TUNING FLINK FOR ROBUSTNESS AND PERFORMANCE

STEFAN RICHTER

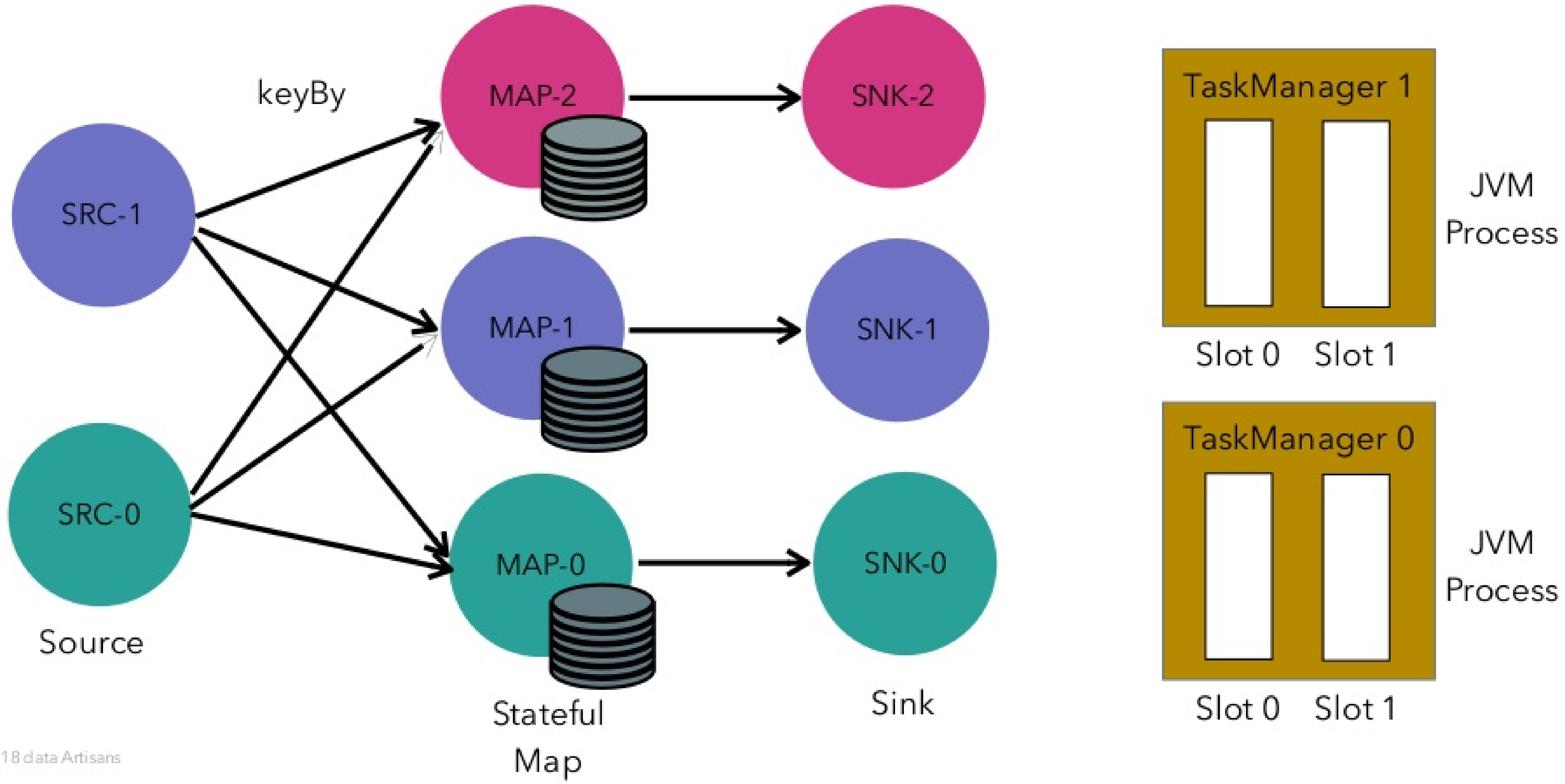
@STEFANRRICHTER

SEPT 4, 2018

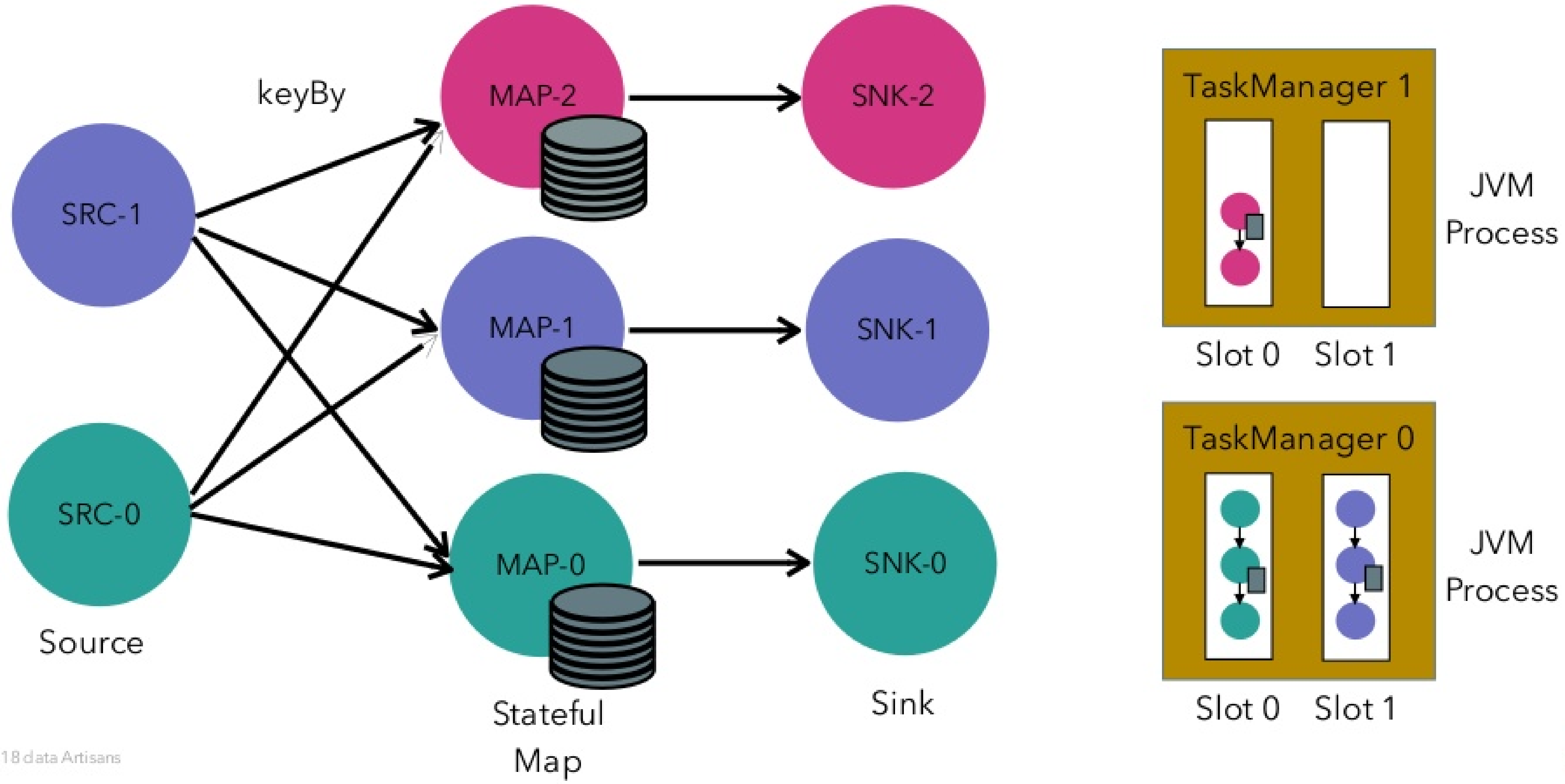
dataArtisans

GENERAL MEMORY CONSIDERATIONS

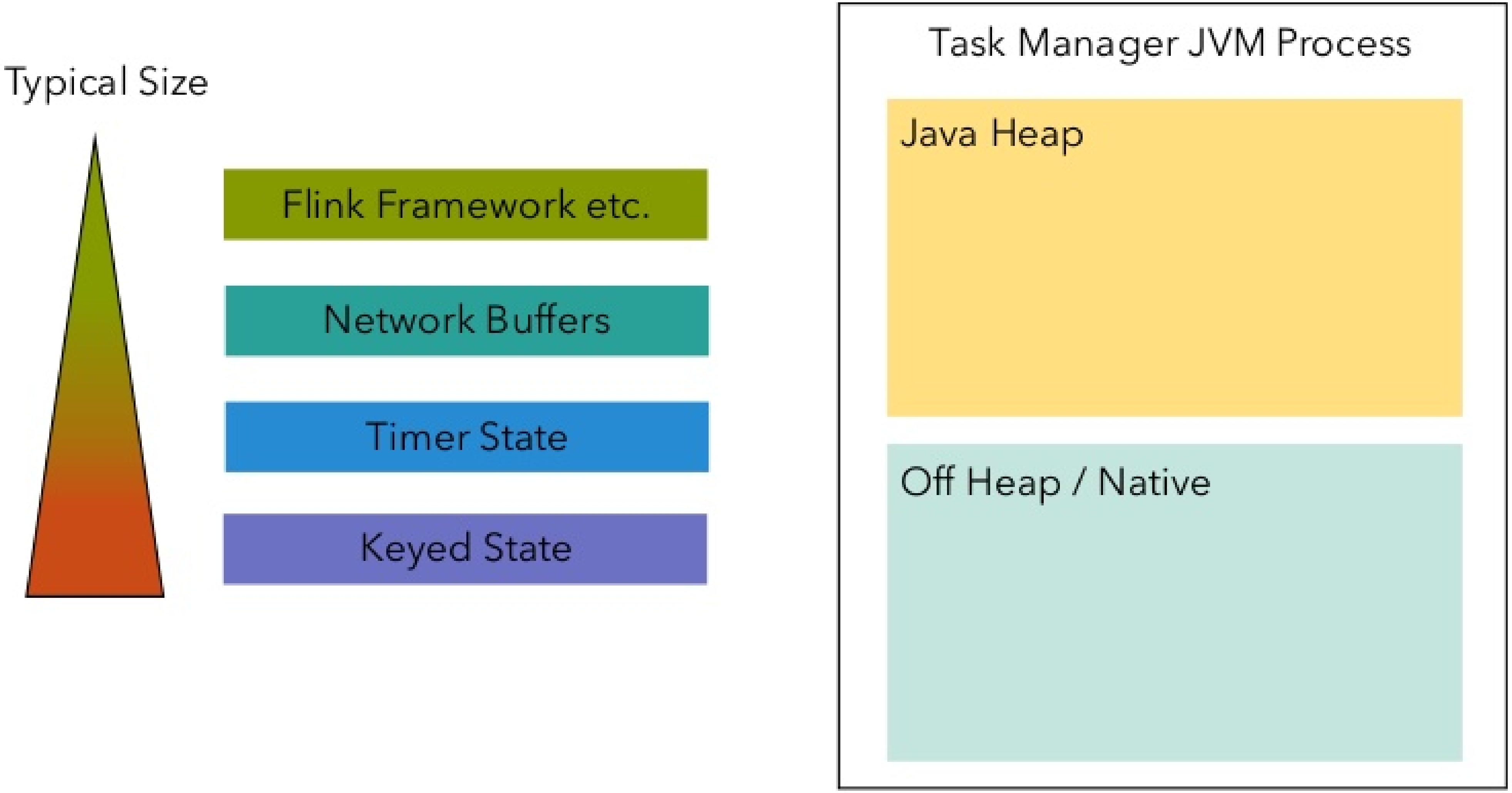
BASIC TASK SCHEDULING



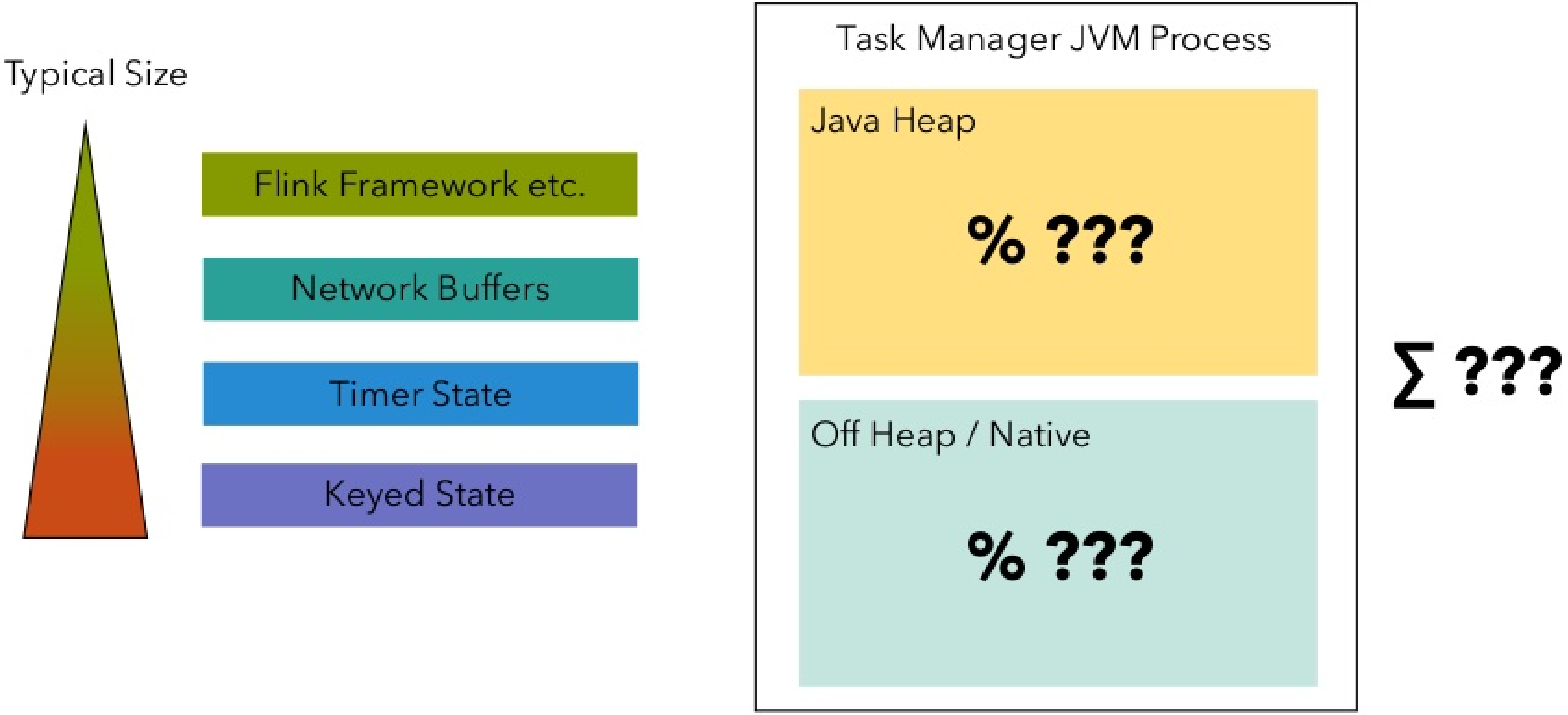
BASIC TASK SCHEDULING



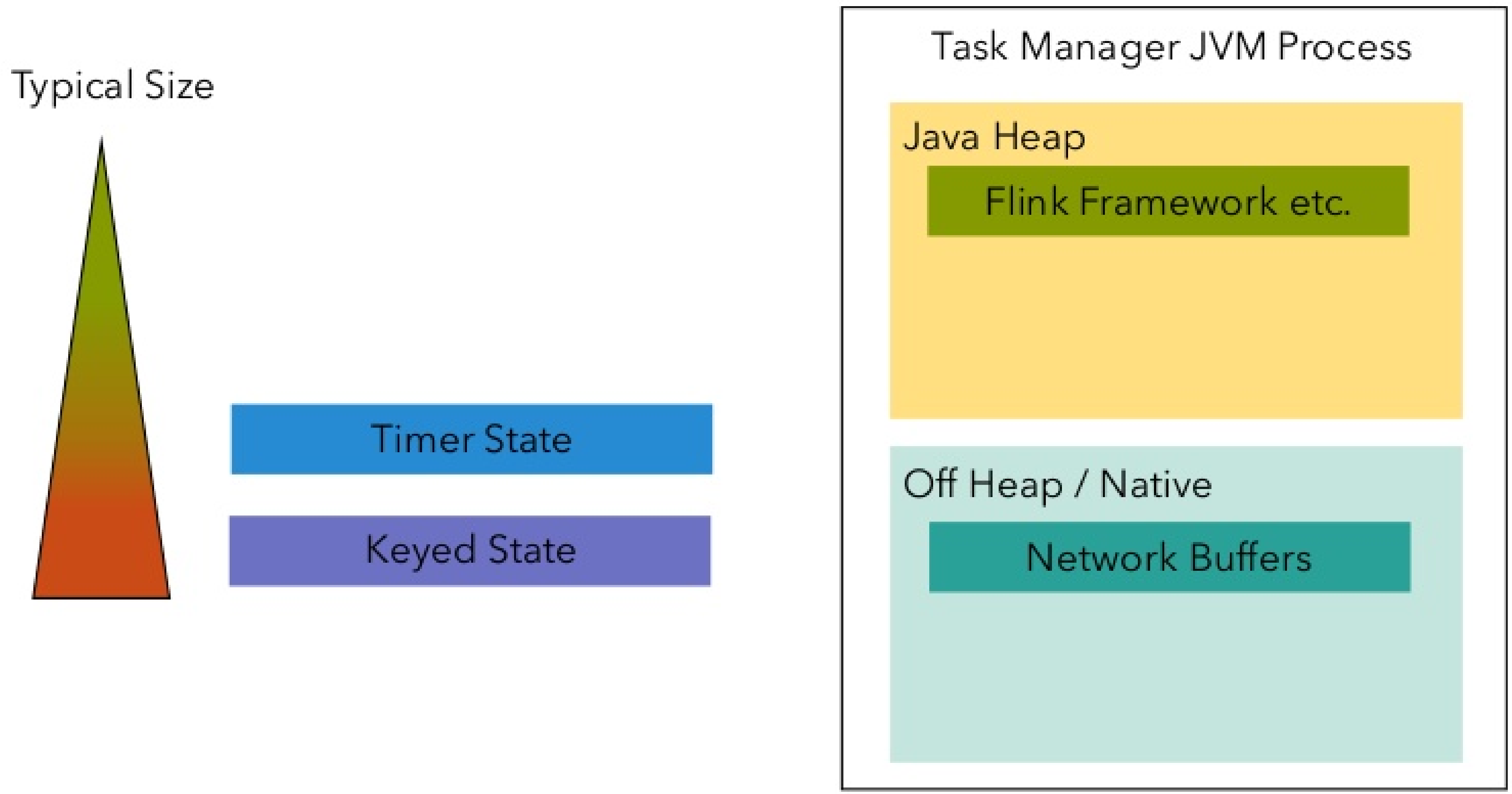
TASK MANAGER PROCESS MEMORY LAYOUT



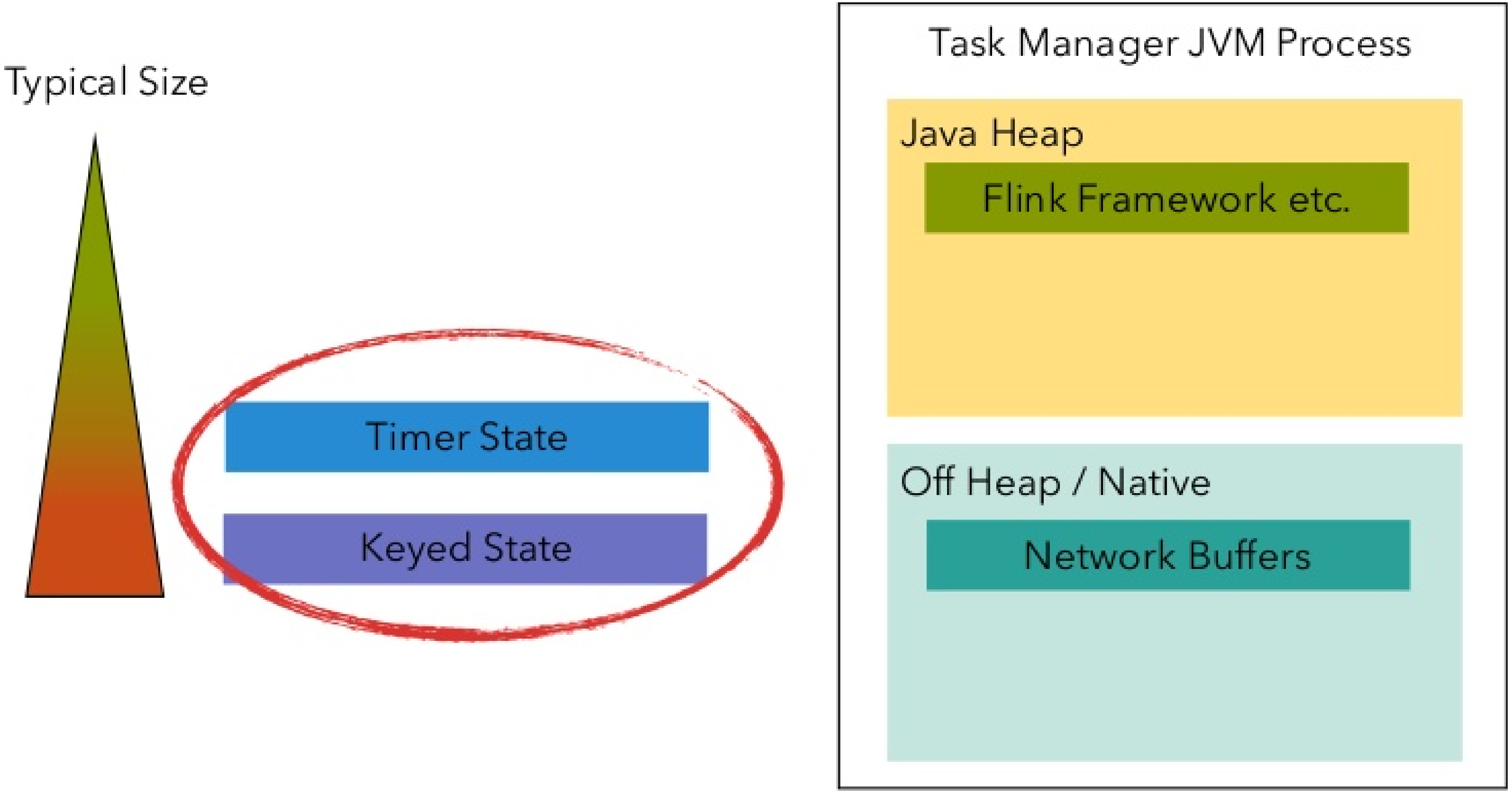
TASK MANAGER PROCESS MEMORY LAYOUT



TASK MANAGER PROCESS MEMORY LAYOUT

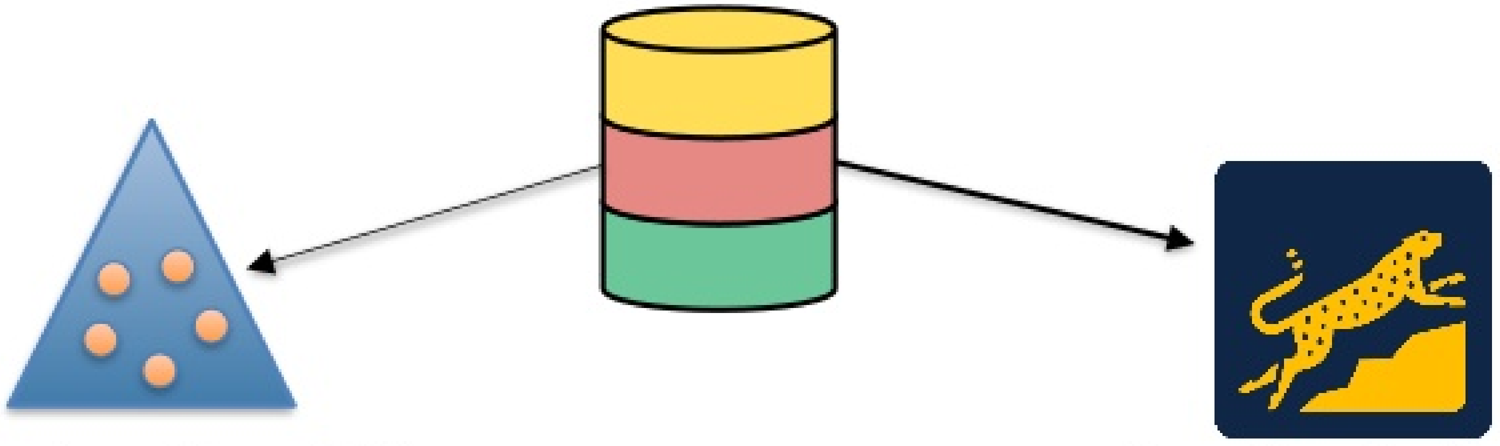


TASK MANAGER PROCESS MEMORY LAYOUT



STATE BACKENDS

FLINK KEYED STATE BACKENDS CHOICES



Based on Java Heap Objects

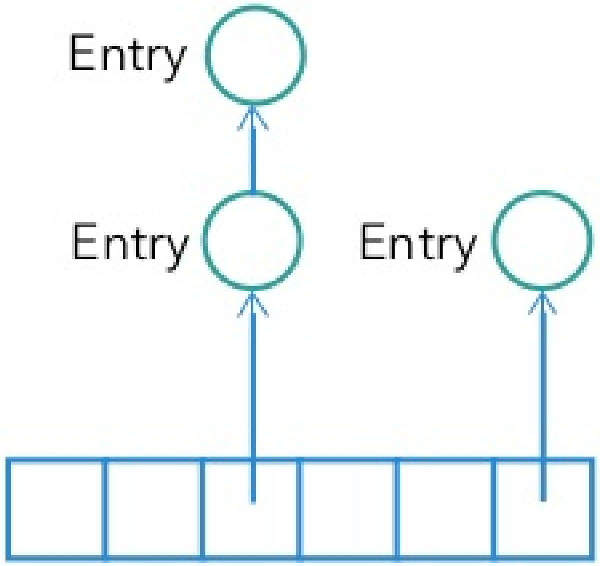
Based on RocksDB

HEAP KEYED STATE BACKEND CHARACTERISTICS

- State lives as Java objects on the heap.
- Organized as chained hash table, key ↦ state.
- One hash table per registered state.
- Supports asynchronous state snapshots through copy-on-write MVCC.
- Data is de/serialized only during state snapshot and restore.

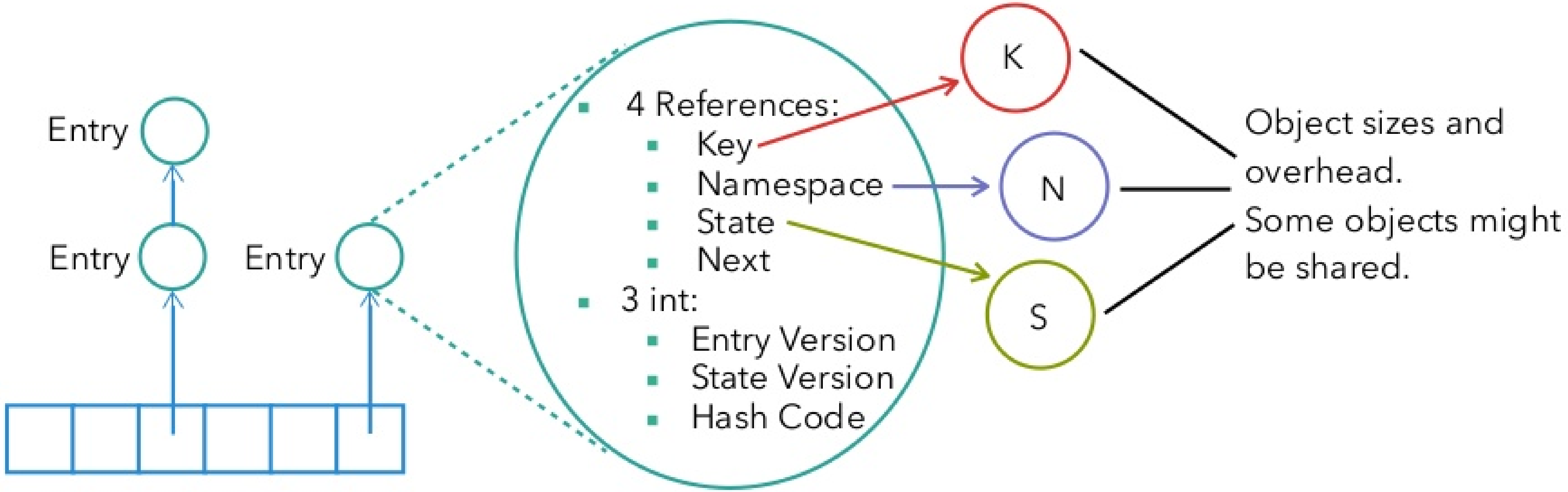
- Highest performance.
- Affected by garbage collection overhead / pauses.
- Currently no incremental checkpoints.
- Memory overhead of representation.
- State size limited by available heap memory.

HEAP STATE TABLE ARCHITECTURE



- Hash buckets (Object[]), 4B-8B per slot
- Load factor <= 75%
- Incremental rehash

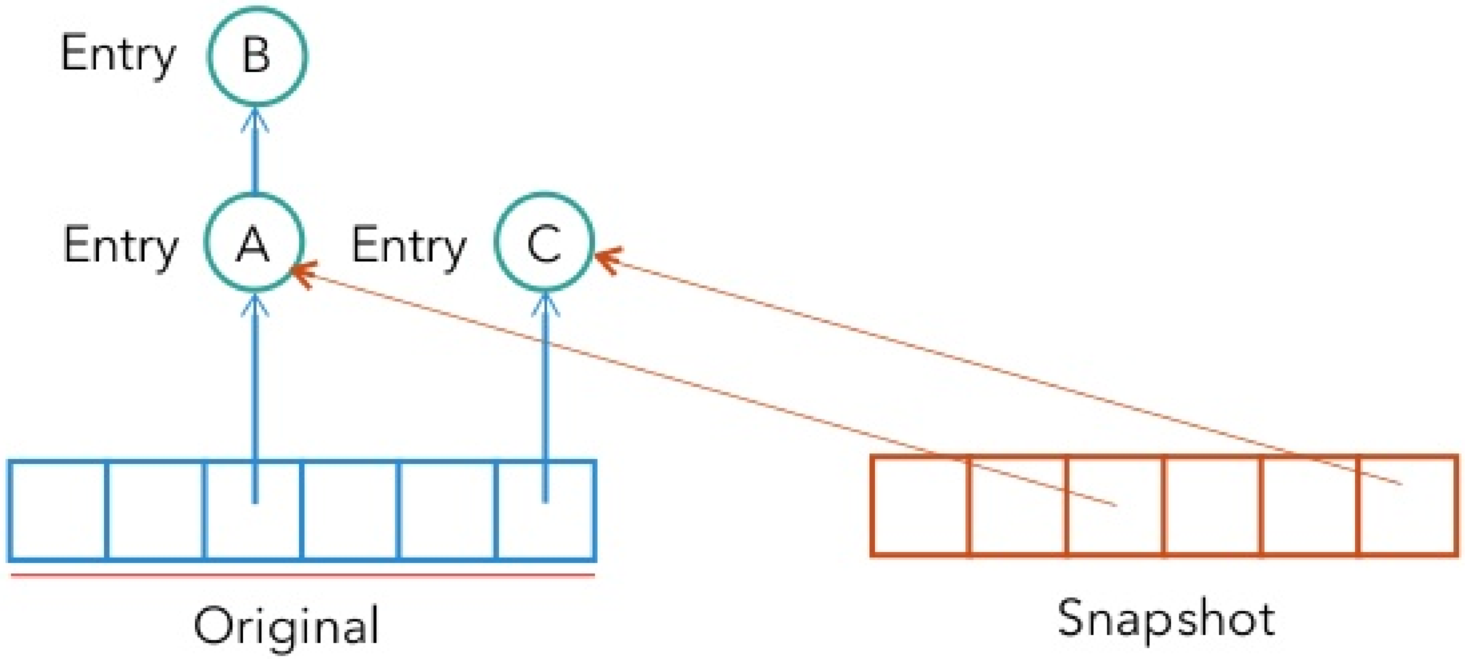
HEAP STATE TABLE ARCHITECTURE



- Hash buckets (Object[]), 4B-8B per slot
- Load factor <= 75%
- Incremental rehash

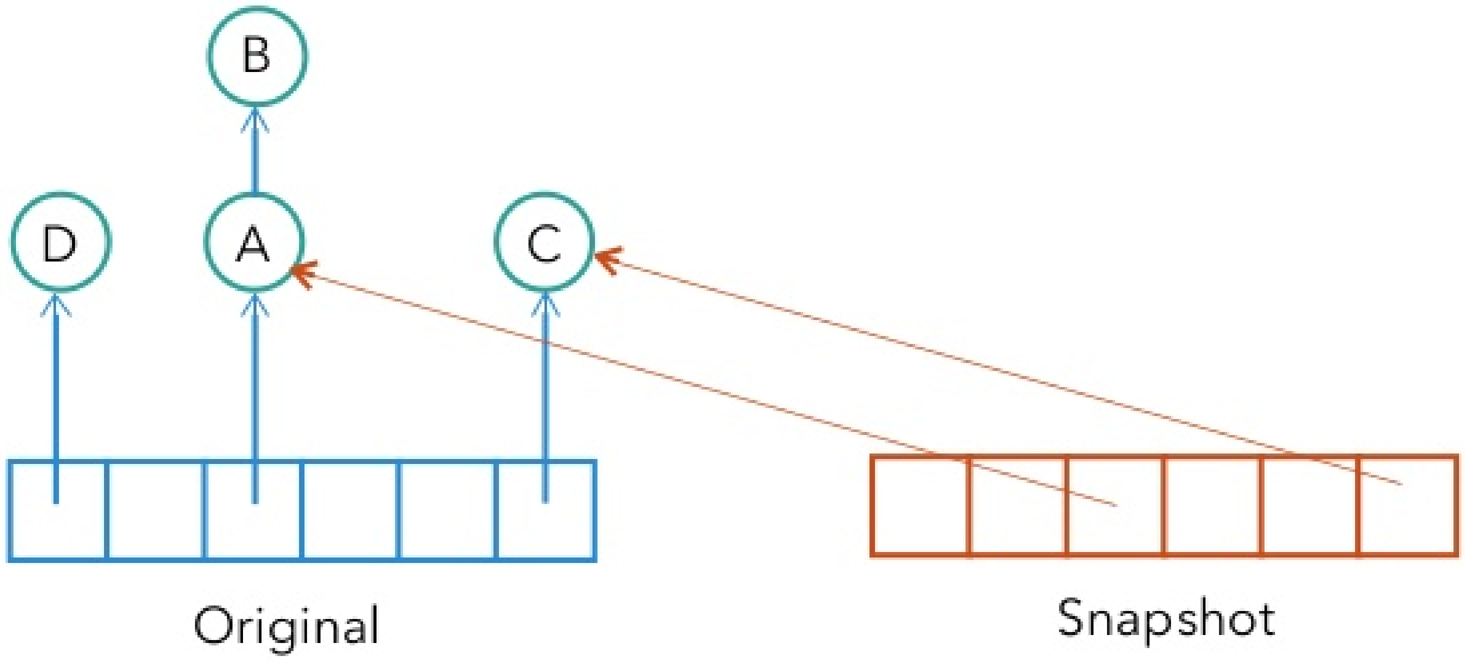
4 x (4B-8B)
+ 3 x 4B
+ ~8B-16B (Object overhead)

HEAP STATE TABLE SNAPSHOT MVCC



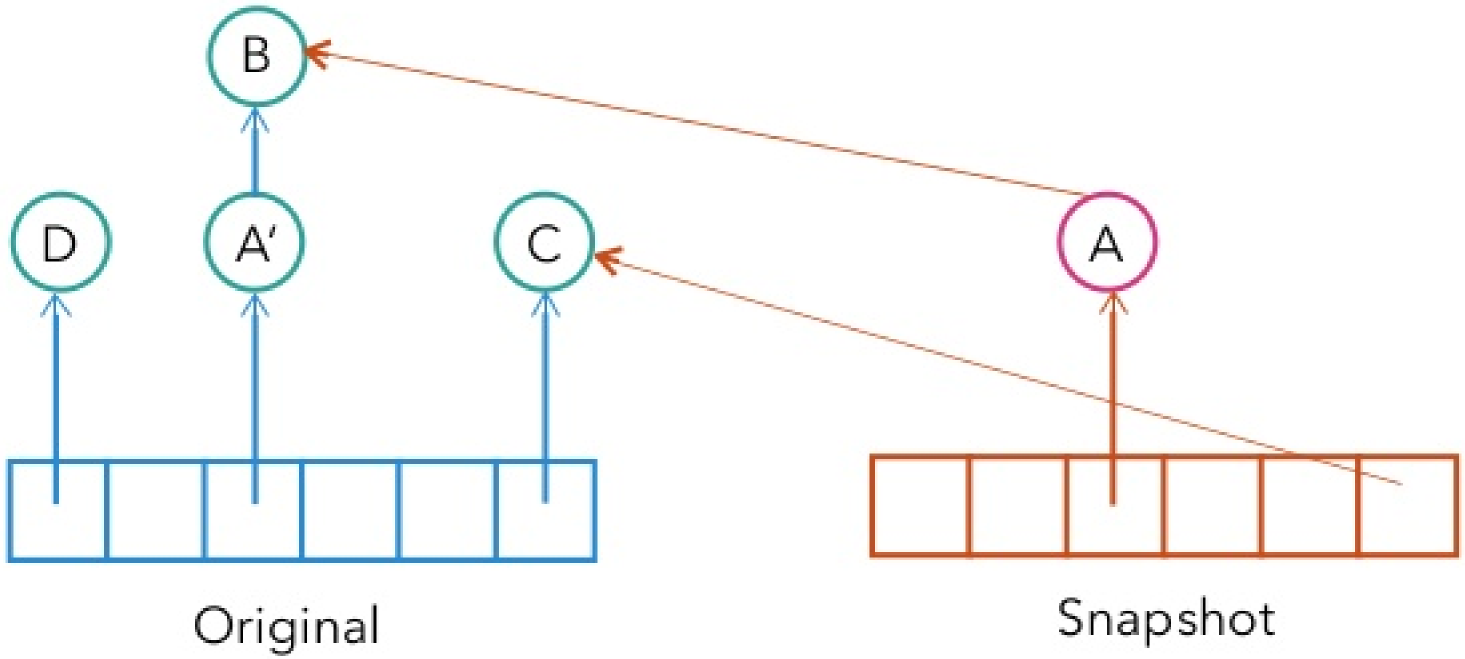
Copy of hash bucket array is snapshot overhead

HEAP STATE TABLE SNAPSHOT MVCC



No conflicting modification = no overhead

HEAP STATE TABLE SNAPSHOT MVCC



Modifications trigger deep copy of entry - only as much as required. This depends on what was modified and what is immutable (as determined by type serializer).
Worst case overhead = size of original state table at time of snapshot.

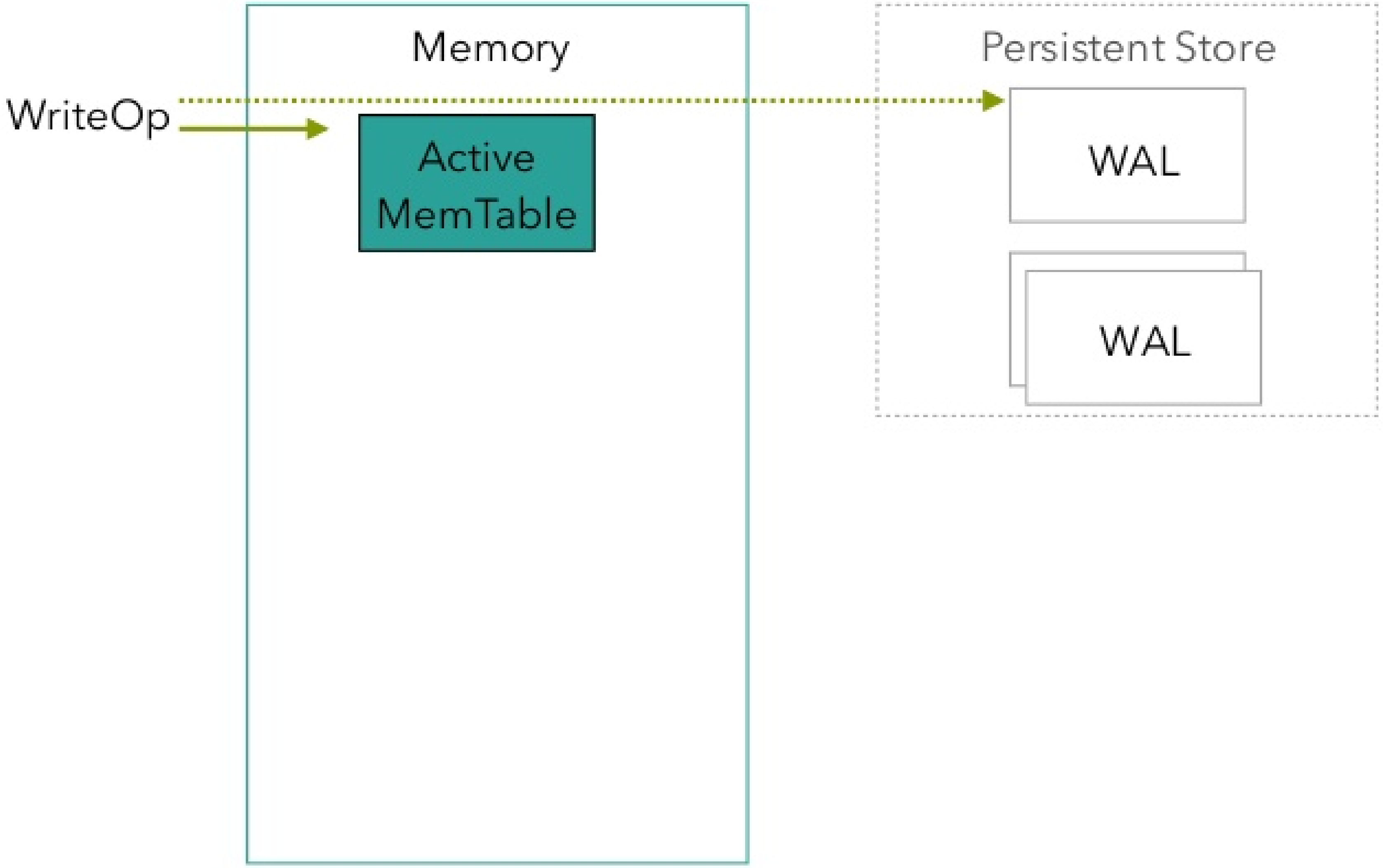
HEAP BACKEND TUNING CONSIDERATIONS

- Chose type serializer with efficient copy-method (for copy-on-write).
- Flag immutability of objects where possible to avoid copy completely.
- Flatten POJOs / avoid deep objects. Reduces object overheads and following references = potential cache misses.
- GC choice/tuning can help. Follow future GC developments.
- Scale-out using multiple task manager per node to support larger state over multiple heap backends rather than having fewer and large heaps.

ROCKSDB KEYED STATE BACKEND CHARACTERISTICS

- State lives as serialized byte-strings in off-heap memory and on local disk.
- Key-Value store, organized as log-structured merge tree (LSM-tree).
  - Key: serialized bytes of <Keygroup, Key, Namespace>.
  - Value: serialized bytes of the state.
- One column family per registered state (~table).
- LSM naturally supports MVCC.
- Data is de/serialized on every read and update.
- Not affected by garbage collection.
- Relative low overhead of representation.
- LSM naturally supports incremental snapshots.
- State size limited by available local disk space.
- Lower performance (~order of magnitude compared to Heap state backend).

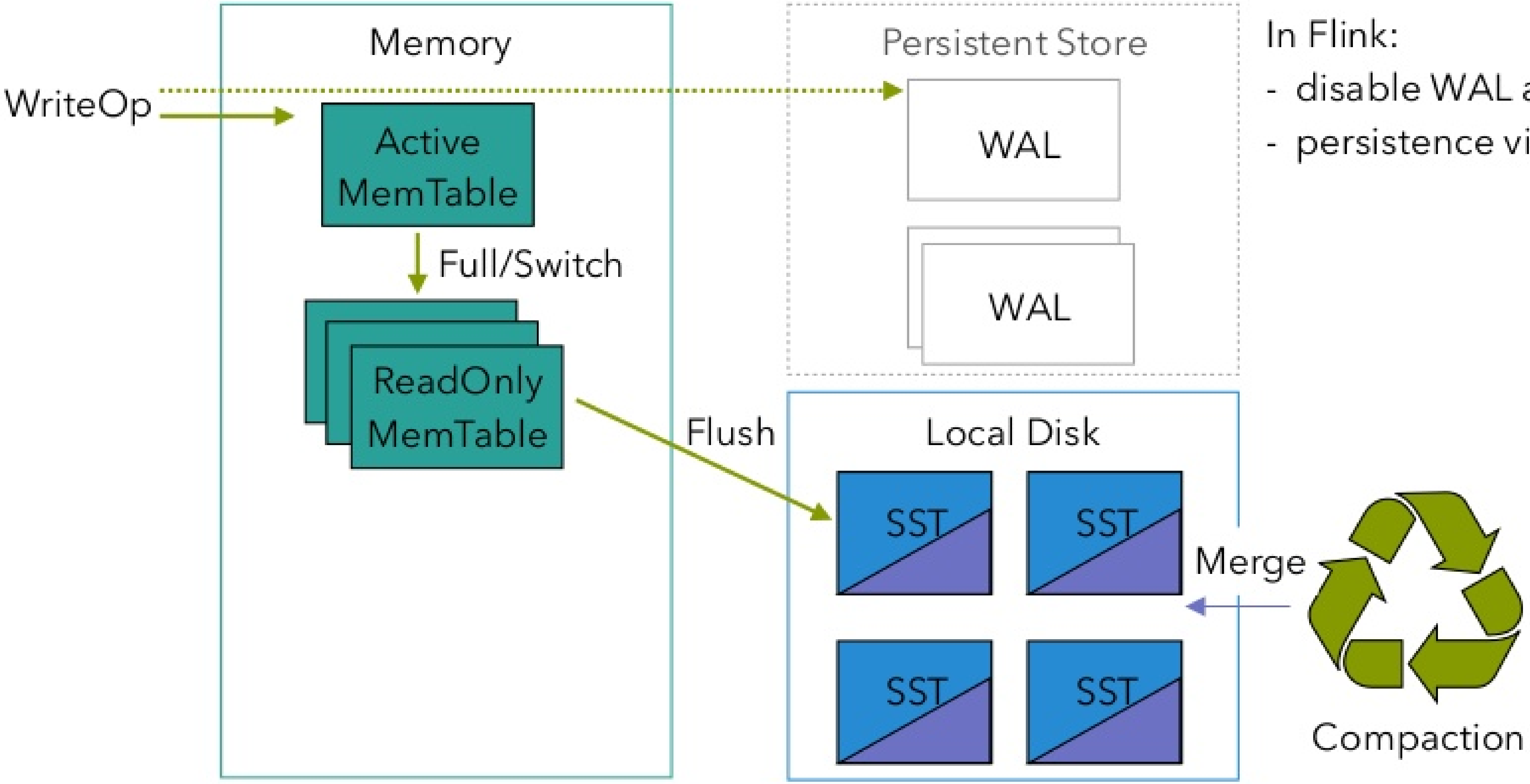
ROCKSDB ARCHITECTURE



In Flink:

- disable WAL and sync
- persistence via checkpoints

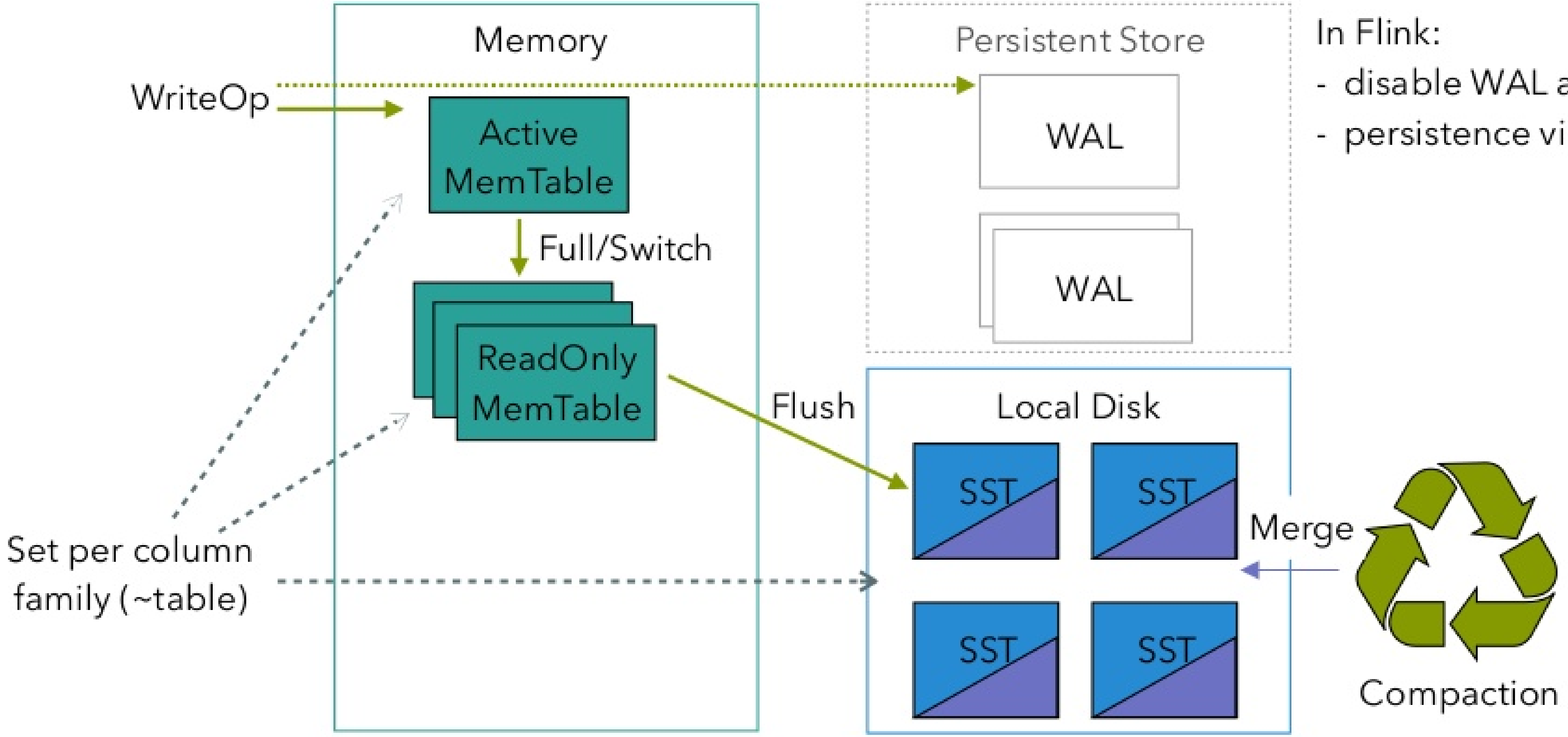
ROCKSDB ARCHITECTURE



In Flink:

- disable WAL and sync
- persistence via checkpoints

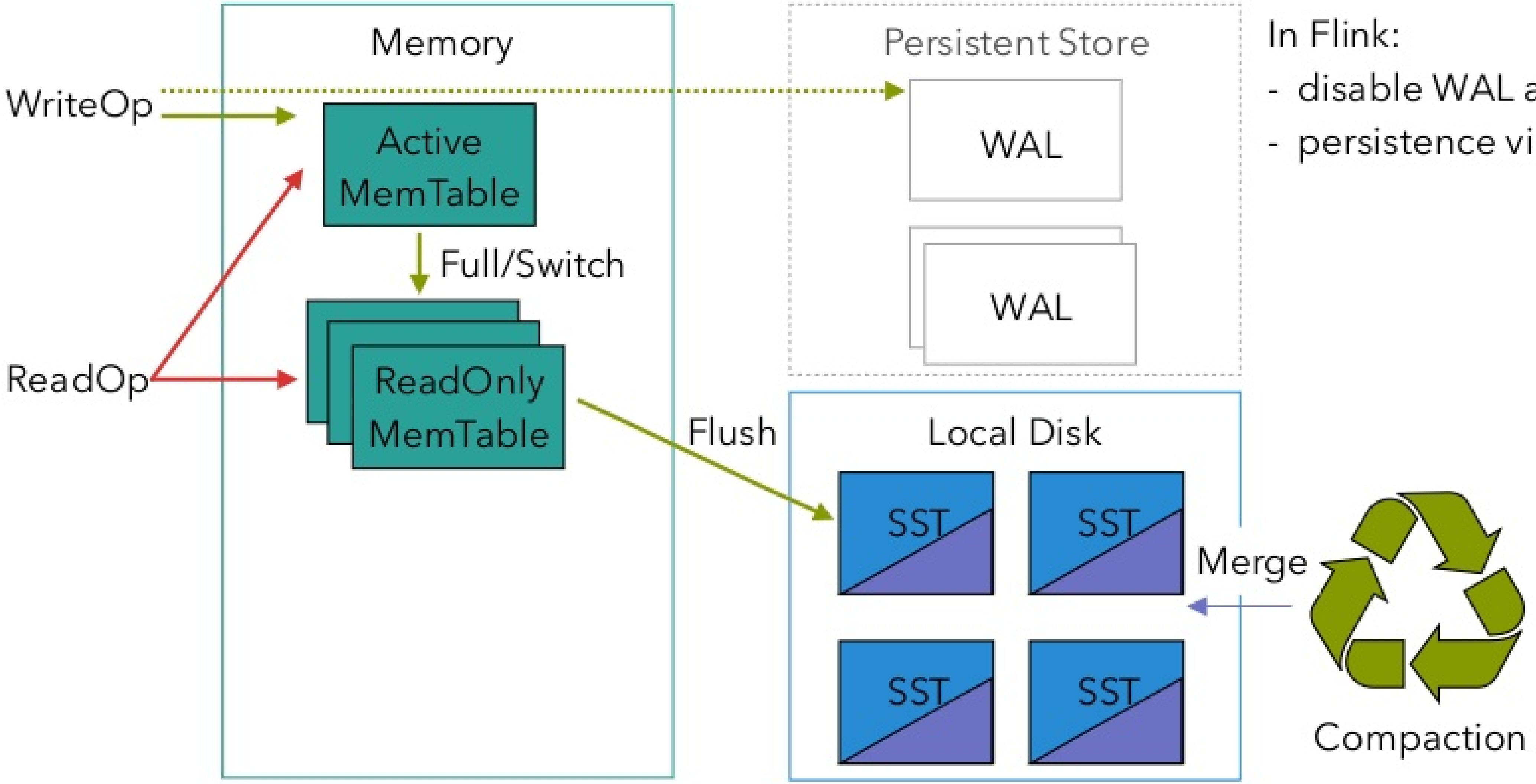
ROCKSDB ARCHITECTURE



In Flink:

- disable WAL and sync
- persistence via checkpoints

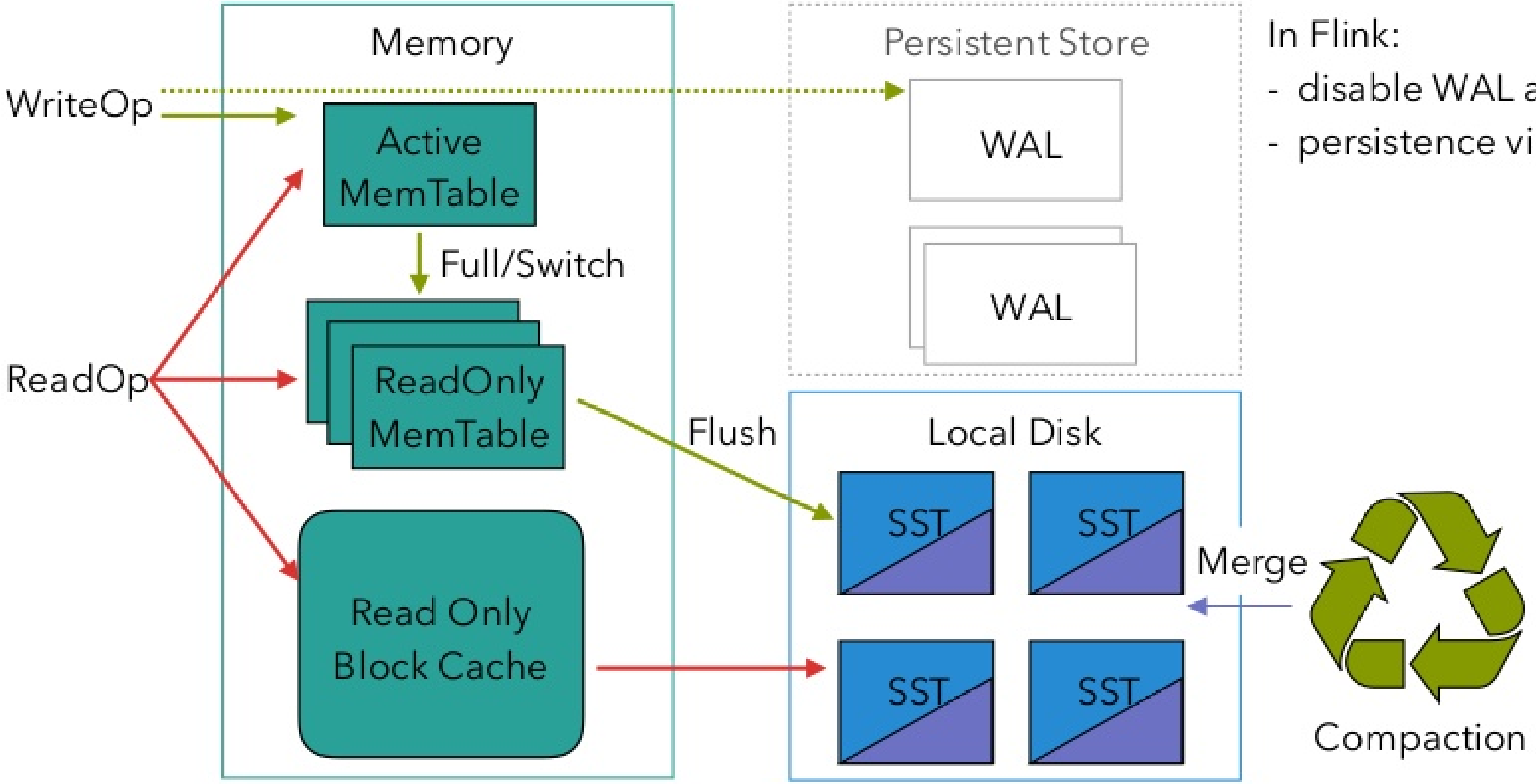
ROCKSDB ARCHITECTURE



In Flink:

- disable WAL and sync
- persistence via checkpoints

ROCKSDB ARCHITECTURE



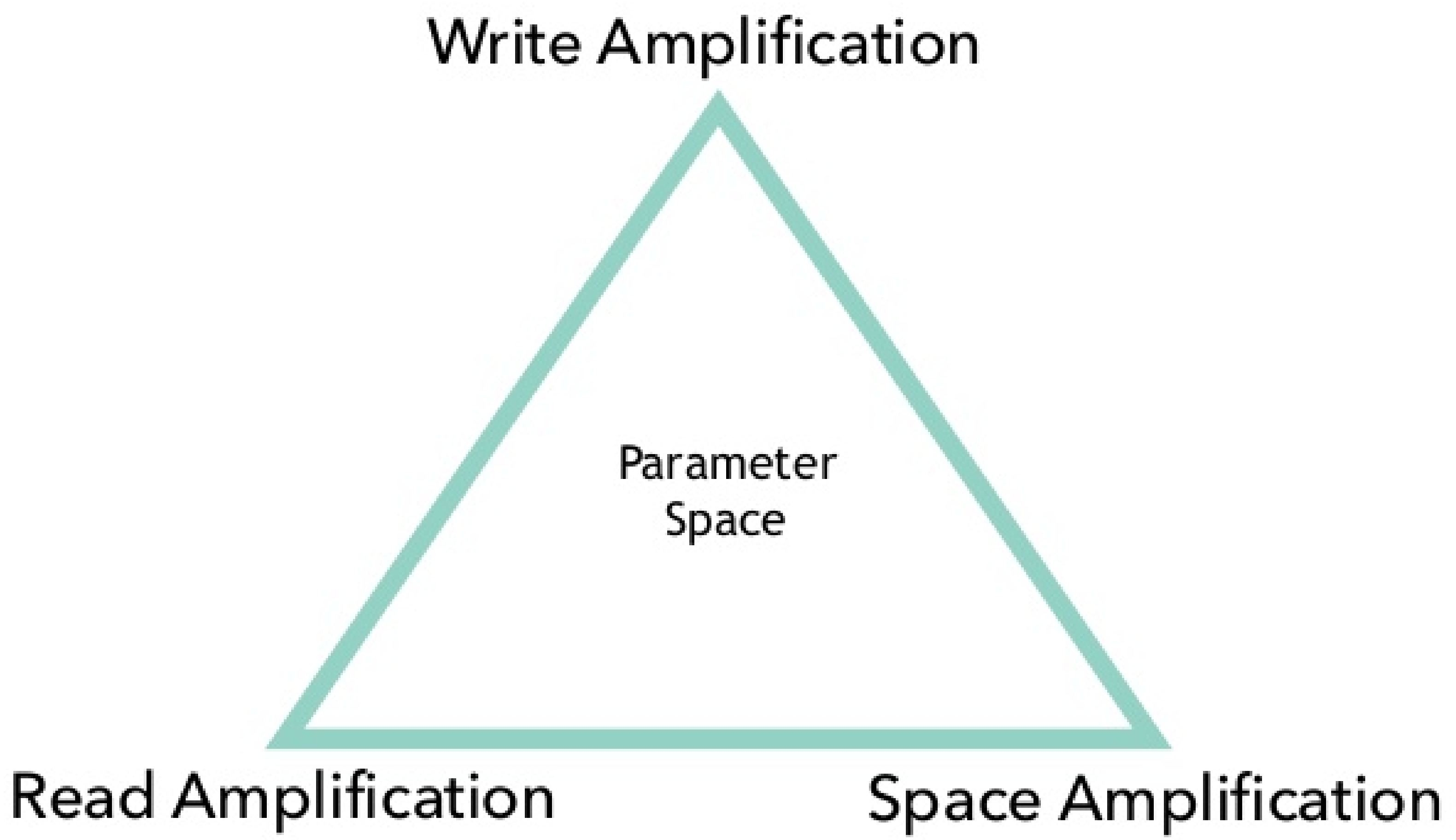
In Flink:

- disable WAL and sync
- persistence via checkpoints

ROCKSDB RESOURCE CONSUMPTION

- One RocksDB instance per keyed operator subtask.
- **block_cache_size**:
  - Size of the block cache.
- **write_buffer_size**:
  - Max. size of a MemTable.
- **max_write_buffer_number**:
  - The maximum number of MemTables in memory before flush to SST files.
- **Indexes and bloom filters** (optional).
- **Table Cache**:
  - Caches open file descriptors to SST files. Default: unlimited!

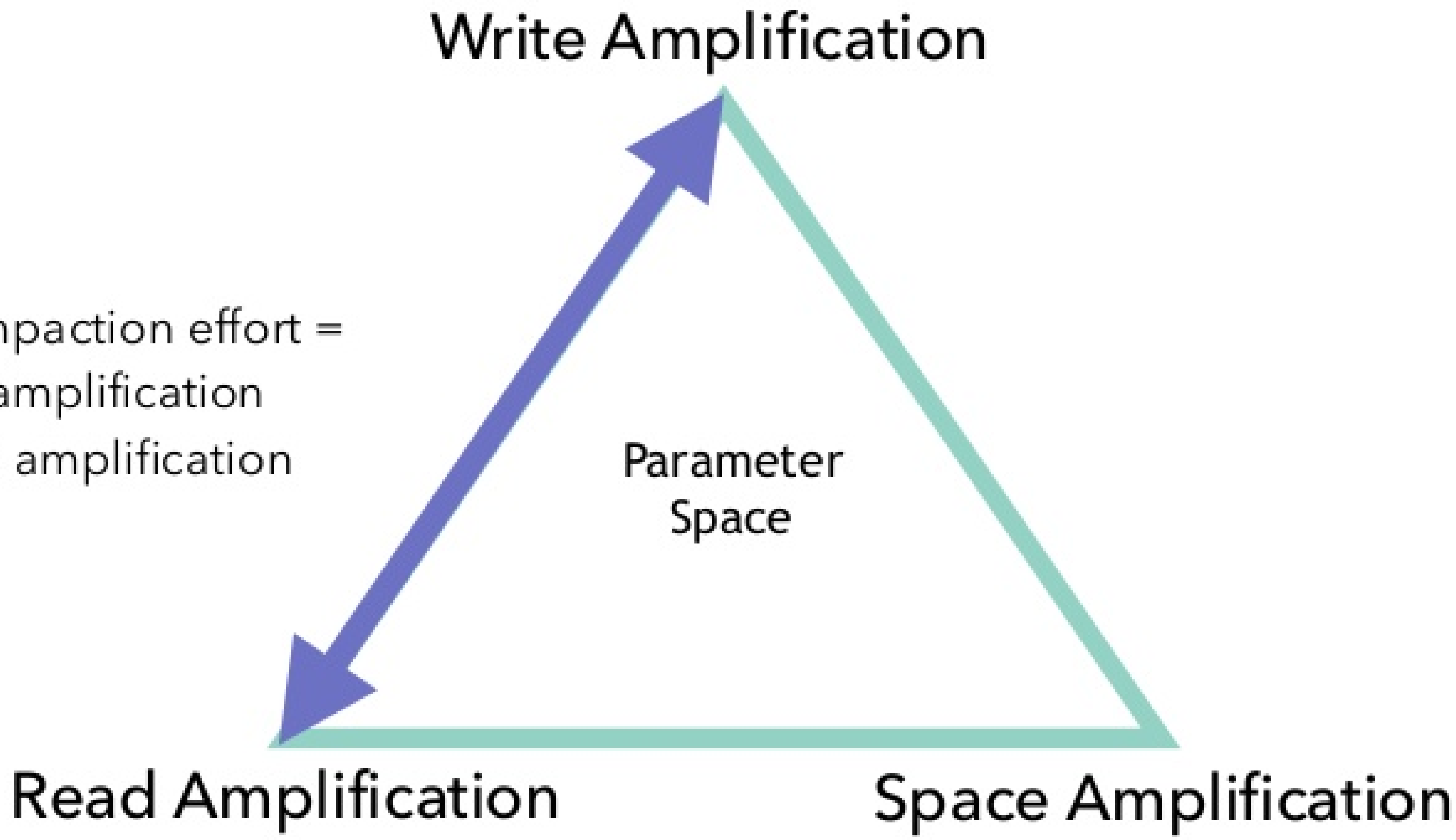
PERFORMANCE TUNING - AMPLIFICATION FACTORS



More details: https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide

PERFORMANCE TUNING - AMPLIFICATION FACTORS

Example: More compaction effort = increased write amplification and reduced read amplification



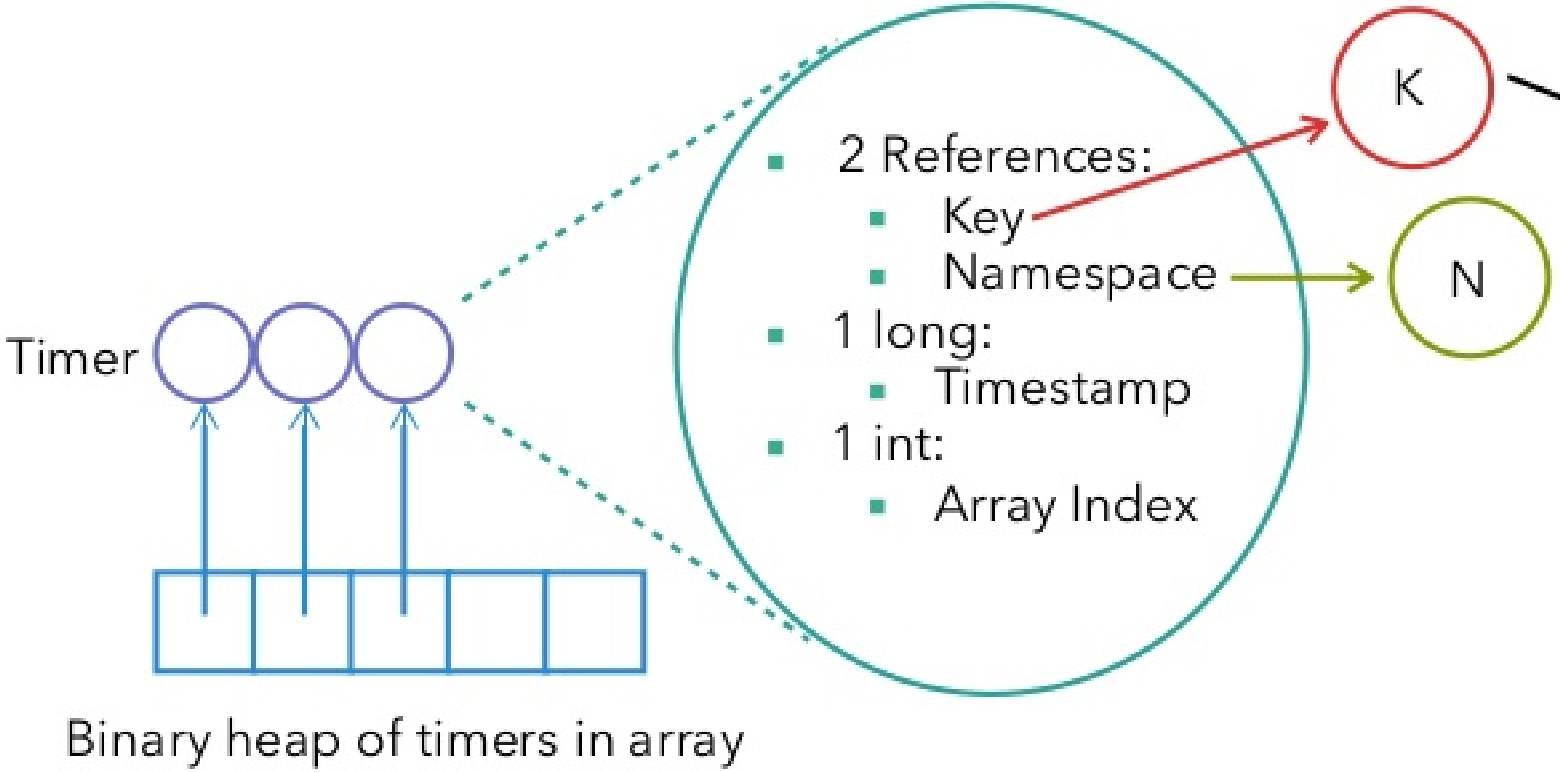
More details: https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide

GENERAL PERFORMANCE CONSIDERATIONS

- Efficient type serializer and serialization formats.
- Decompose user-code objects: business logic / efficient state representation.
- Extreme: „Flightweight Pattern“, e.g. wrapper object that interprets/manipulates stored byte array on the fly and uses only byte-array type serializer.
- File Systems:
    - Working directory on fast storage, ideally local SSD. Could even be memory file system because it is transient for Flink. EBS performance can be problematic.
    - Checkpoint directory: Persistence happens here. Can be slower but should be fault tolerant.

TIMER SERVICE

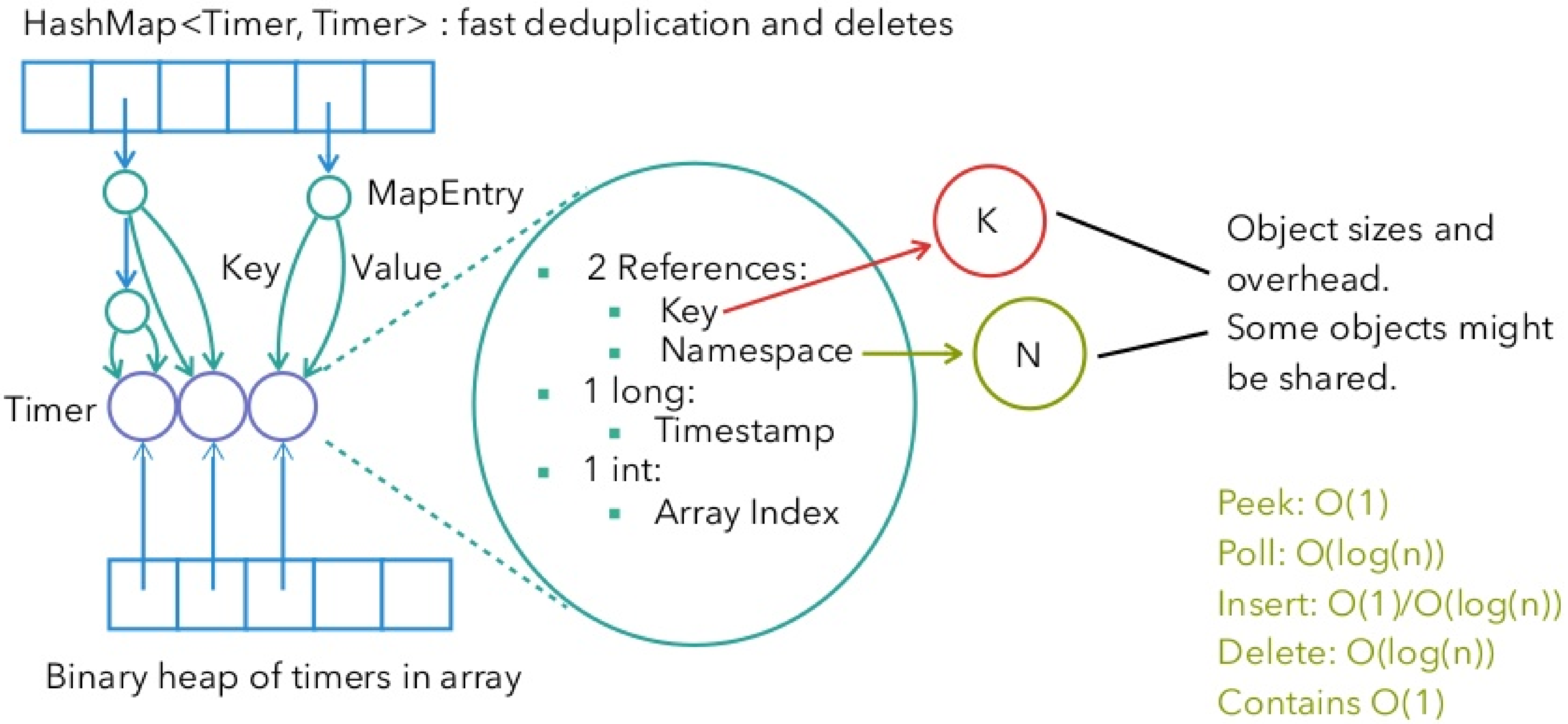
HEAP TIMERS



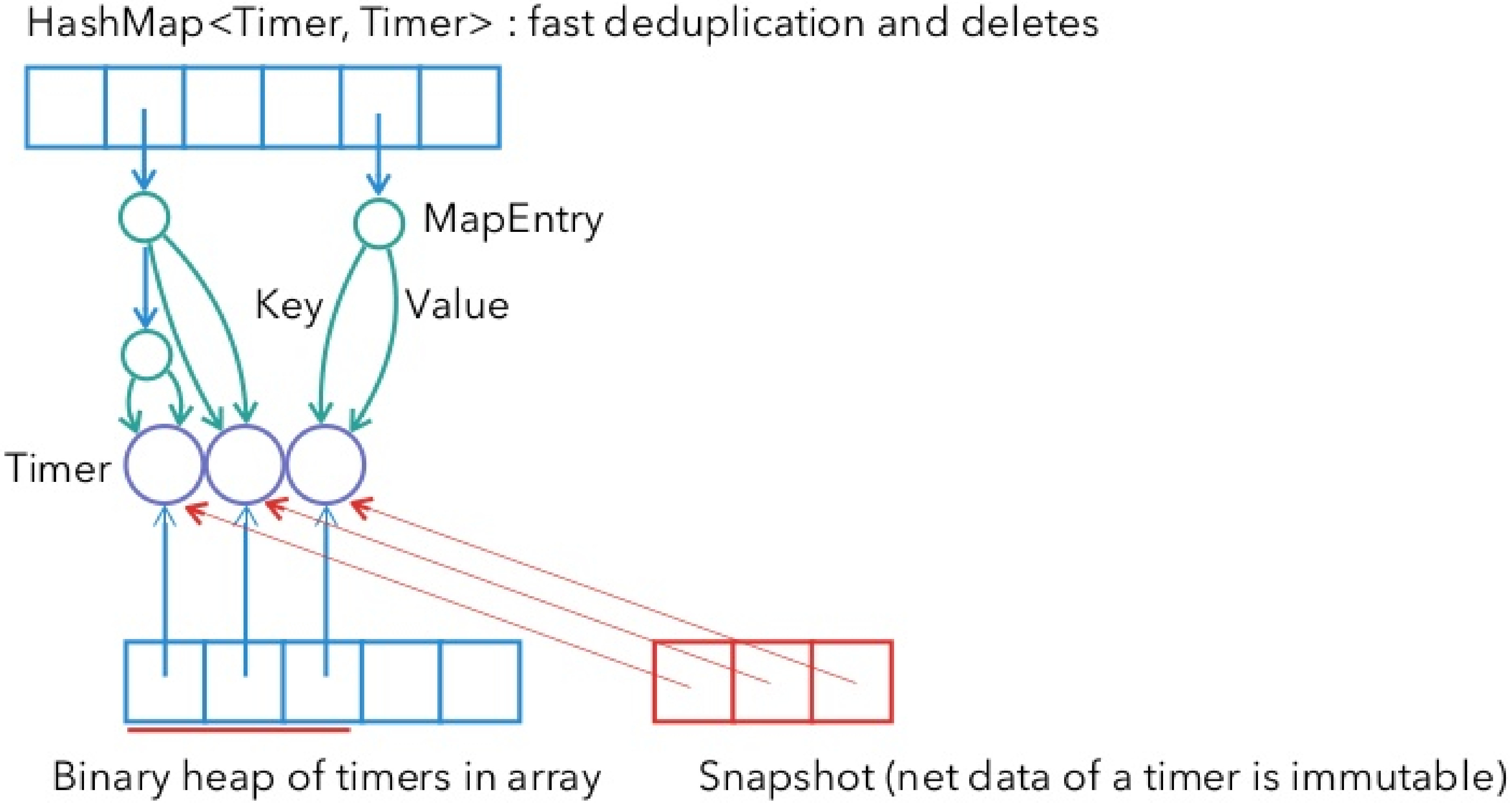
Object sizes and overhead.
Some objects might be shared.

Peek: O(1)
Poll: O(log(n))
Insert: O(1)/O(log(n))
Delete: O(n)
Contains O(n)

HEAP TIMERS


HashMap<Timer, Timer> : fast deduplication and deletes
MapEntry
Key
Value
Timer
Binary heap of timers in array
2 References:
Key
Namespace
1 long:
Timestamp
1 int:
Array Index
K
N
Object sizes and overhead.
Some objects might be shared.
Peek: O(1)
Poll: O(log(n))
Insert: O(1)/O(log(n))
Delete: O(log(n))
Contains O(1)

HEAP TIMERS


HashMap<Timer, Timer> : fast deduplication and deletes
MapEntry
Key
Value
Timer
Binary heap of timers in array
Snapshot (net data of a timer is immutable)

ROCKSDB TIMERS

Column Family - only key, no value

Key Group	Time stamp	Key	Name space
0	20	A	X
0	40	D	Z

...

1	10	D	Z
1	20	C	Y

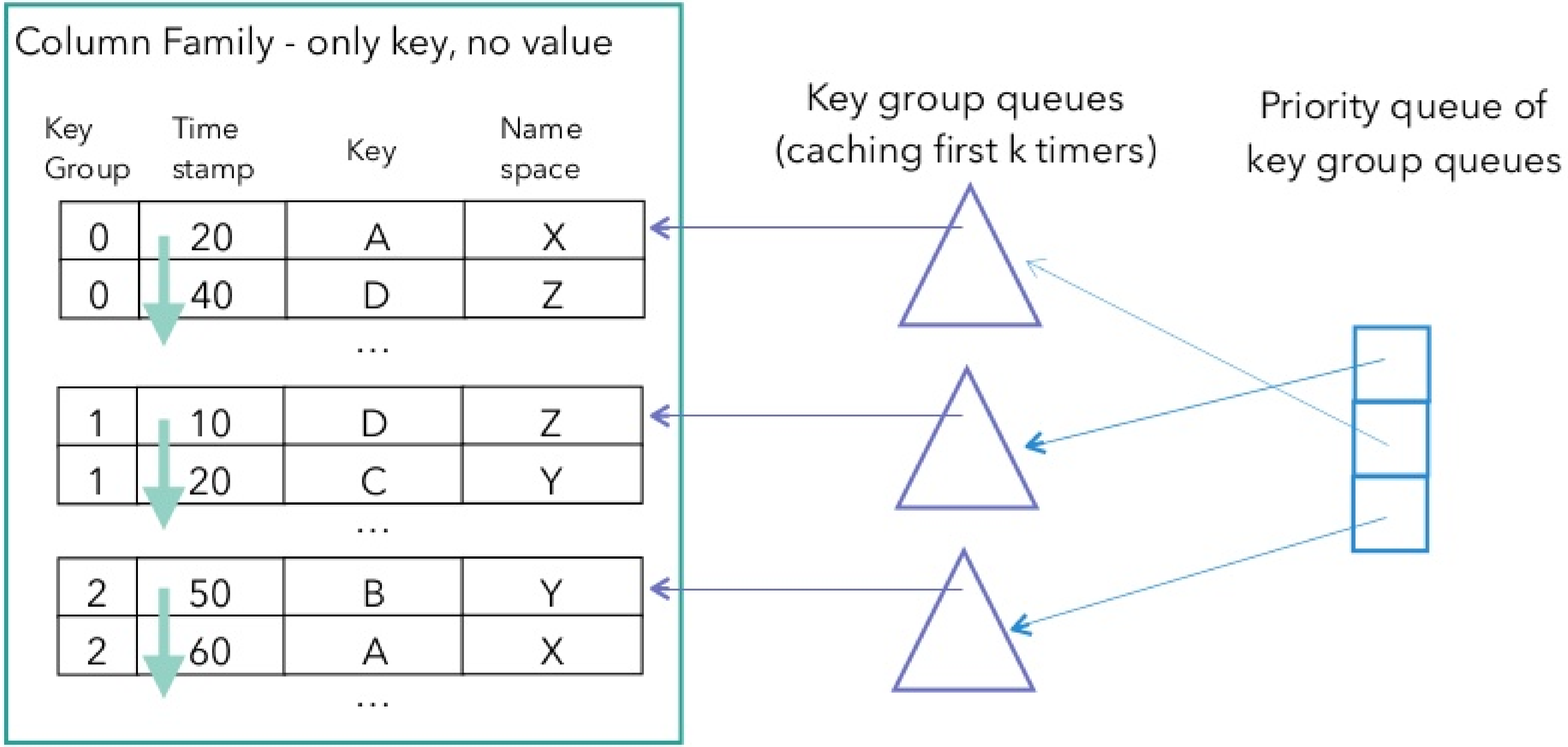
...

2	50	B	Y
2	60	A	X

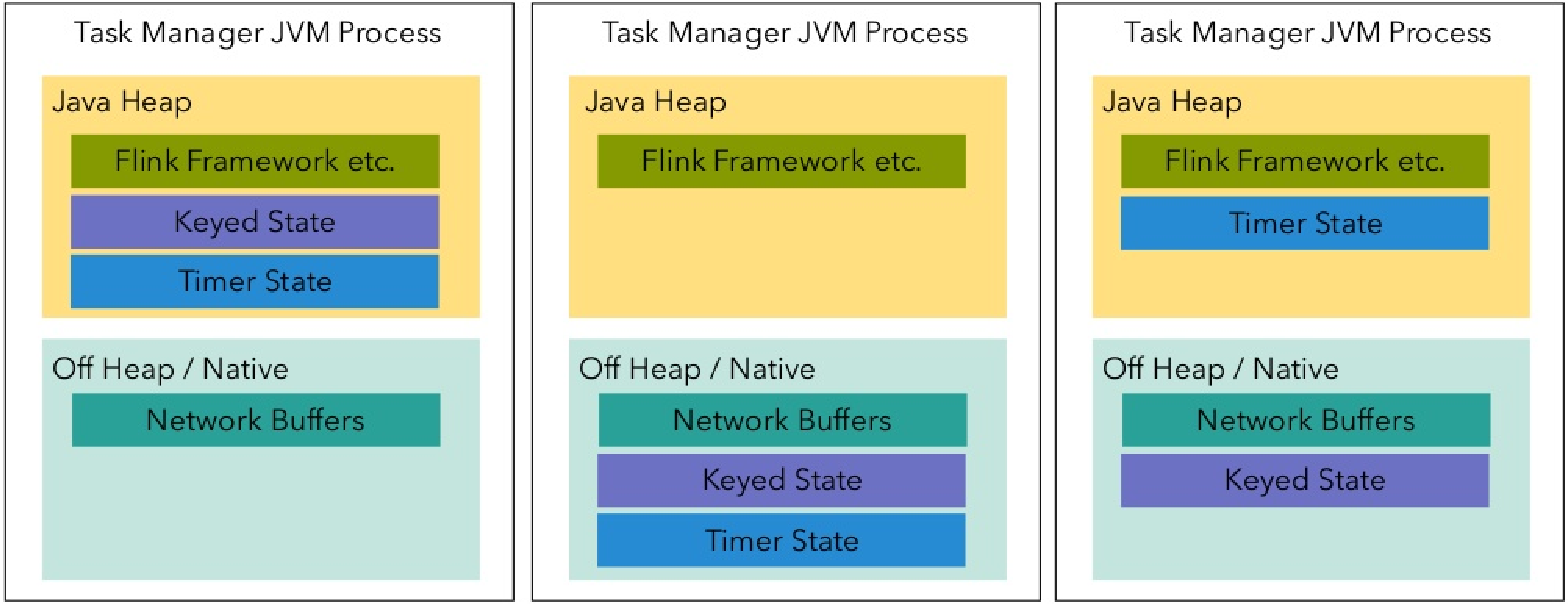
...

Lexicographically ordered
byte sequences as key, no value

ROCKSDB TIMERS



3 TASK MANAGER MEMORY LAYOUT OPTIONS

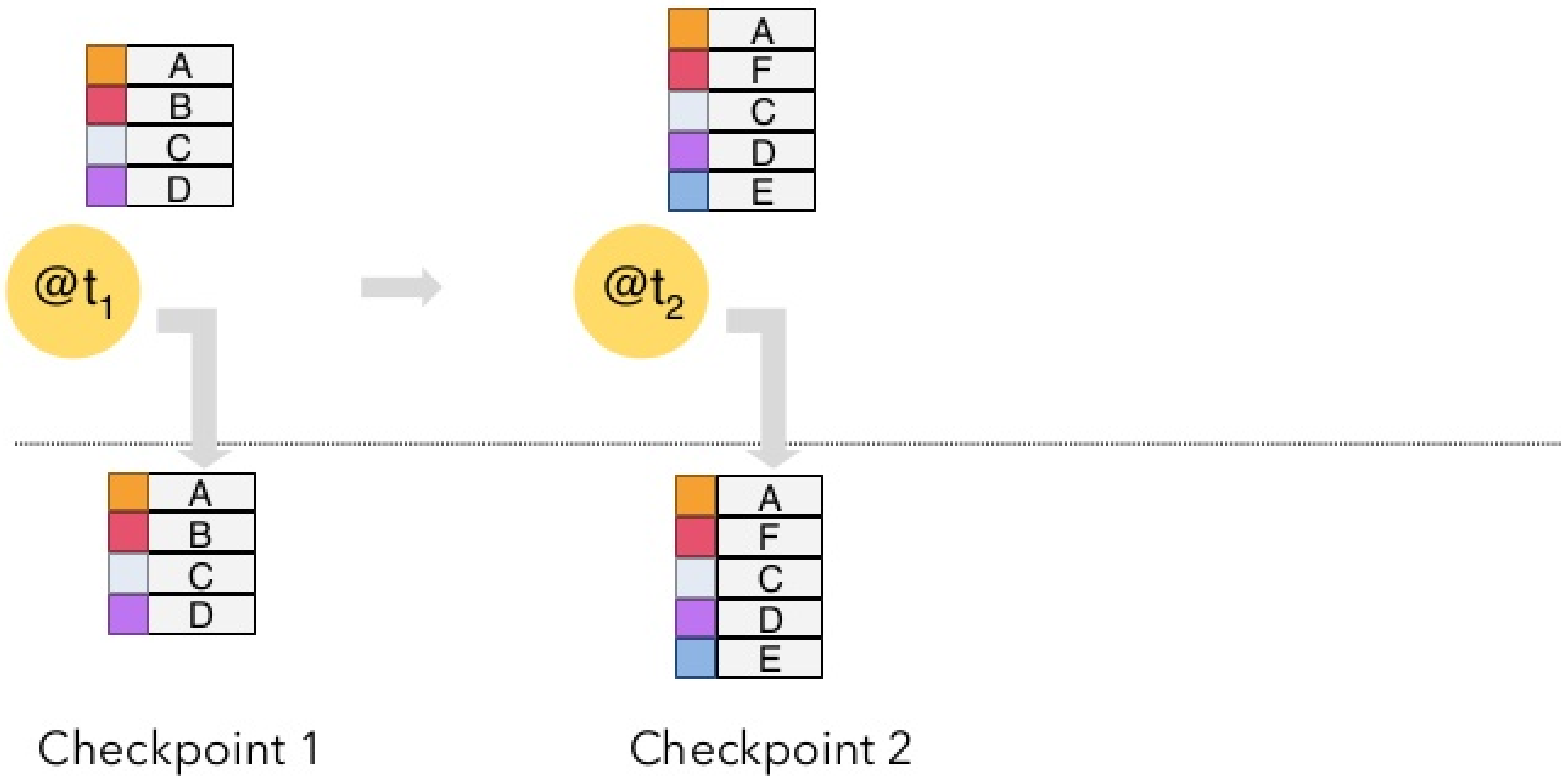


FULL / INCREMENTAL CHECKPOINTS

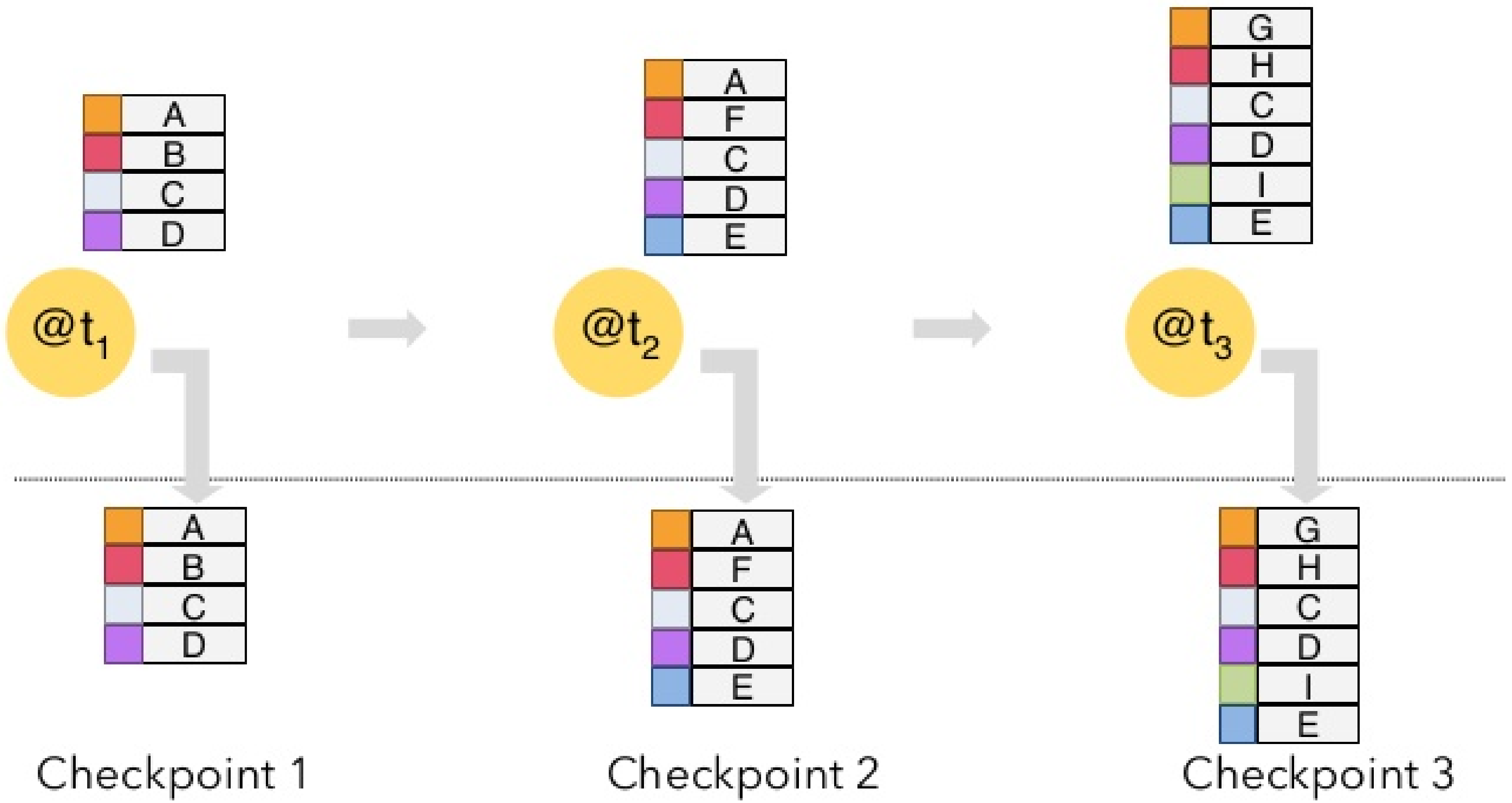
FULL CHECKPOINT

Checkpoint 1

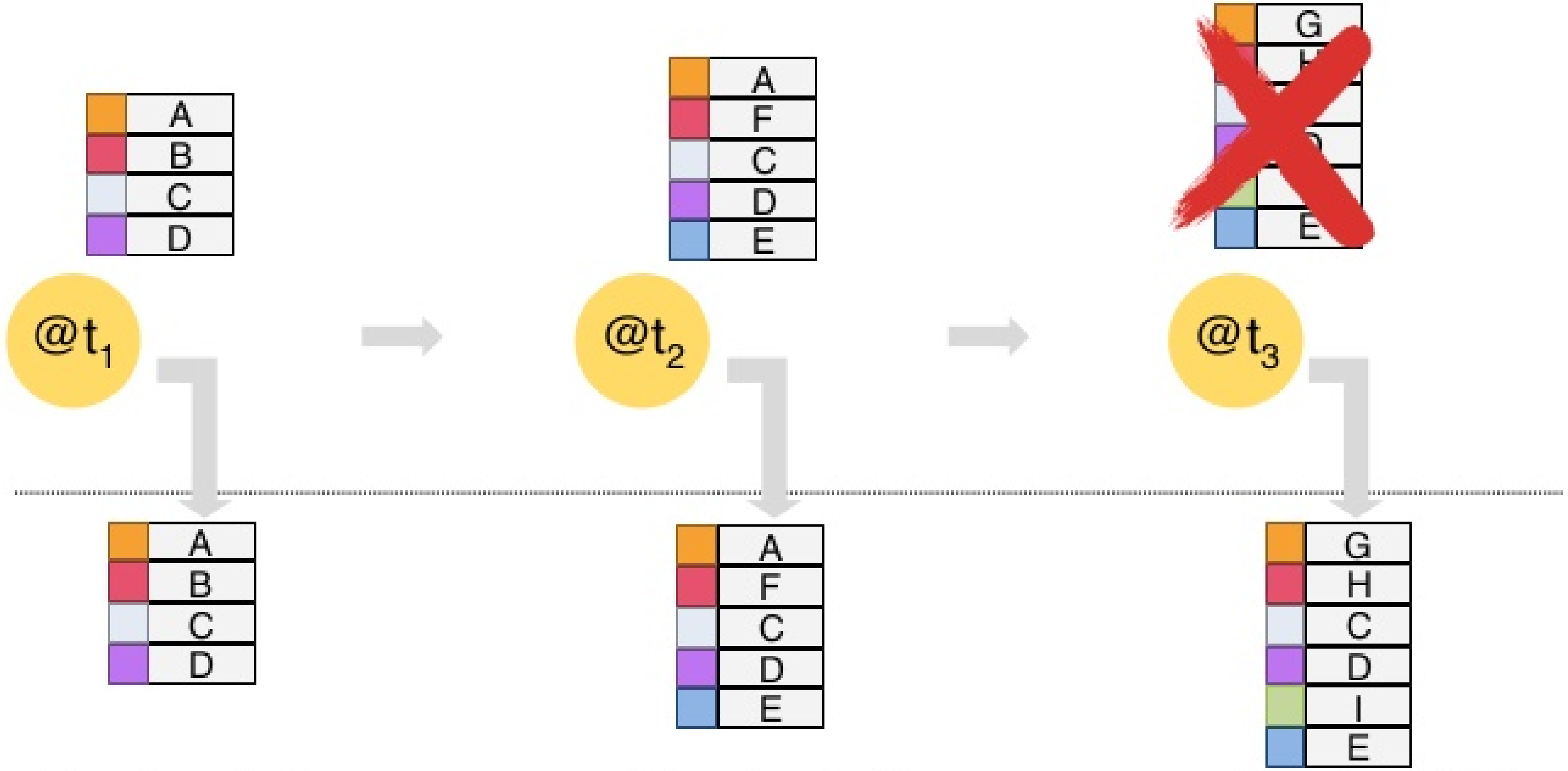
FULL CHECKPOINT



FULL CHECKPOINT



FULL CHECKPOINT

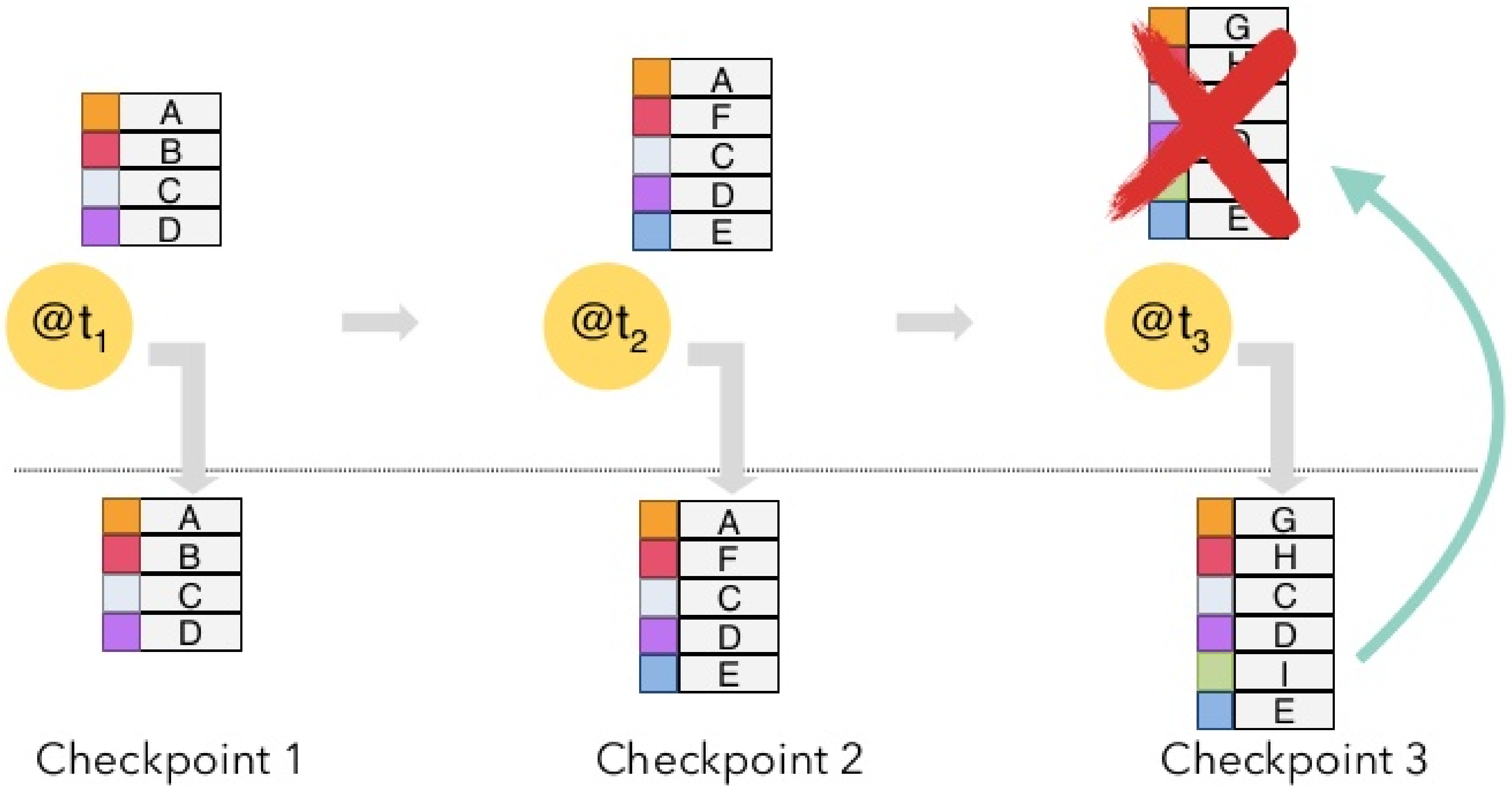


Checkpoint 1

Checkpoint 2

Checkpoint 3

FULL CHECKPOINT

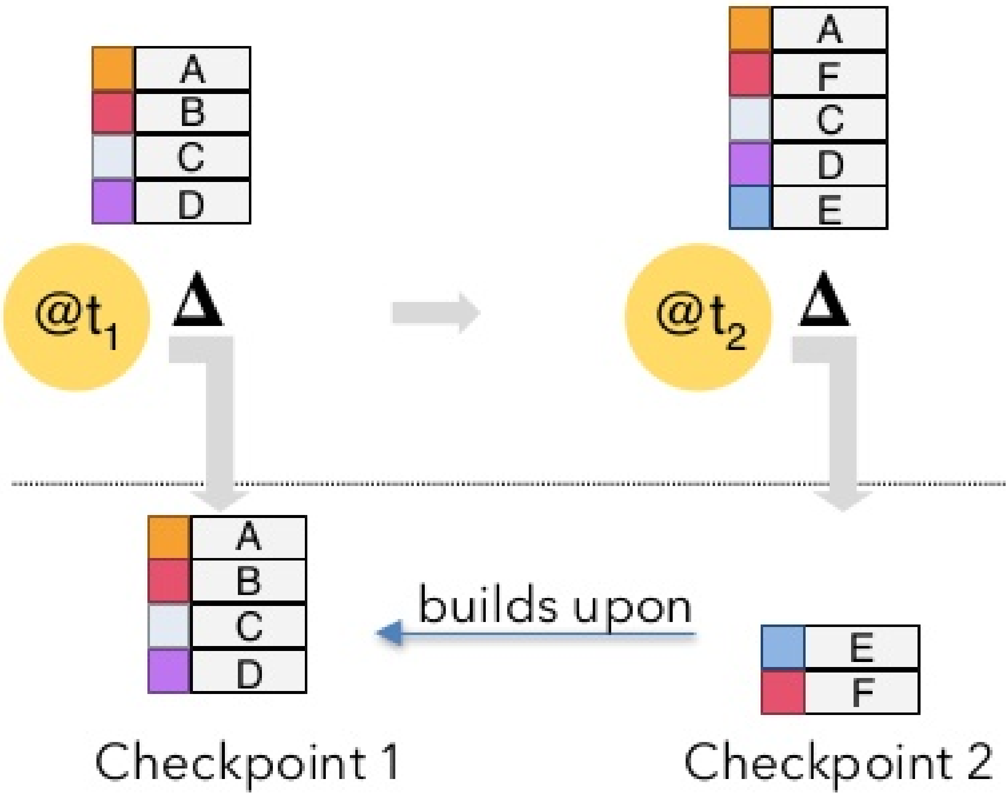


FULL CHECKPOINT OVERVIEW

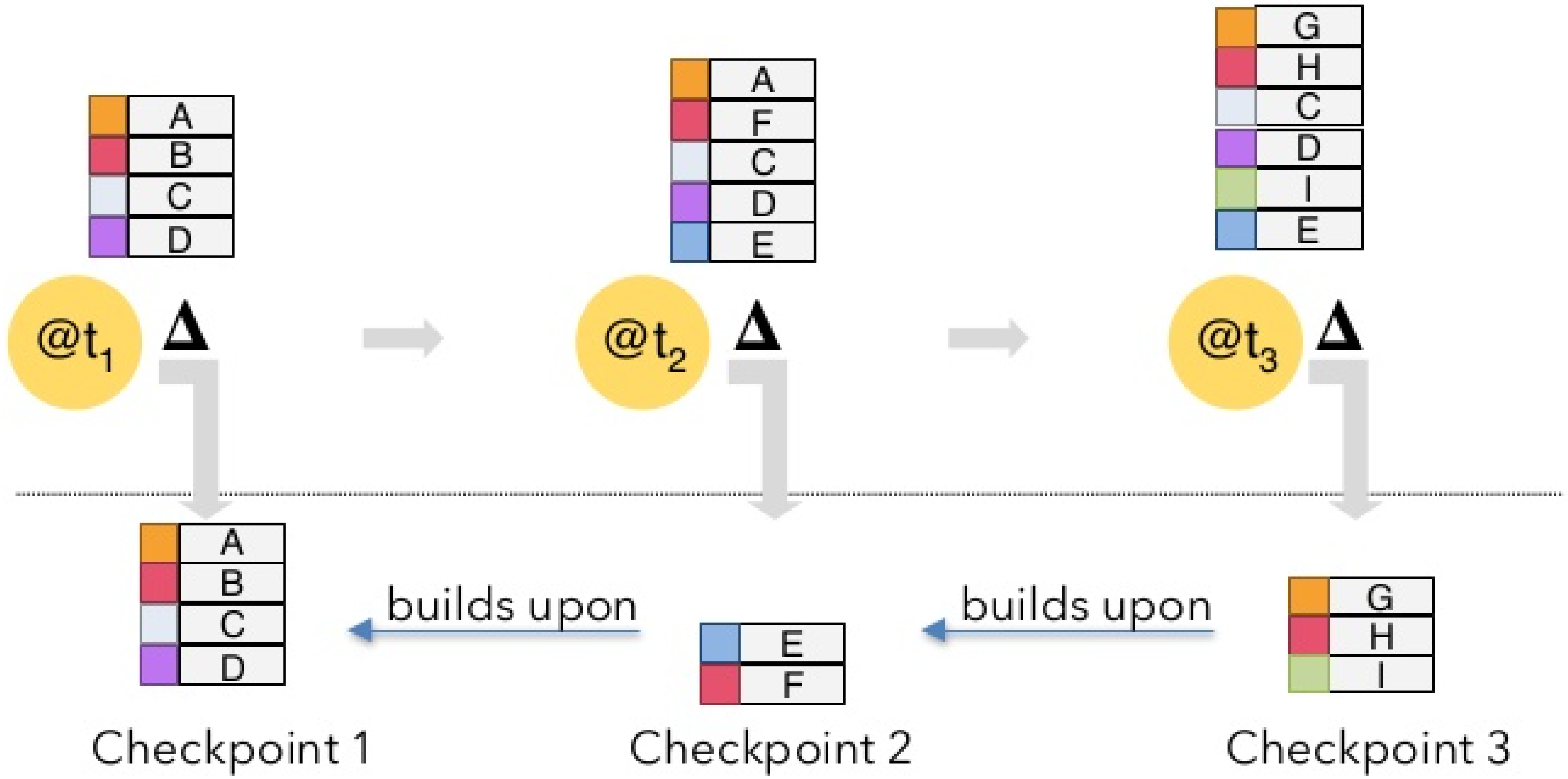
- Creation iterates and writes full database snapshot as stream to stable storage.
- Restore reads data as stream from stable storage and re-inserts into backend.
- Each checkpoint is self contained, size is proportional to size of full state.
- Optional: compression with Snappy.

INCREMENTAL CHECKPOINT

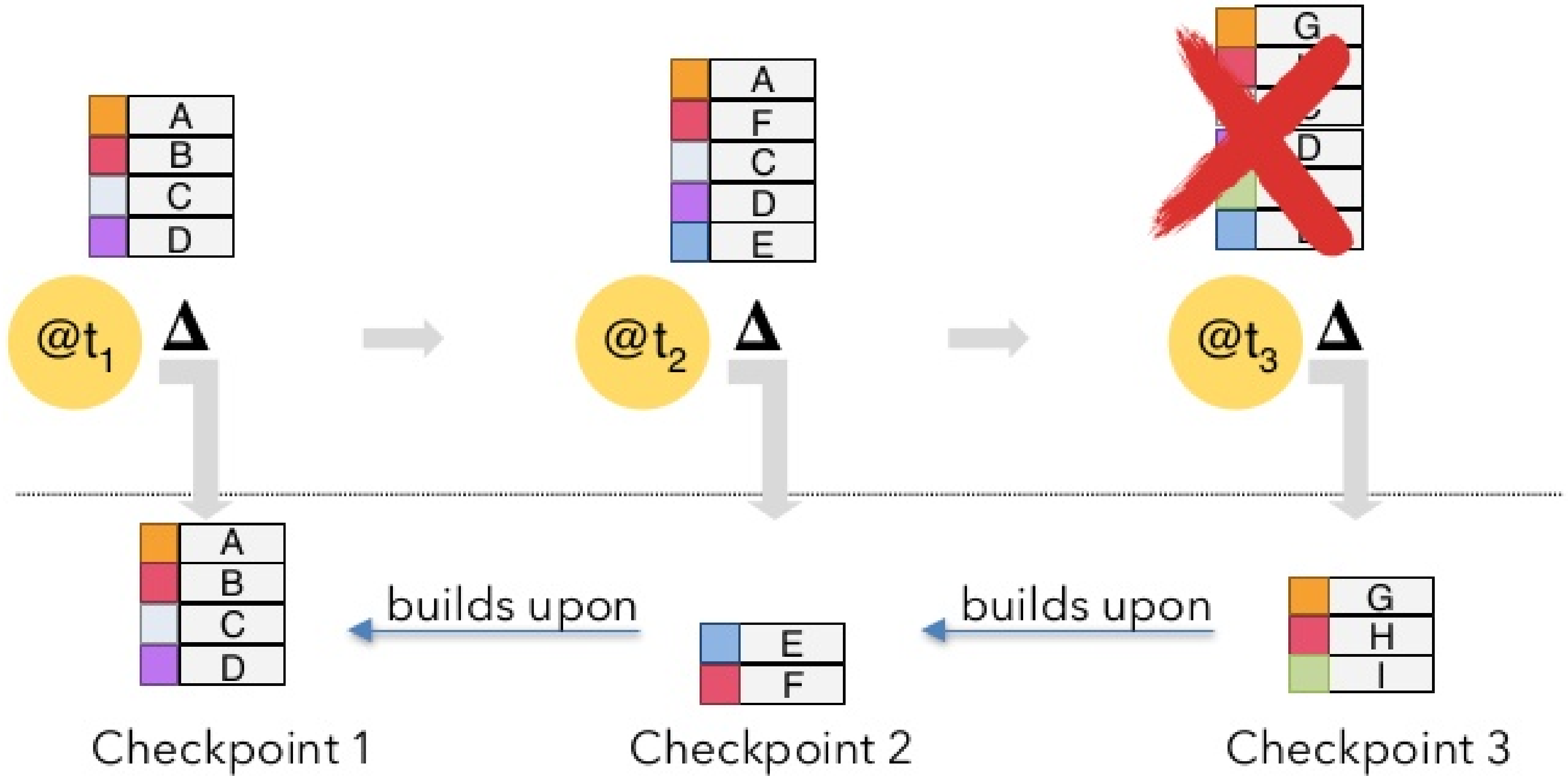
INCREMENTAL CHECKPOINT



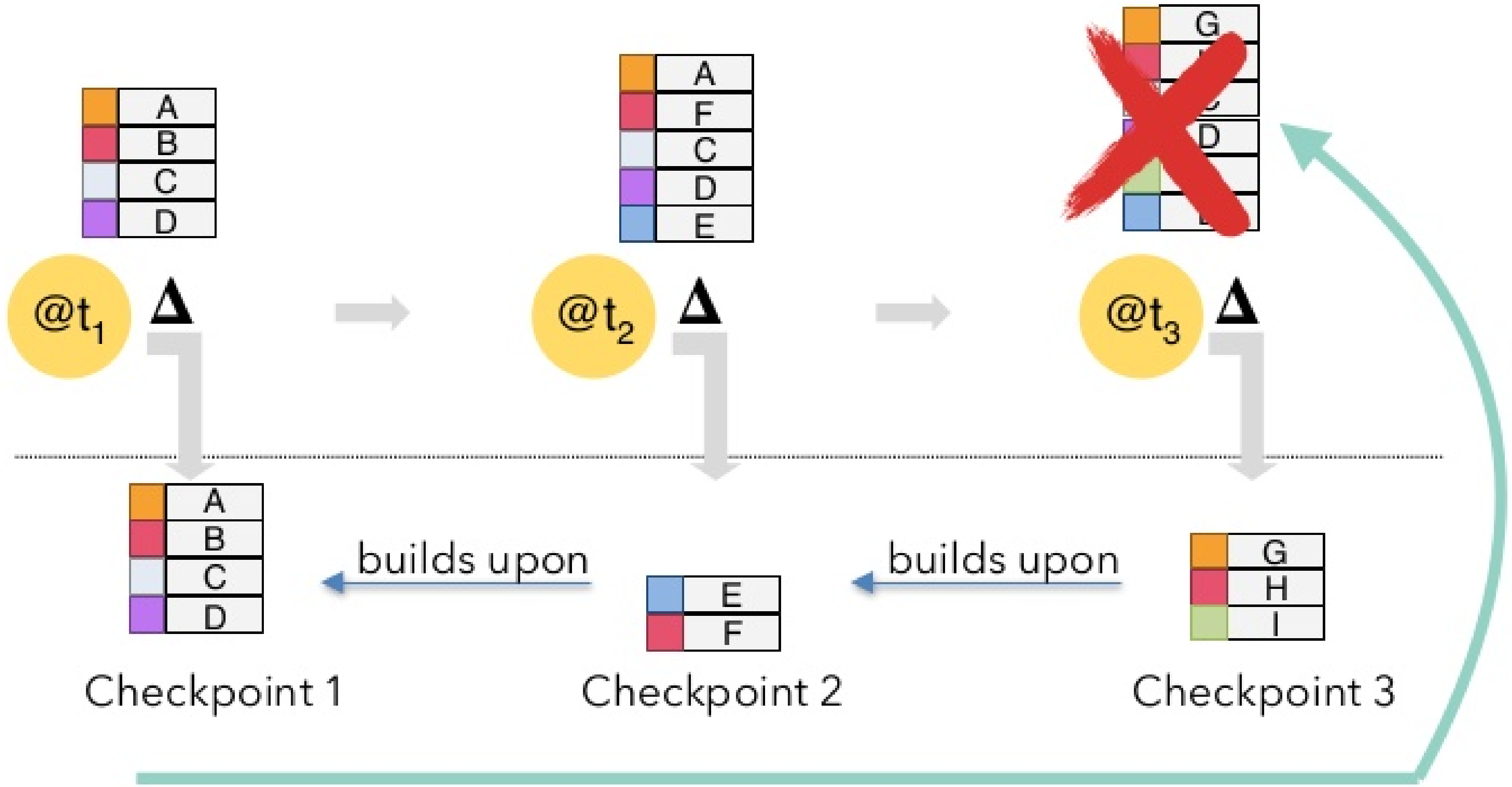
INCREMENTAL CHECKPOINT



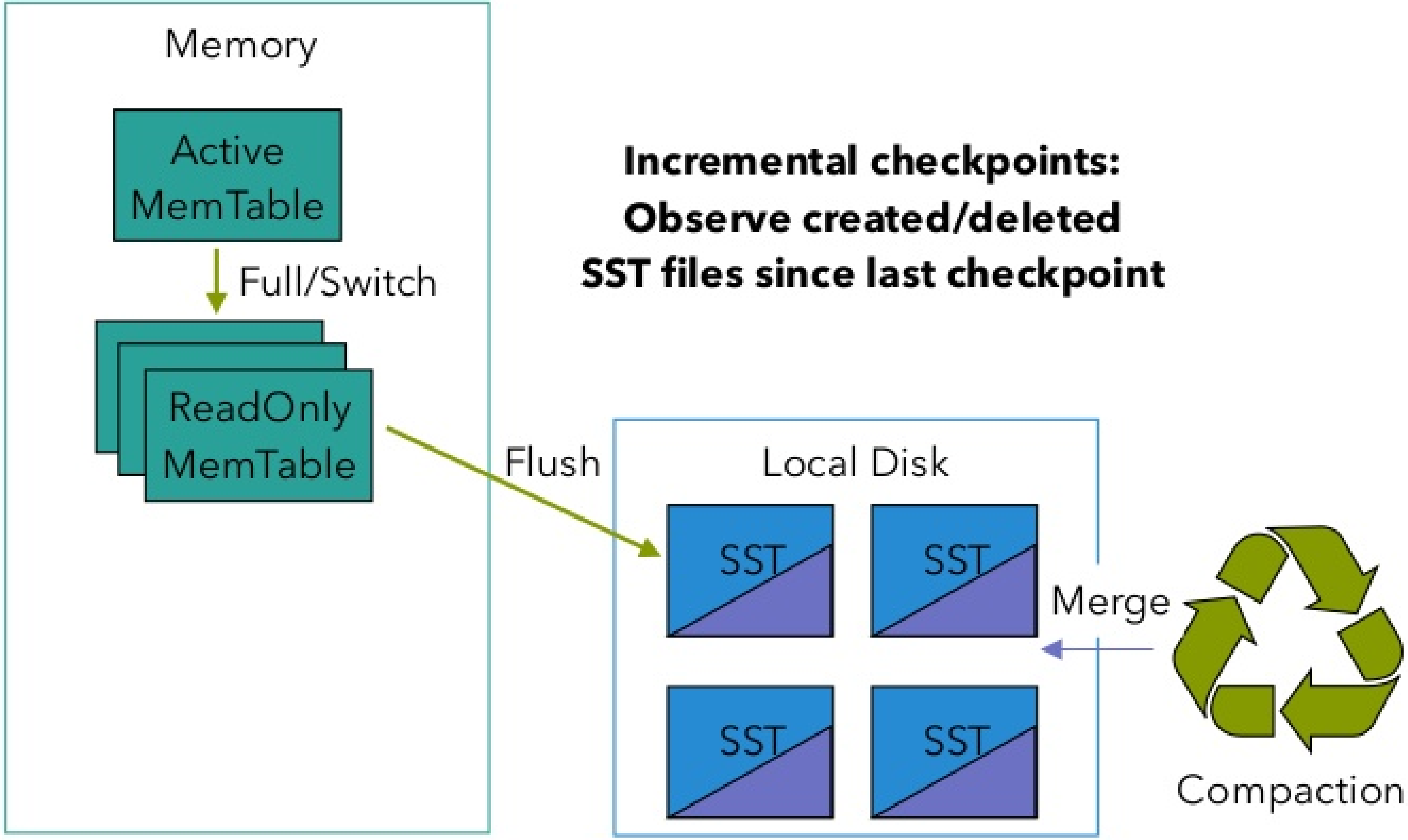
INCREMENTAL CHECKPOINT



INCREMENTAL CHECKPOINT



INCREMENTAL CHECKPOINTS WITH ROCKSDB



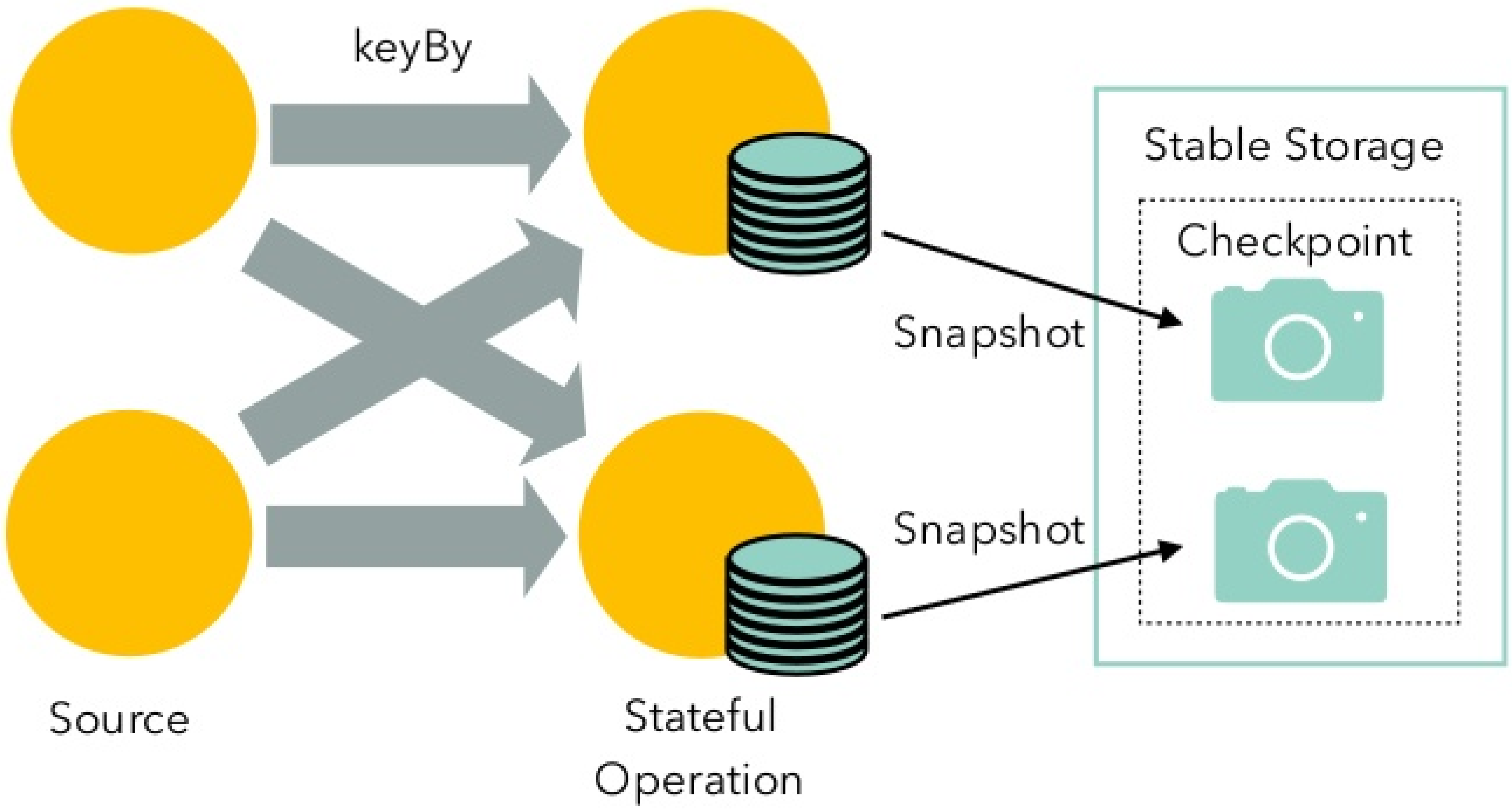
Incremental checkpoints:
Observe created/deleted
SST files since last checkpoint

INCREMENTAL CHECKPOINT OVERVIEW

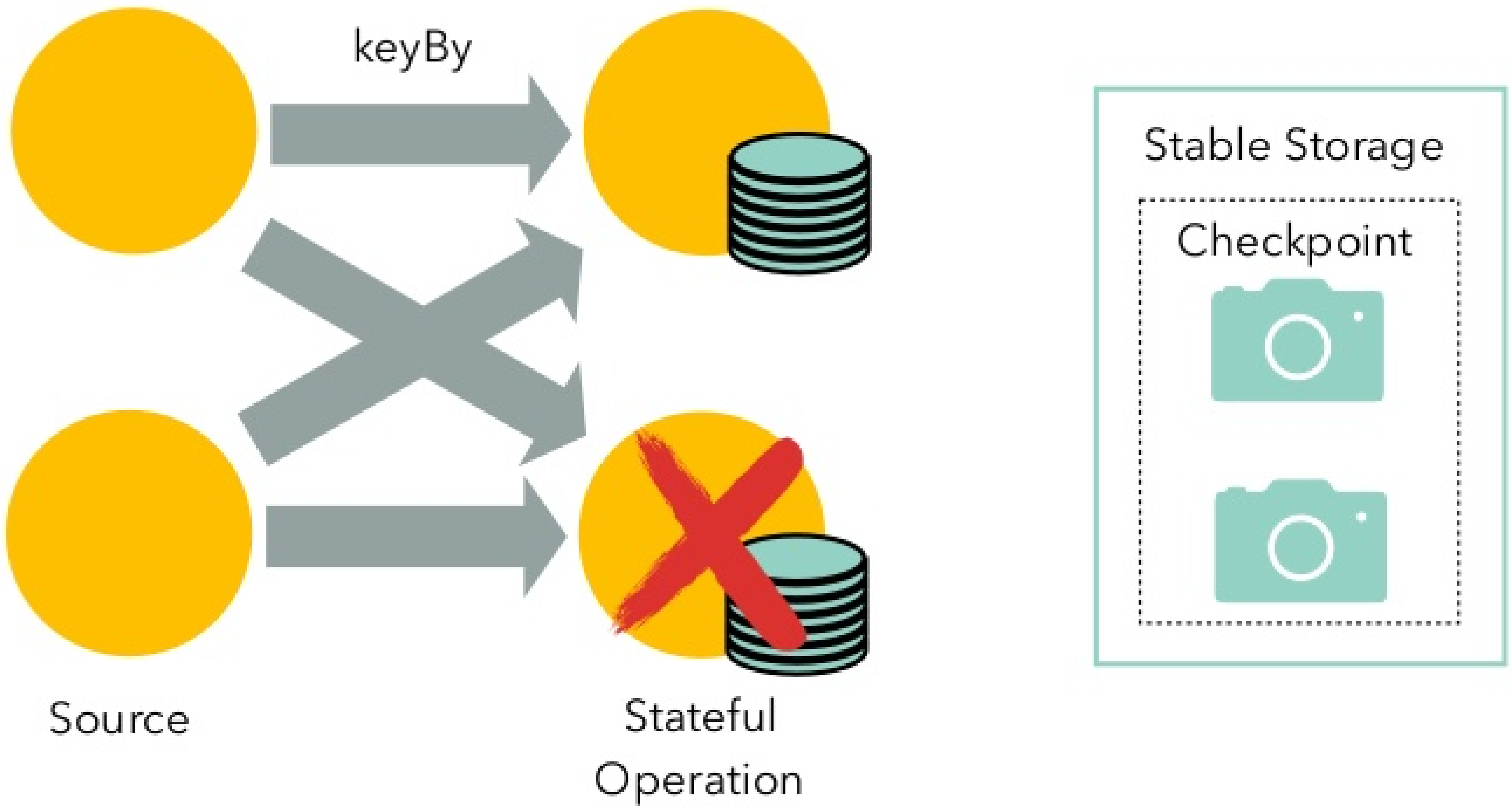
- Expected trade-off: faster* checkpoints, slower* recovery
- Creation only copies deltas (new local SST files) to stable storage.
- Write amplification because we also upload compacted SST files so that we can prune checkpoint history.
- Sum of all increments that we read from stable storage can be larger than the full state size. Deletes are also explicit as tombstones.
- But no rebuild required because we simply re-open the RocksDB backend from the SST files.
- SST files are snappy-compressed by default.

LOCAL RECOVERY

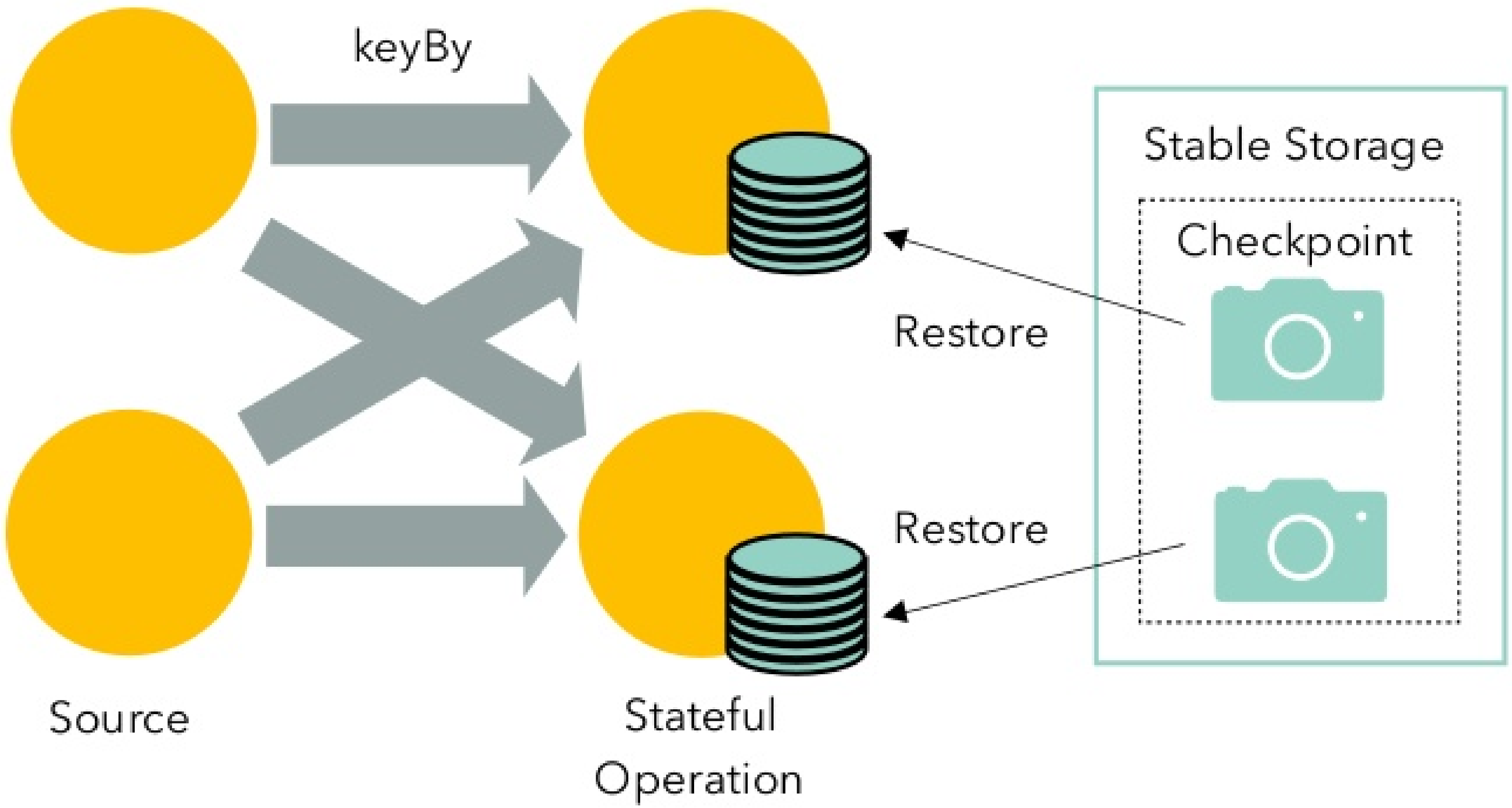
CHECKPOINTING WITHOUT LOCAL RECOVERY



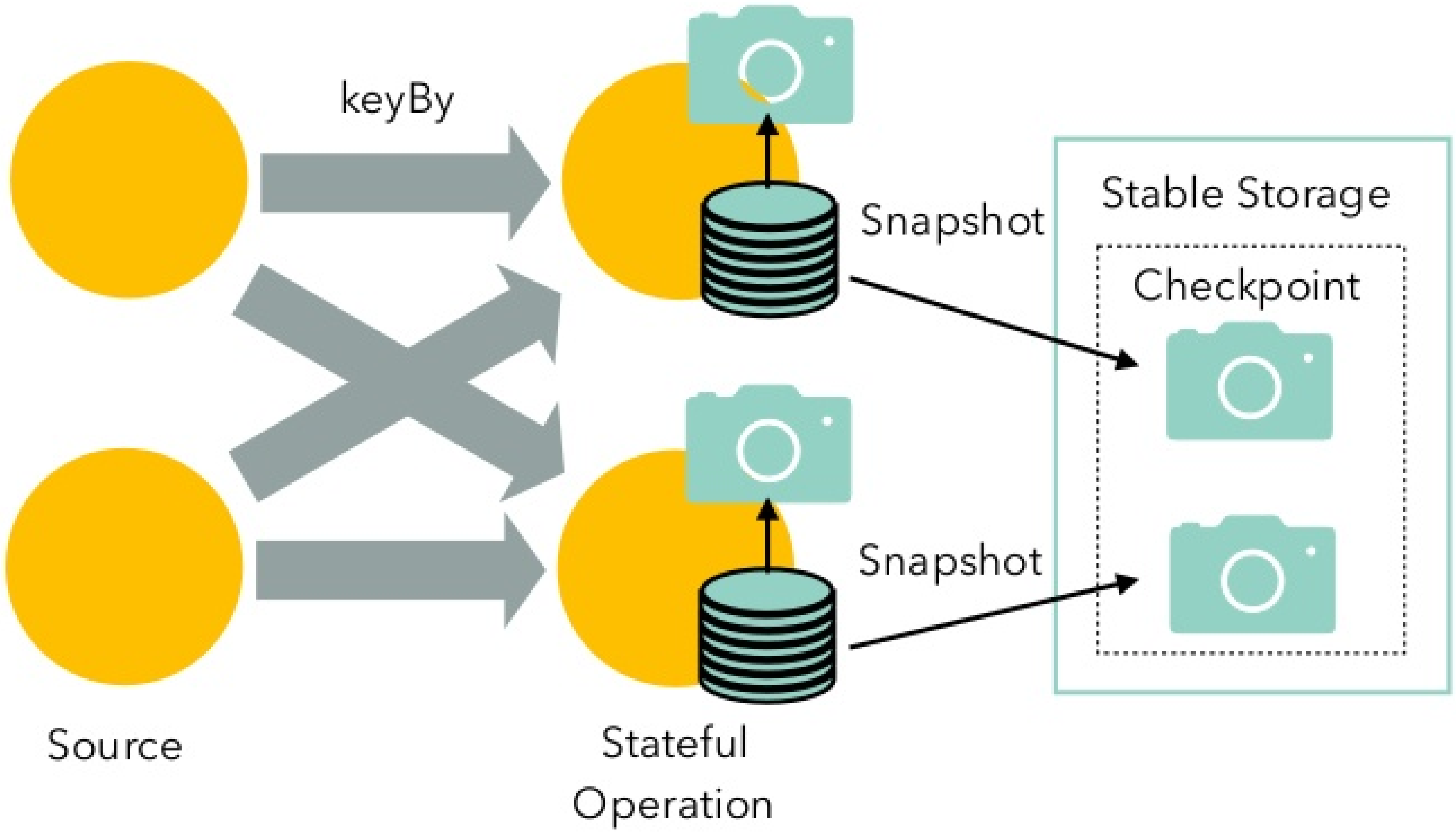
RESTORE WITHOUT LOCAL RECOVERY



RESTORE WITHOUT LOCAL RECOVERY

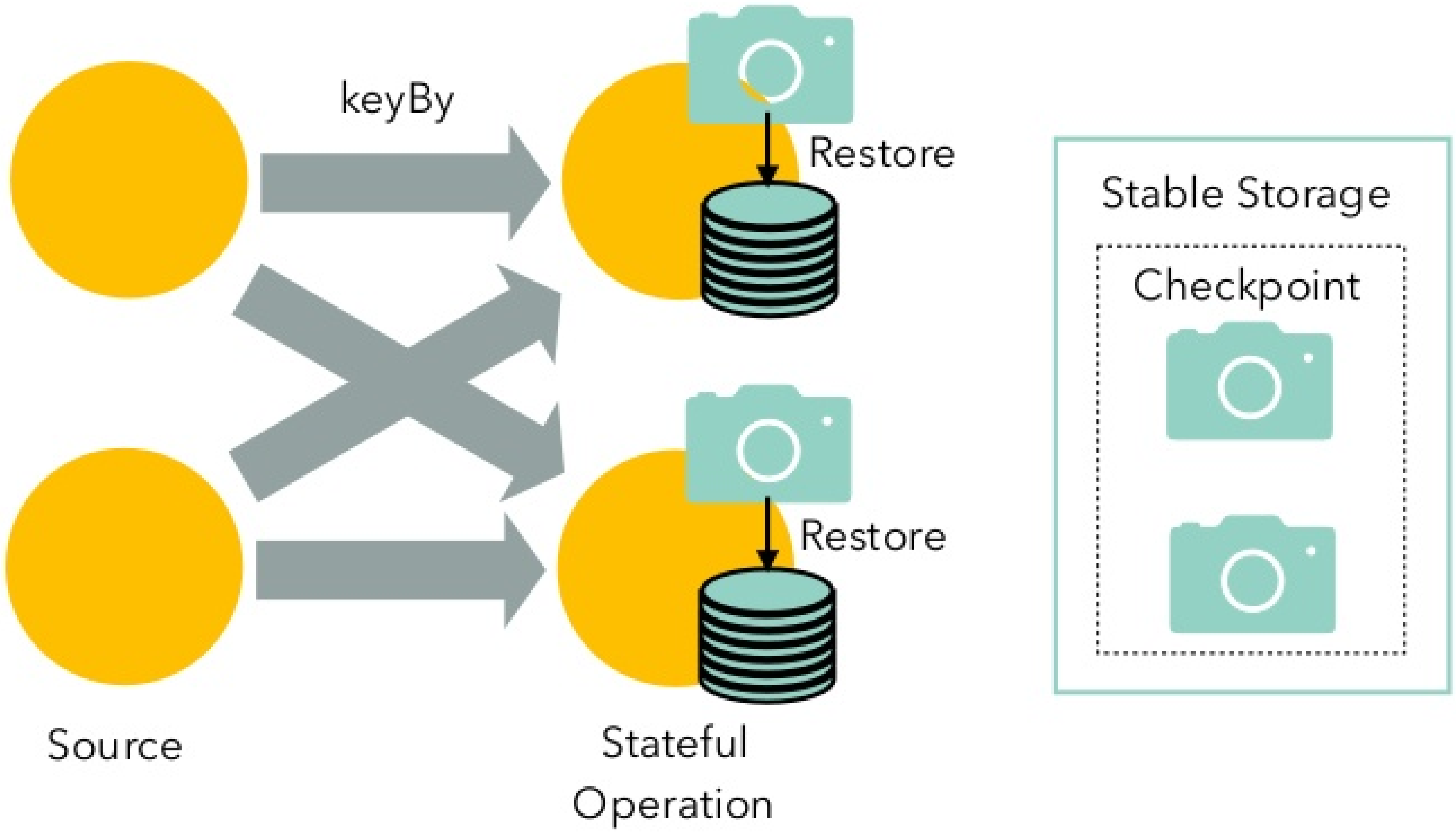


CHECKPOINTING WITH LOCAL RECOVERY



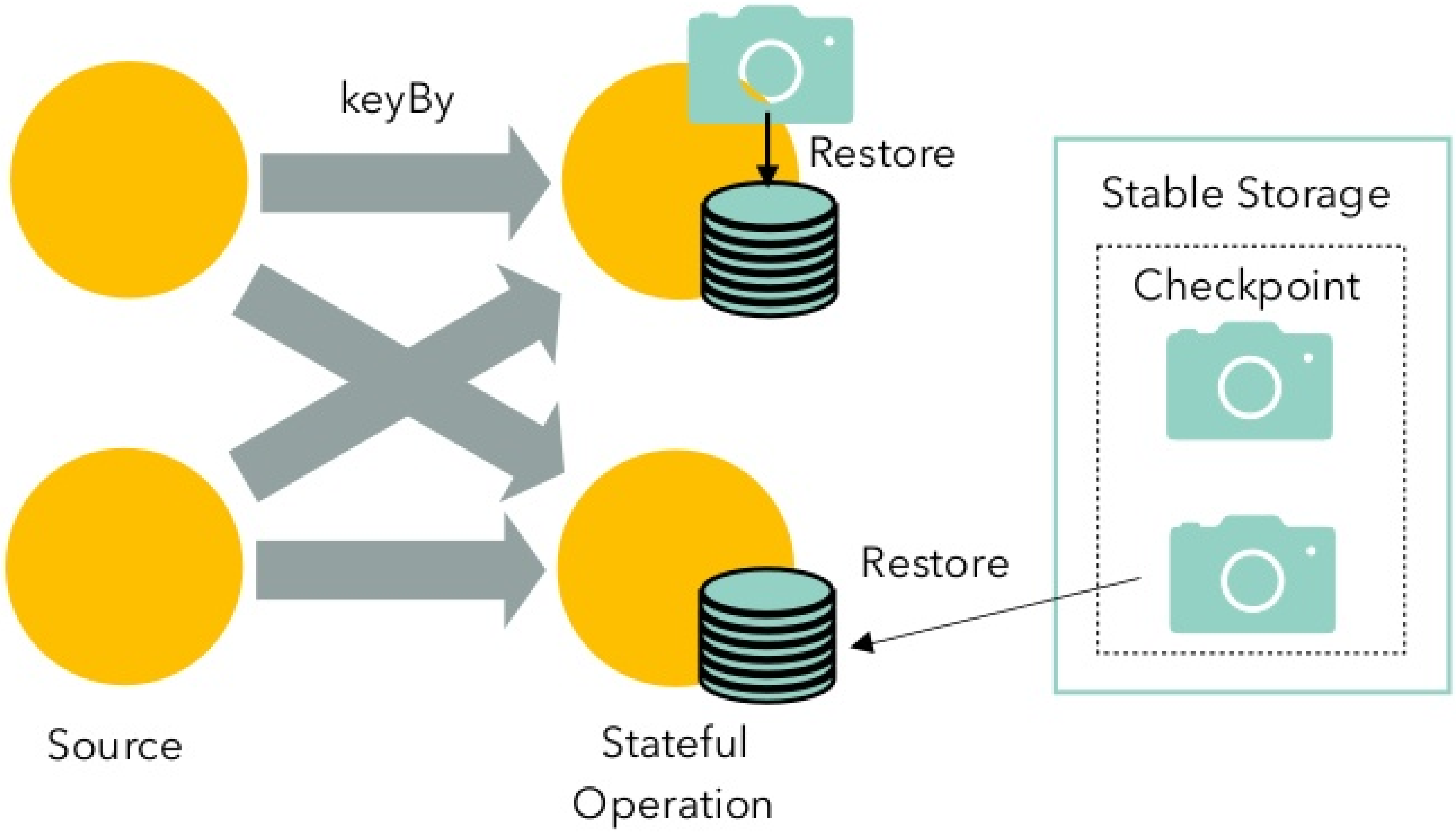
RESTORE WITH LOCAL RECOVERY

Scenario 1: No task manager failures, e.g. user code exception



RESTORE WITH LOCAL RECOVERY

Scenario 2: With task manager failure, e.g. disk failure



LOCAL RECOVERY TAKEAWAY POINTS

- Works with both state backends, for full and incremental checkpoints.
  - Keeps a local copy of the snapshot. Typically, this comes at the cost of mirroring the snapshot writes to remote storage also to local storage.
  - Restore with LR avoids the transfer of state from stable to local storage.

- LR works particularly well with RocksDB incremental checkpoints.
  - No new local files created, existing files might only live a bit longer.
  - Opening database from local, native table files - no ingestion / rebuild.

- Under TM failure recovery still bounded by slowest restore, but still saves a lot of resources!

REINTERPRET STREAM AS KEYED STREAM

REINTERPRETING A DATASTREAM AS KEYED

```
env.addSource(new InfiniteTupleSource(1000))
        .keyBy(0)
        .map((in) -> in)
        .timeWindow(Time.seconds(3));
```

Problem: Will not compile because we can no longer ensure a keyed stream!

REINTERPRETING A DATASTREAM AS KEYED

```
env.addSource(new InfiniteTupleSource(1000))
        .keyBy(0)
        .map((in) -> in)
        .timeWindow(Time.seconds(3));
```

Problem: Will not compile because we can no longer ensure a keyed stream!

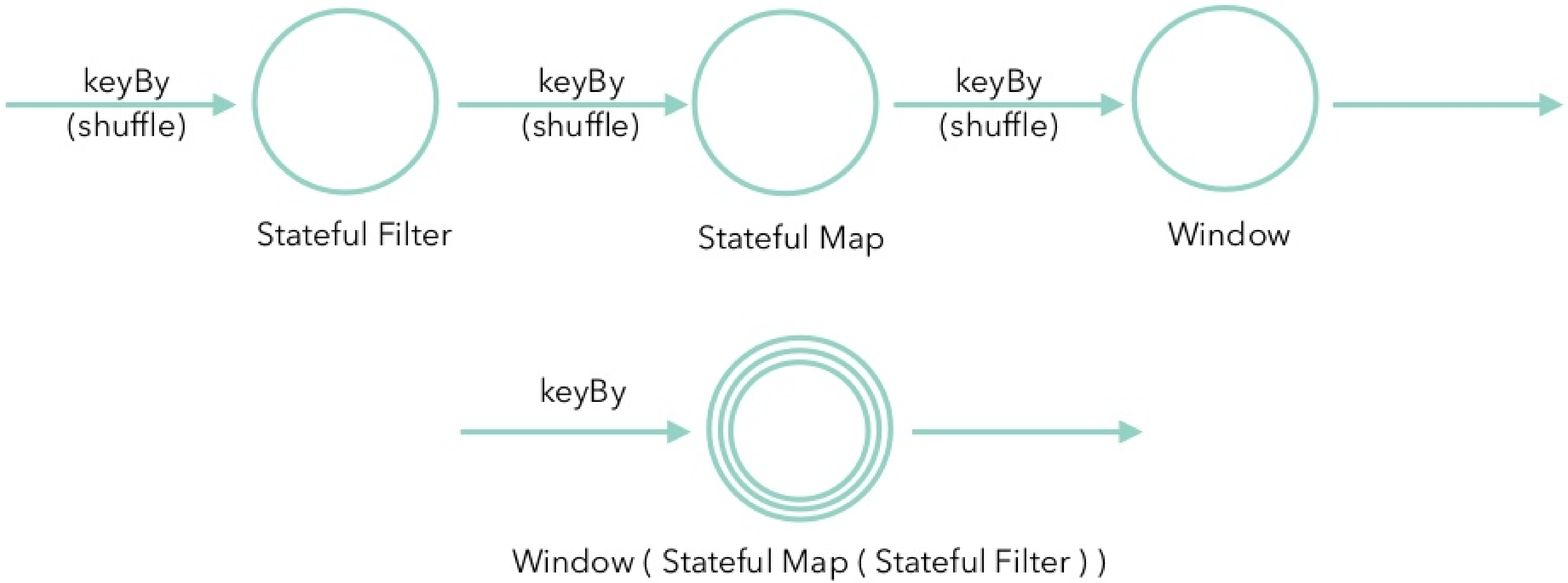
```
KeyedStream<T, K> reinterpretAsKeyedStream(
    DataStream<T> stream,
    KeySelector<T, K> keySelector)
```

Solution: Method to explicitly give (back) „keyed" property to any data stream

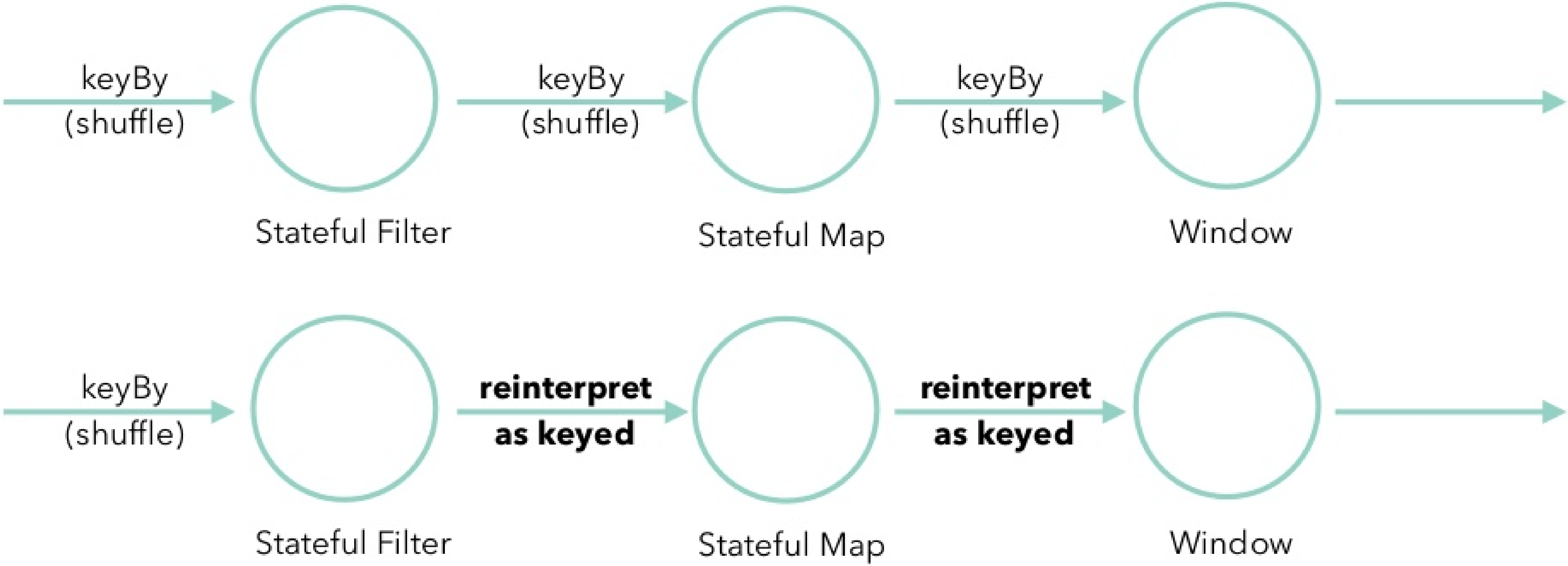
```
DataStreamUtils.reinterpretAsKeyedStream(
    env.addSource(new InfiniteTupleSource(1000))
        .keyBy(0)
        .filter((in) -> true), (in) -> in.f0)
    .timeWindow(Time.seconds(3));
```

Warning: Only use this when you are absolutely sure that the elements in the reinterpreted stream follow exactly Flink's keyBy partitioning scheme for the given key selector!

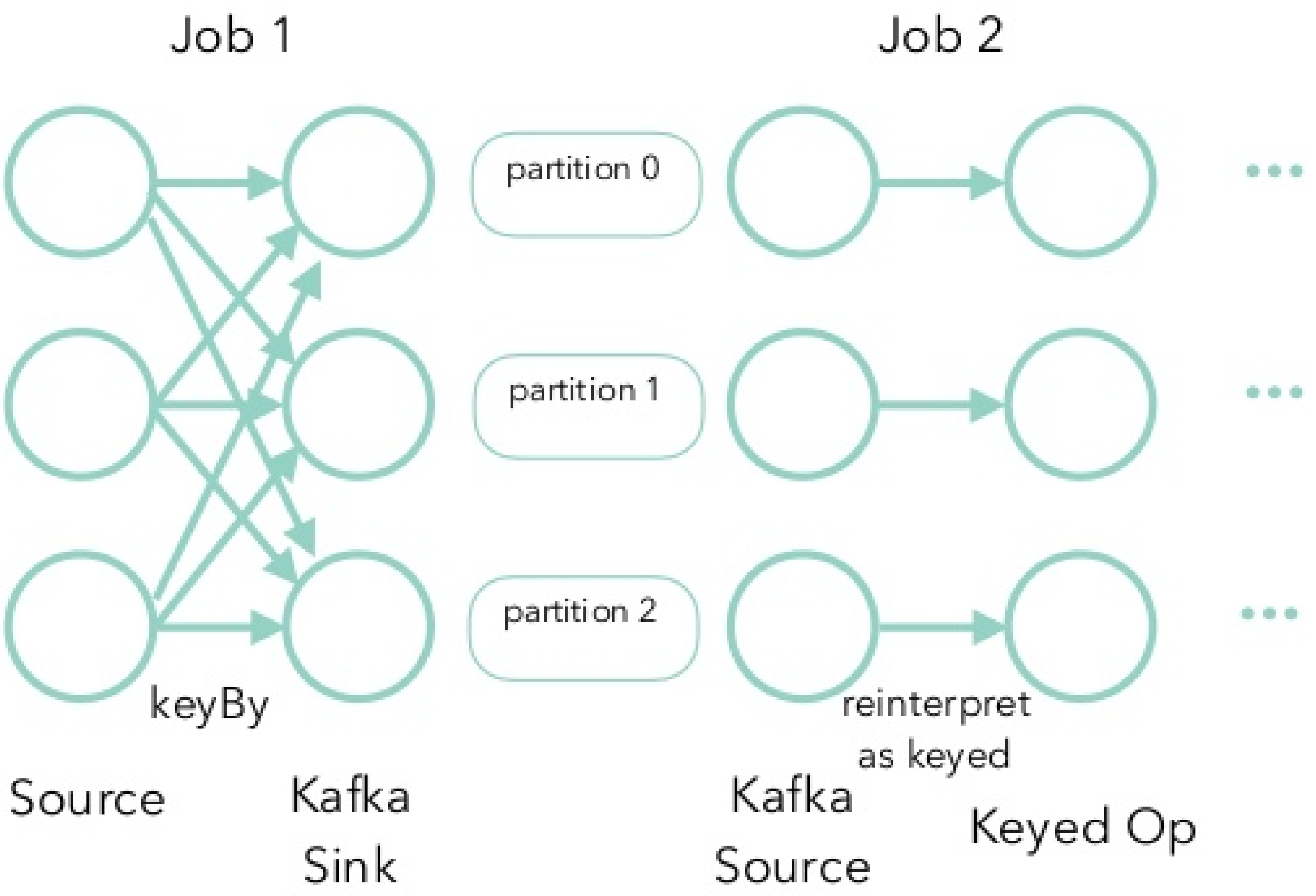
IDEA 1 - REDUCING SHUFFLES



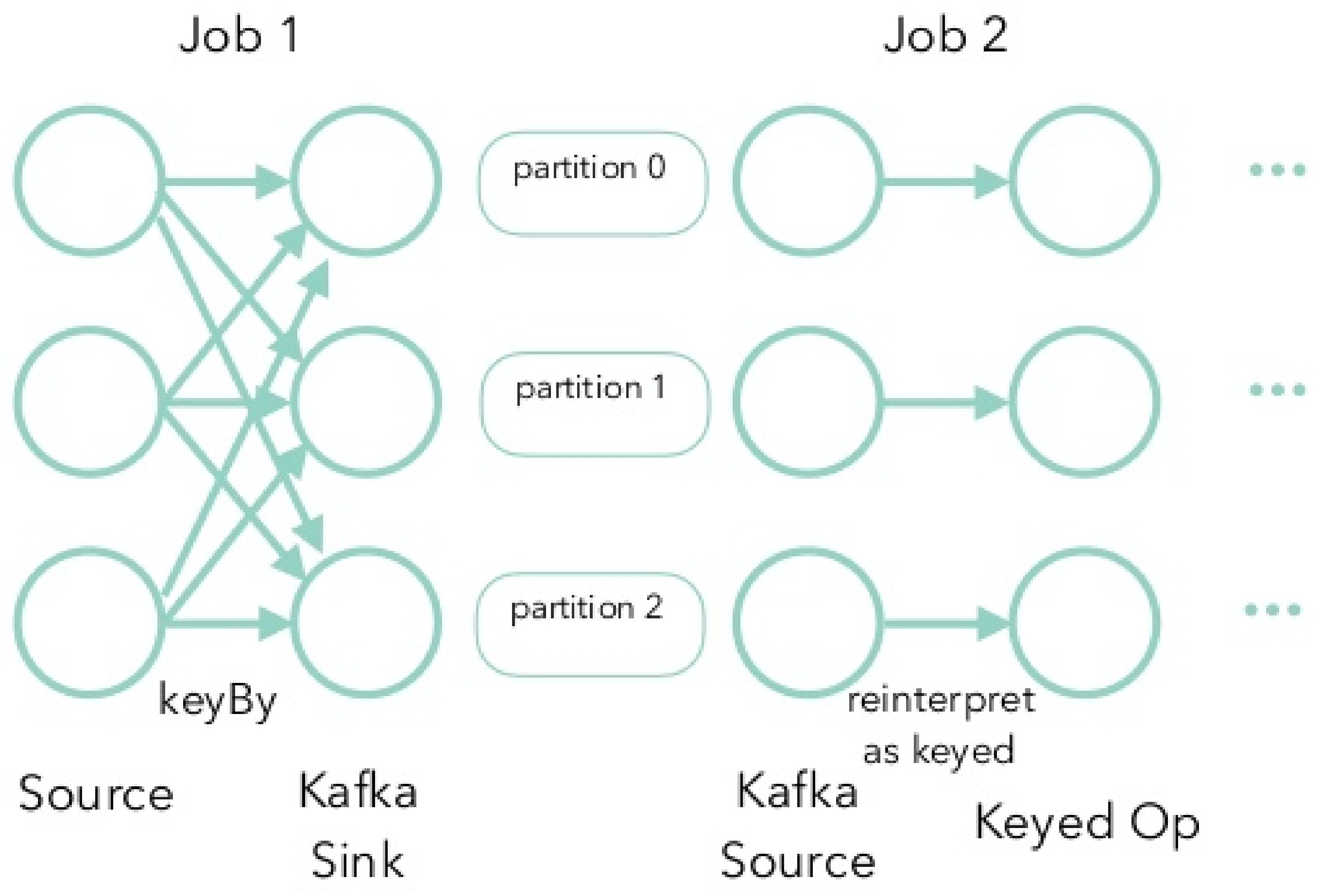
IDEA 1 - REDUCING SHUFFLES



IDEA 2 - PERSISTENT SHUFFLE



IDEA 2 - PERSISTENT SHUFFLE



Job 2 becomes embarrassingly parallel and can use fine grained recovery!

THANK YOU!

@StefanRRichter
@dataArtisans
@ApacheFlink

WE ARE HIRING

data-artisans.com/careers

dataArtisans