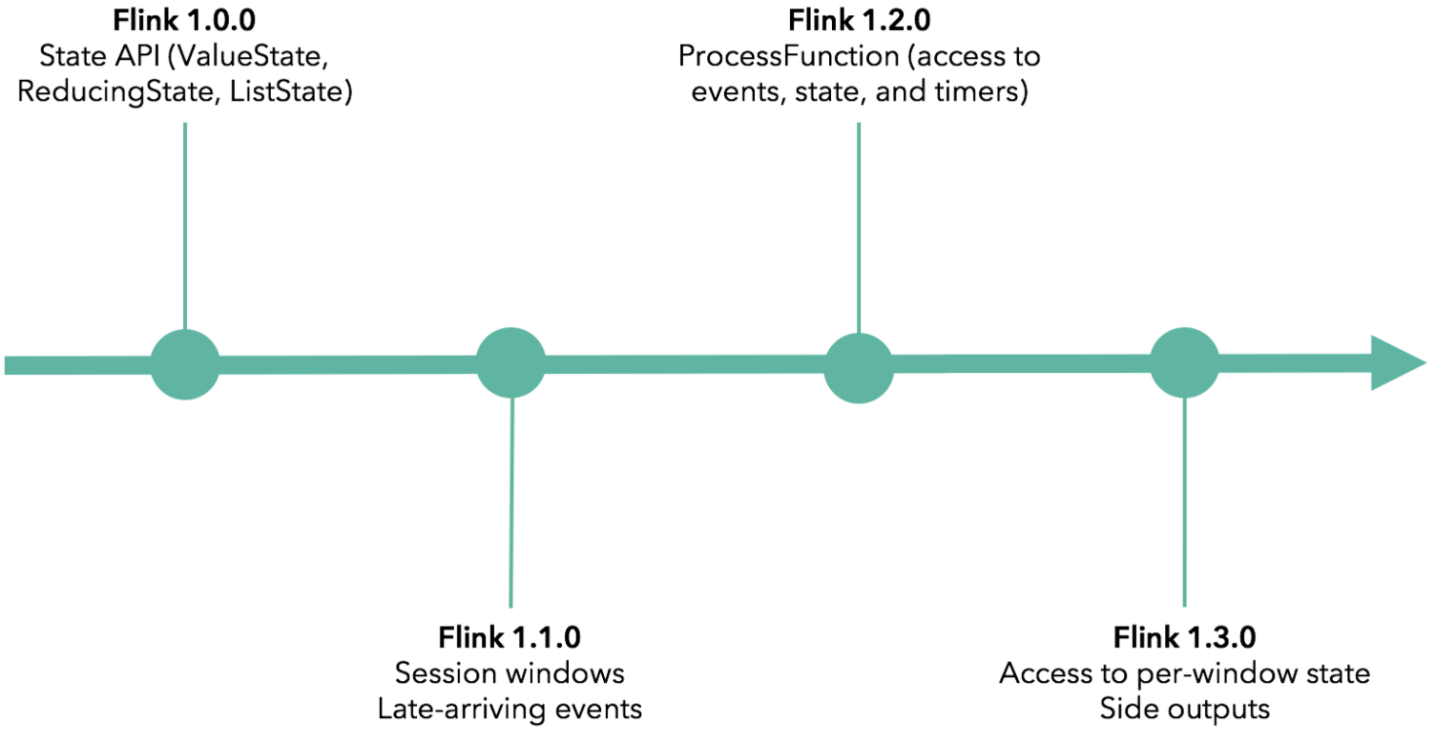
- Rescaling (currently) requires restarting from a savepoint

- Currently, Flink can only restore to the same state backend that created the savepoint

Evolution of Flink

(w.r.t. Stateful Stream Processing)

Evolution of Programming APIs



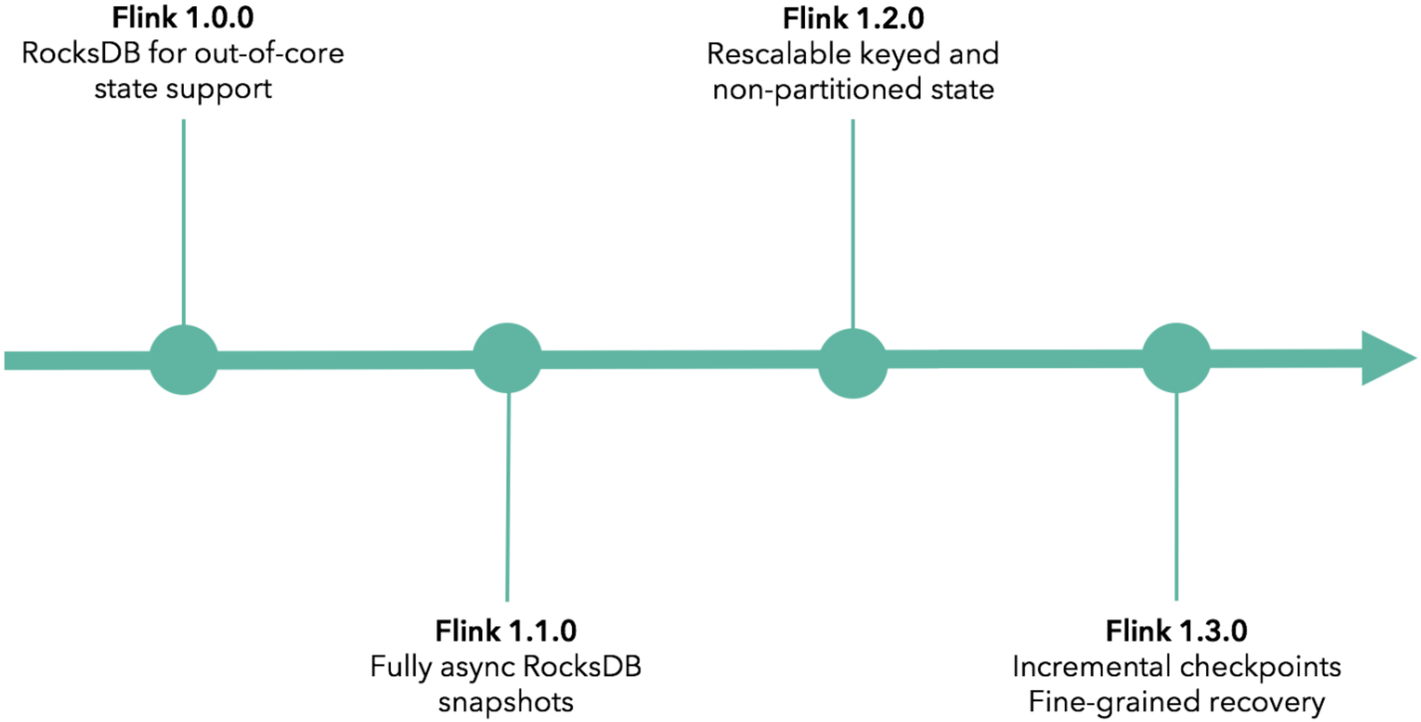
Flink 1.0.0
State API (ValueState, ReducingState, ListState)

Flink 1.1.0
Session windows
Late-arriving events

Flink 1.2.0
ProcessFunction (access to events, state, and timers)

Flink 1.3.0
Access to per-window state
Side outputs

Evolution of Large State Handling



Flink 1.0.0
RocksDB for out-of-core state support

Flink 1.1.0
Fully async RocksDB snapshots

Flink 1.2.0
Rescalable keyed and non-partitioned state

Flink 1.3.0
Incremental checkpoints
Fine-grained recovery