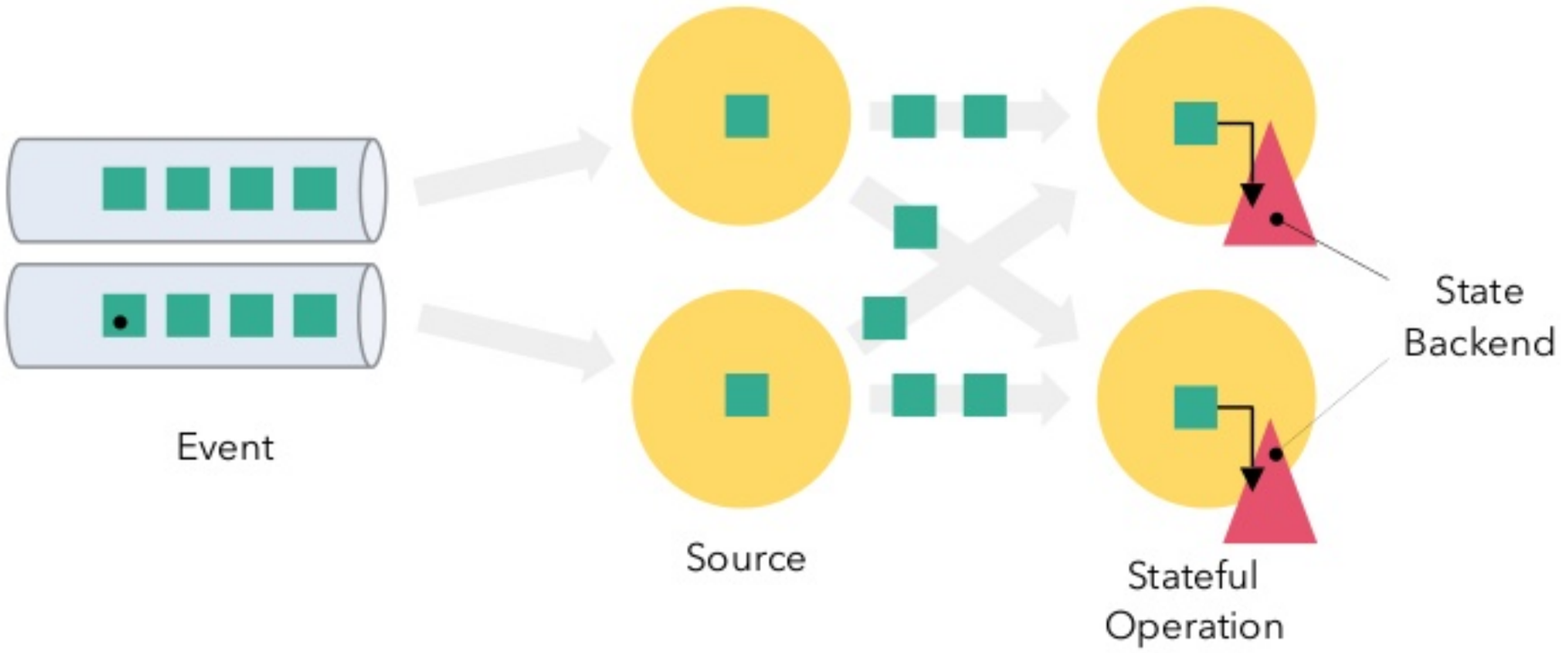
A Look at Flink's Internal Data Structures for Efficient Checkpointing

Stefan Richter
@StefanRRichter

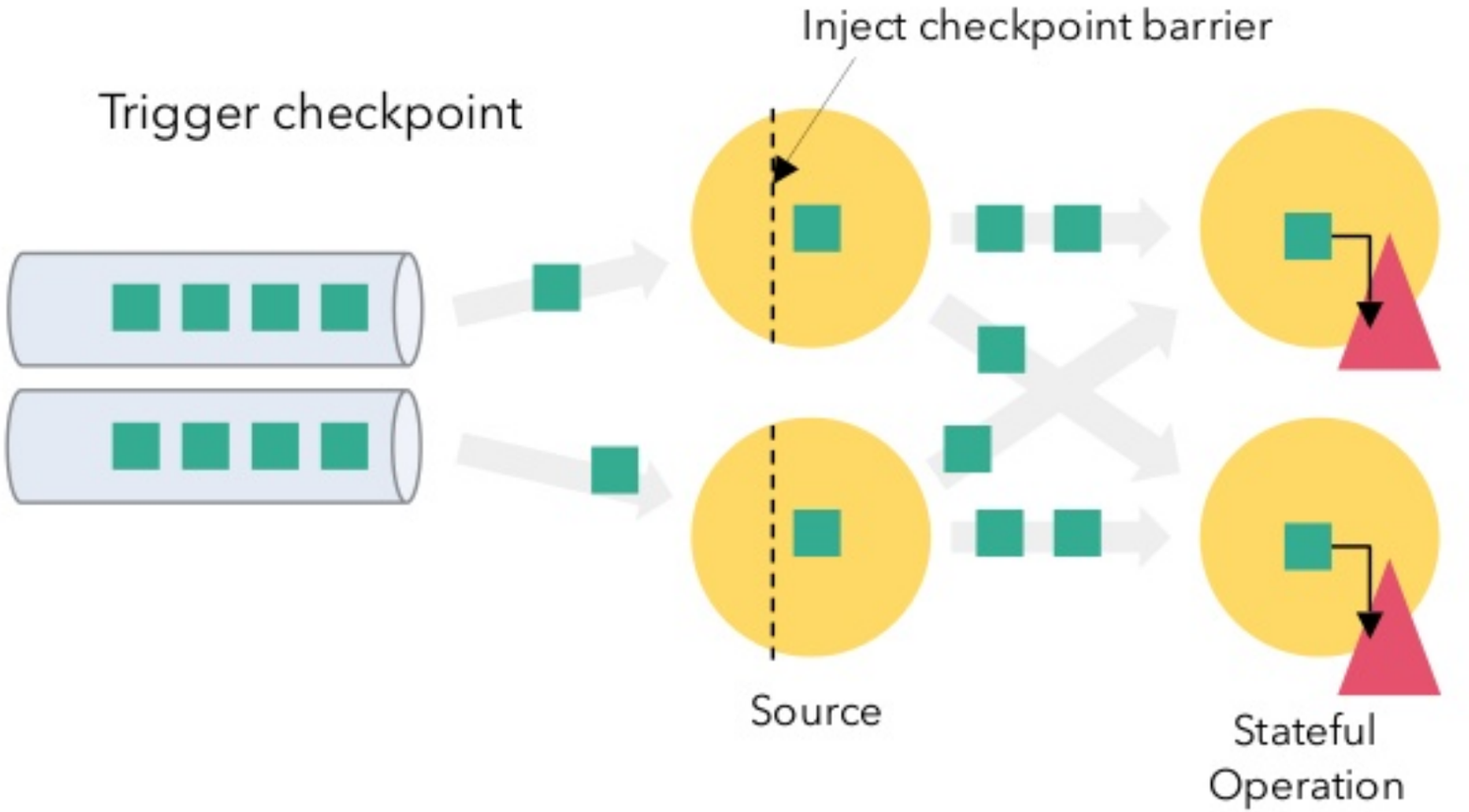
dataArtisans

September 13, 2017

Flink State and Distributed Snapshots

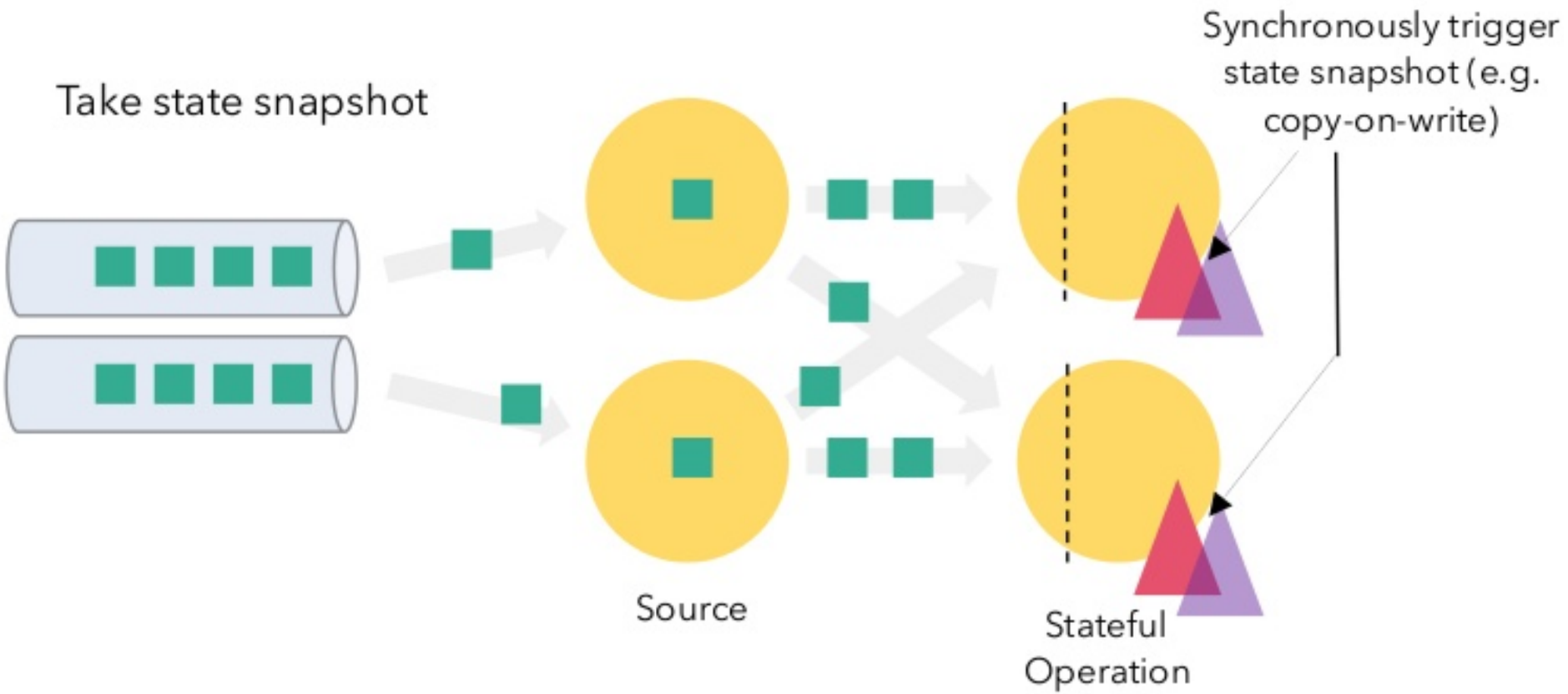


Flink State and Distributed Snapshots



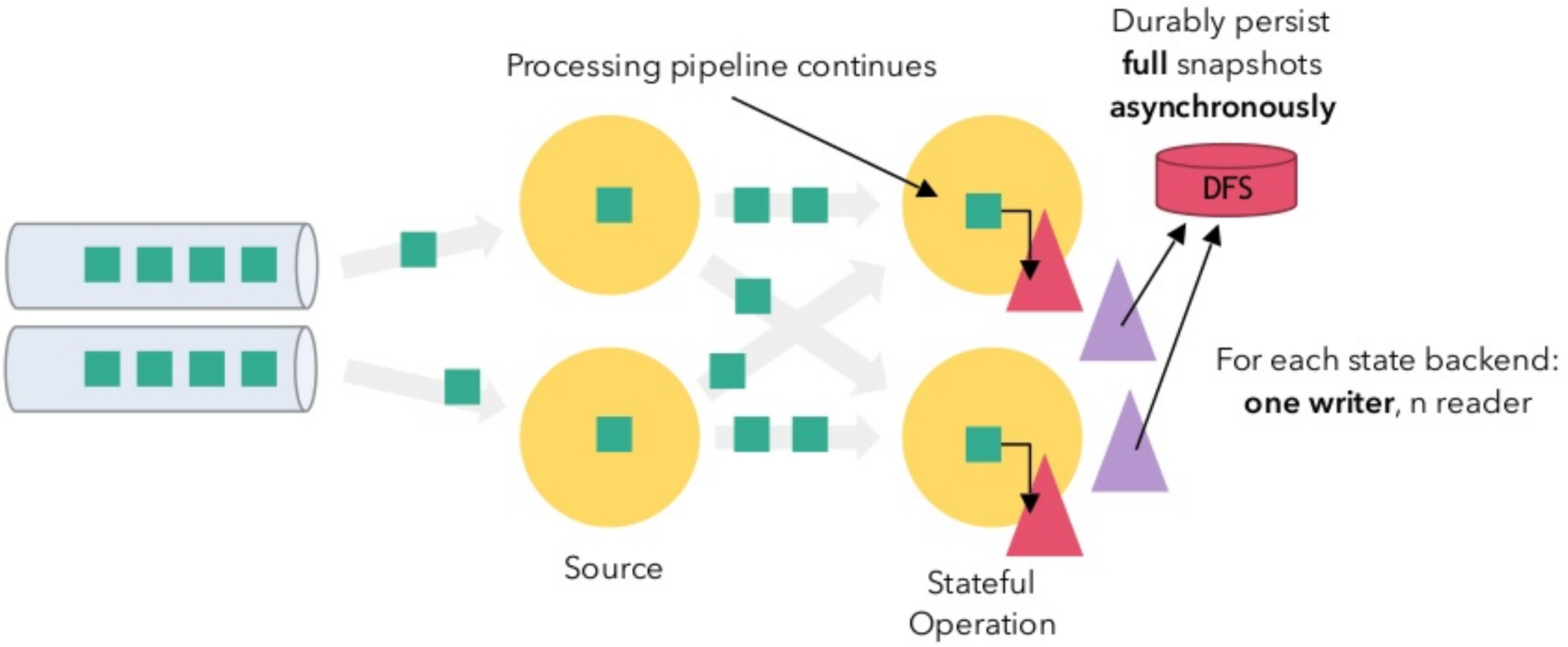
„Asynchronous Barrier Snapshotting“

Flink State and Distributed Snapshots



„Asynchronous Barrier Snapshotting“

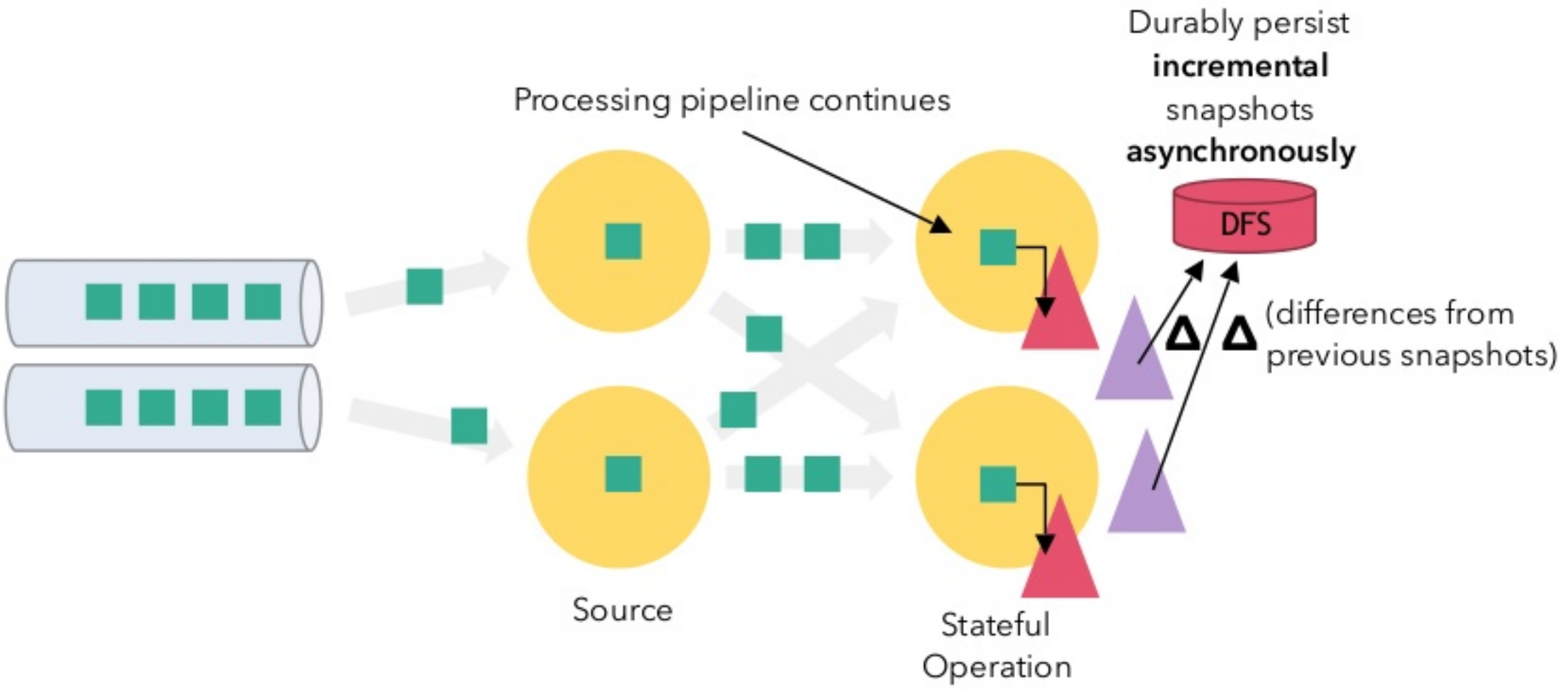
Flink State and Distributed Snapshots



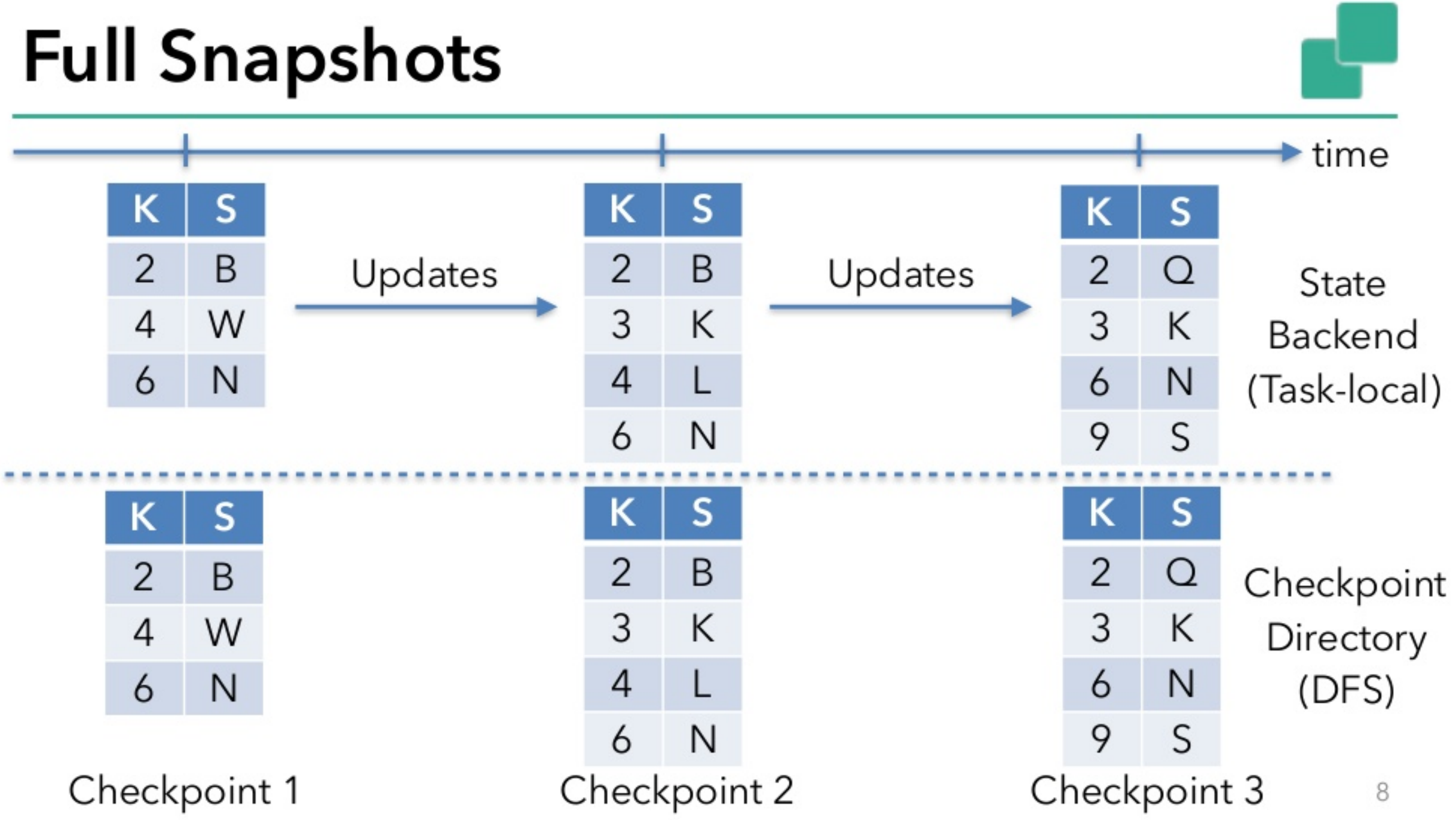
Challenges for Data Structures

- Asynchronous Checkpoints:
  - Minimize pipeline stall time while taking the snapshot.
  - Keep overhead (memory, CPU,…) as low as possible while writing the snapshot.
  - Support multiple parallel checkpoints.

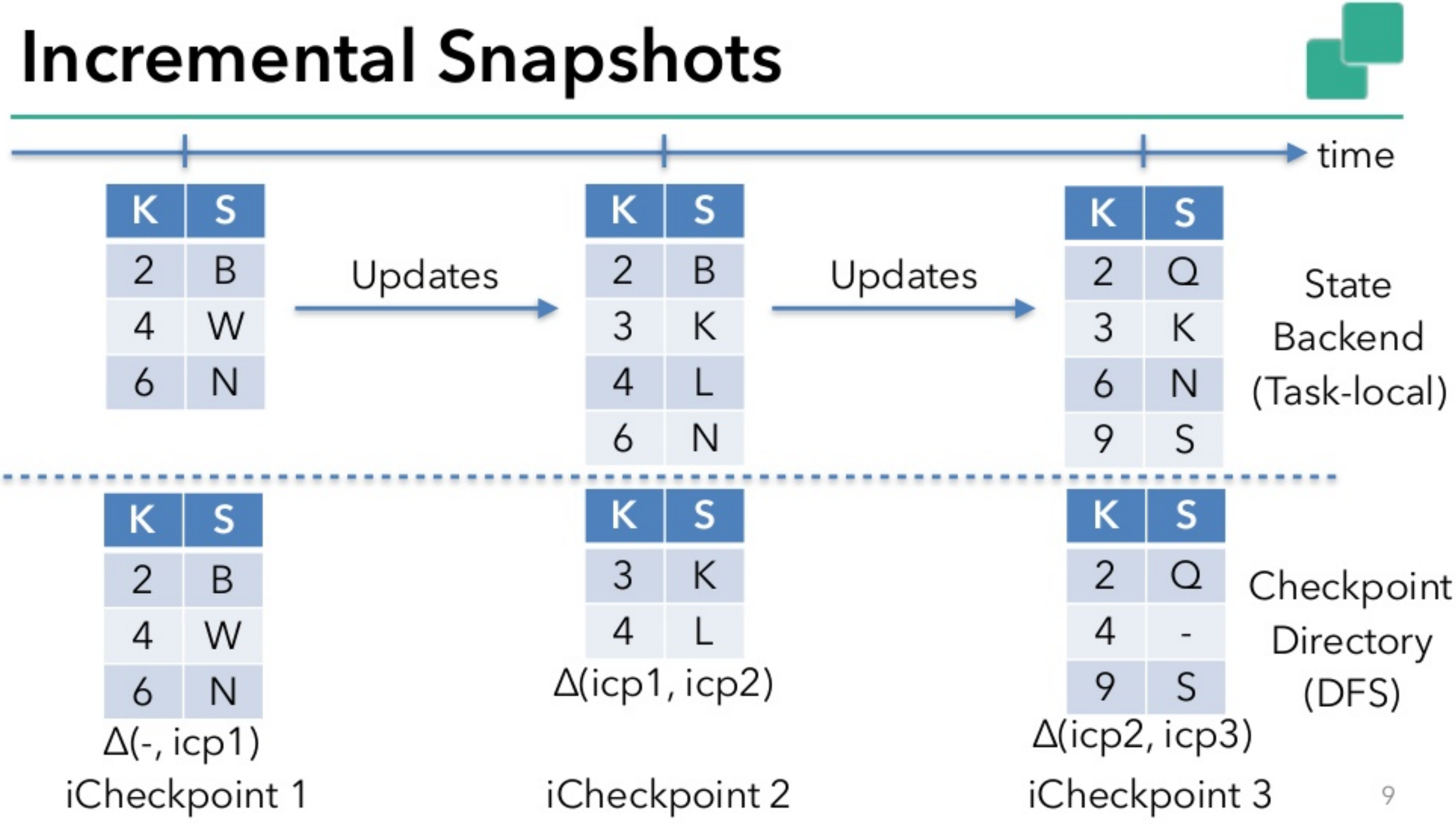
Flink State and Distributed Snapshots



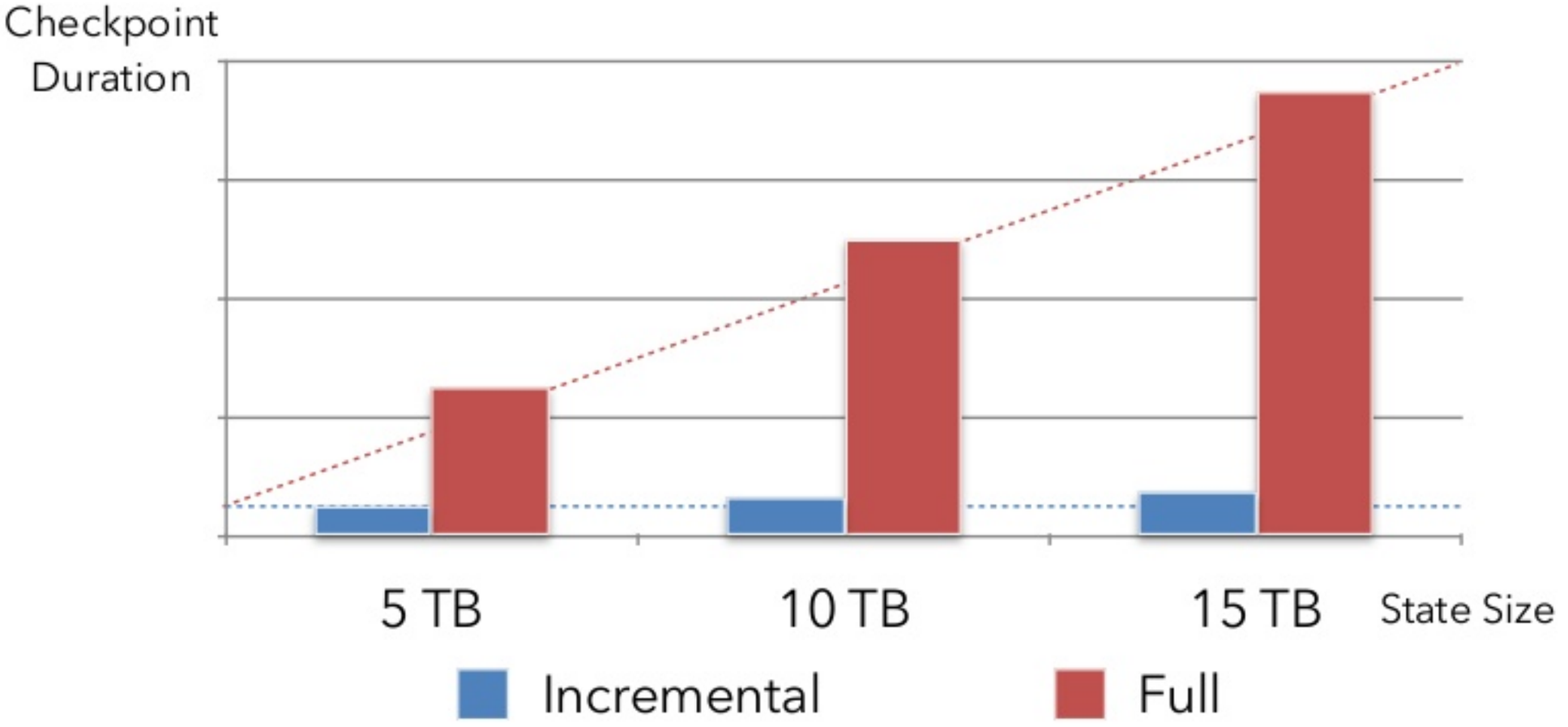
Full Snapshots



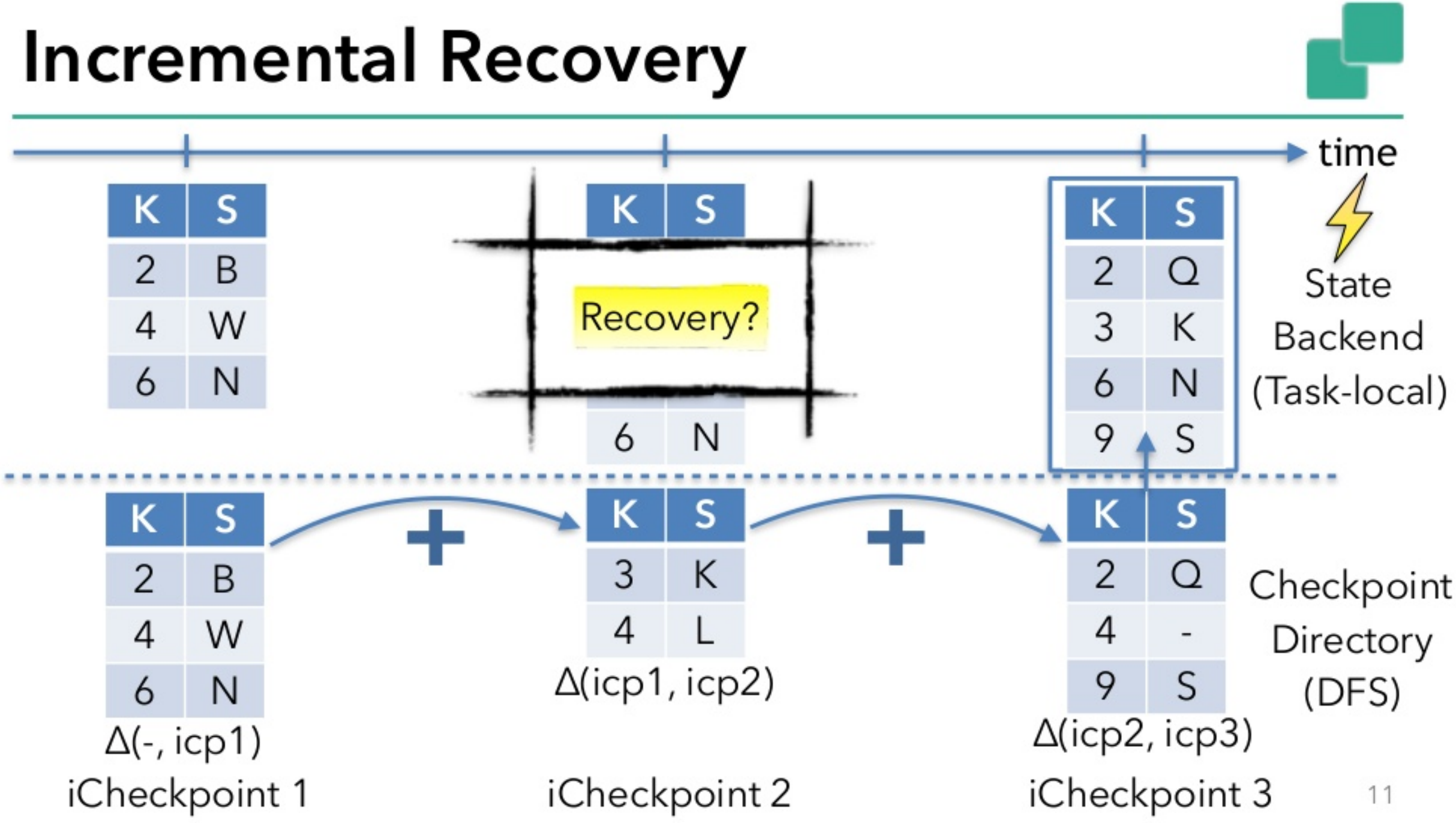
Incremental Snapshots



Incremental vs Full Snapshots



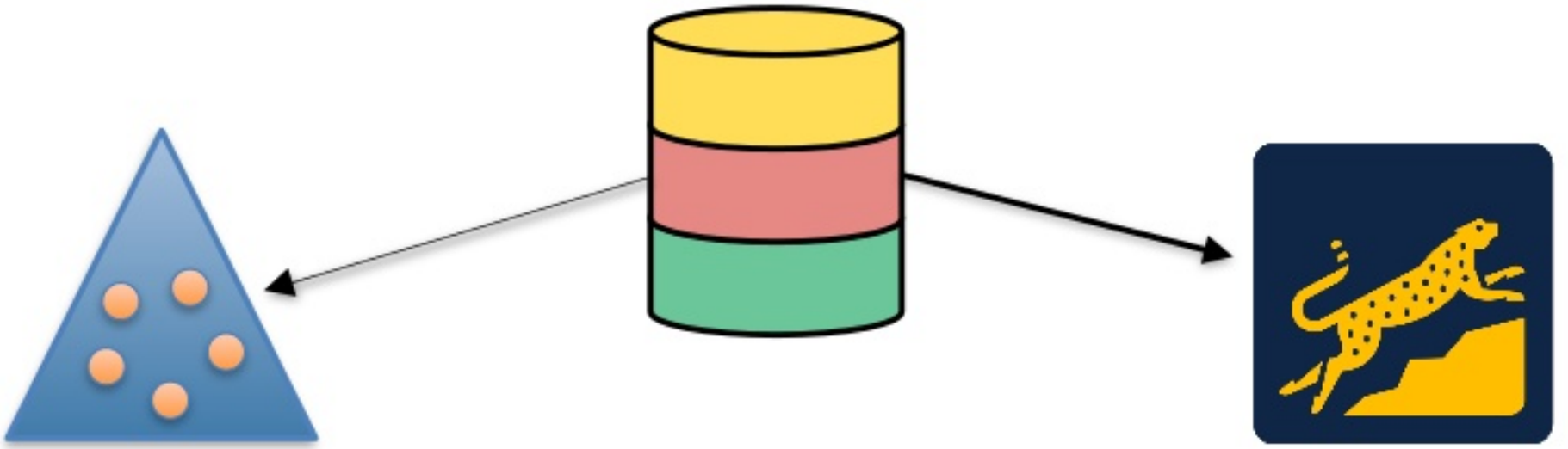
Incremental Recovery



Challenges for Data Structures

- Incremental checkpoints:
  - Efficiently detect the (minimal) set of state changes between two checkpoints.
  - Avoid unbounded checkpoint history.

Two Keyed State Backends



HeapKeyedStateBackend

- State lives in memory, on Java heap.
- Operates on objects.
- Think of a hash map {key obj -> state obj}.
- **Goes through ser/de during snapshot/restore.**
- **Async snapshots supported.**

RocksDBKeyedStateBackend

- State lives in off-heap memory and on disk.
- Operates on serialized bytes.
- Think of K/V store {key bytes -> state bytes}.
- Based on log-structured-merge (LSM) tree.
- **Goes through ser/de for each state access.**
- **Async and incremental snapshots.**

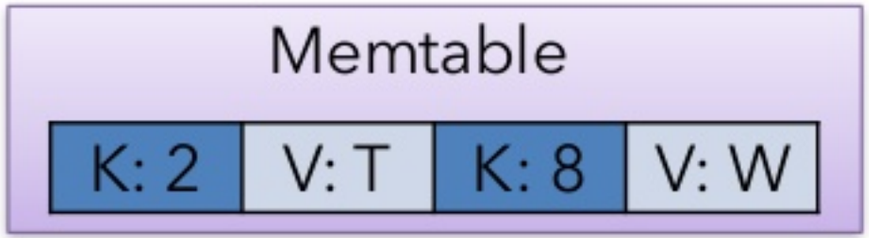
Asynchronous and Incremental Checkpoints with the RocksDB Keyed-State Backend

What is RocksDB

- Ordered Key/Value store (bytes).
- Based on LSM trees.
- Embedded, written in C++, used via JNI.
- Write optimized (all bulk sequential), with acceptable read performance.

RocksDB Architecture (simplified)

Memory



- Mutable Memory Buffer for K/V Pairs (bytes)
- All reads / writes go here first
- Aggregating (overrides same unique keys)
- Asynchronously flushed to disk when full

Local Disk

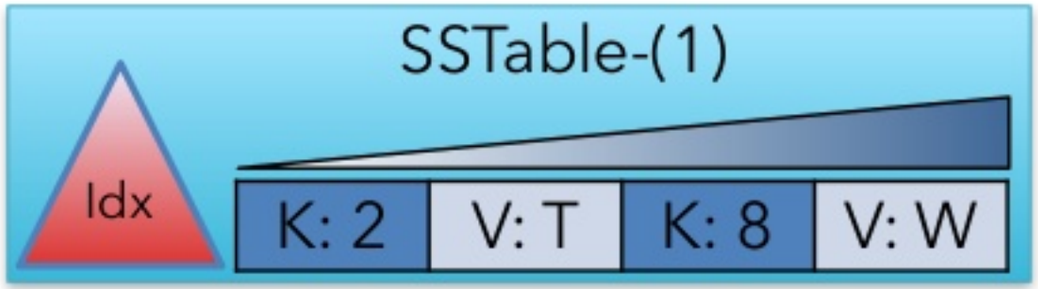
RocksDB Architecture (simplified)

Memory

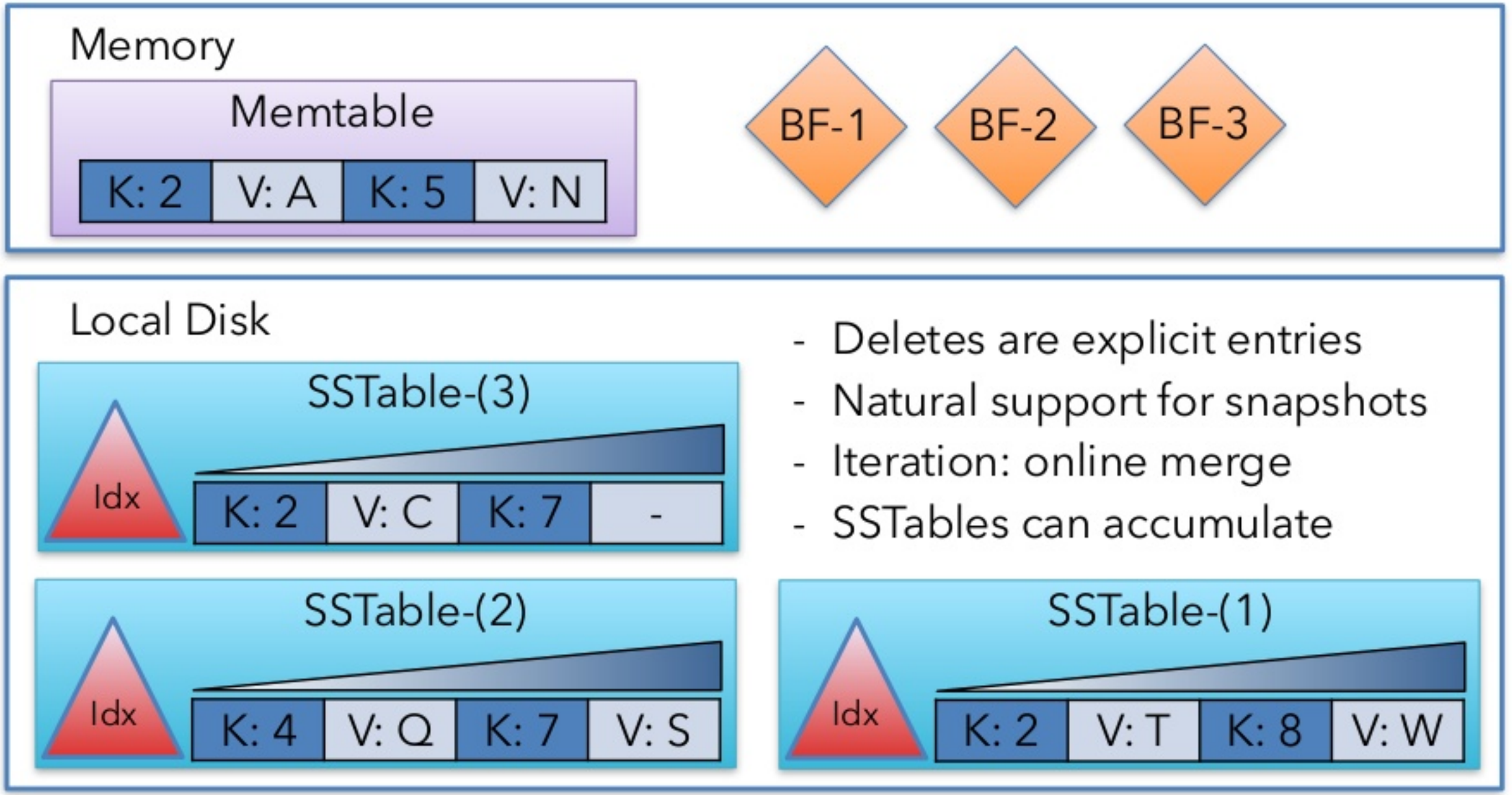
Memtable

Local Disk

- Flushed Memtable becomes **immutable**
- Sorted By Key (unique)
- Read: first check Memtable, then SSTable
- Bloomfilter & Index as optimisations

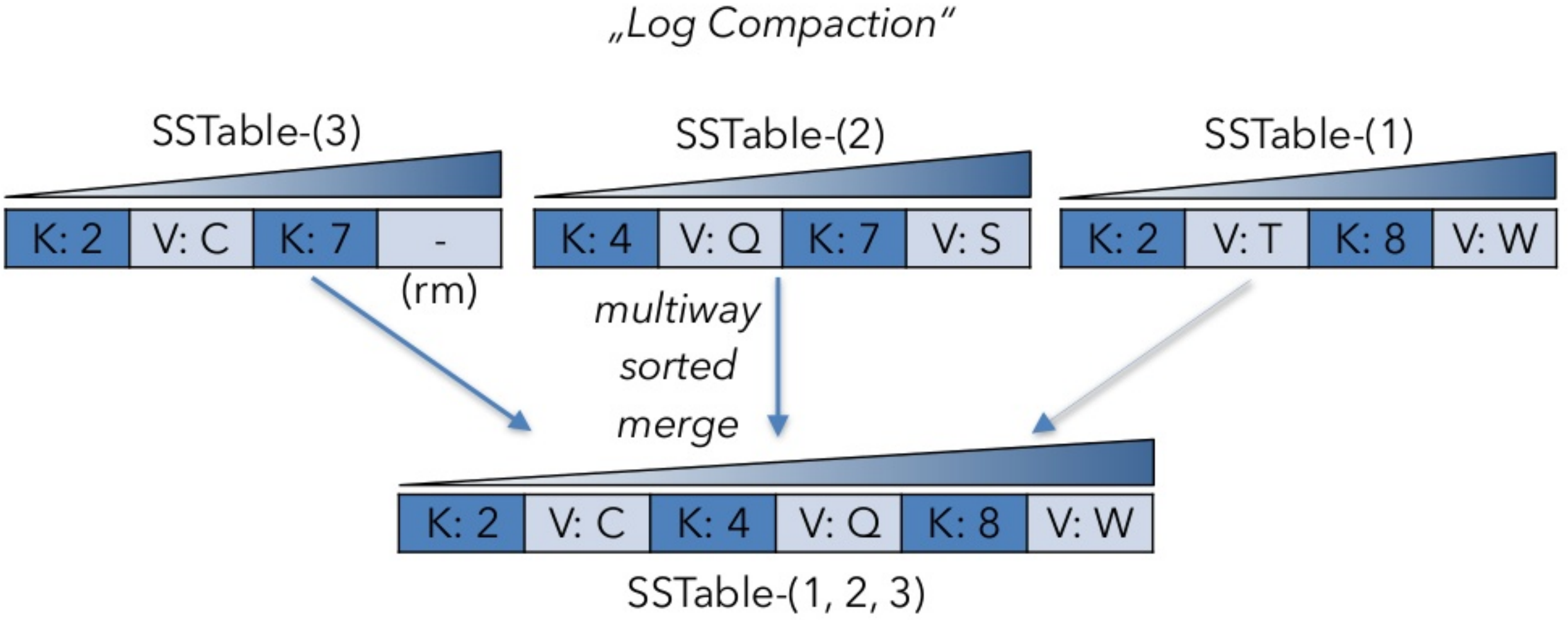


RocksDB Architecture (simplified)



- Deletes are explicit entries
- Natural support for snapshots
- Iteration: online merge
- SSTables can accumulate

RocksDB Compaction (simplified)



RocksDB Asynchronous Checkpoint

- Flush Memtable. (Simplified)
- Then create iterator over all current SSTables (they are immutable).
- New changes go to Memtable and future SSTables and are not considered by iterator.

RocksDB Incremental Checkpoint

- Flush Memtable.
- For all SSTable files in local working directory: upload new files since last checkpoint to DFS, re-reference other files.
- Do reference counting on JobManager for uploaded files.

Incremental Checkpoints - Example

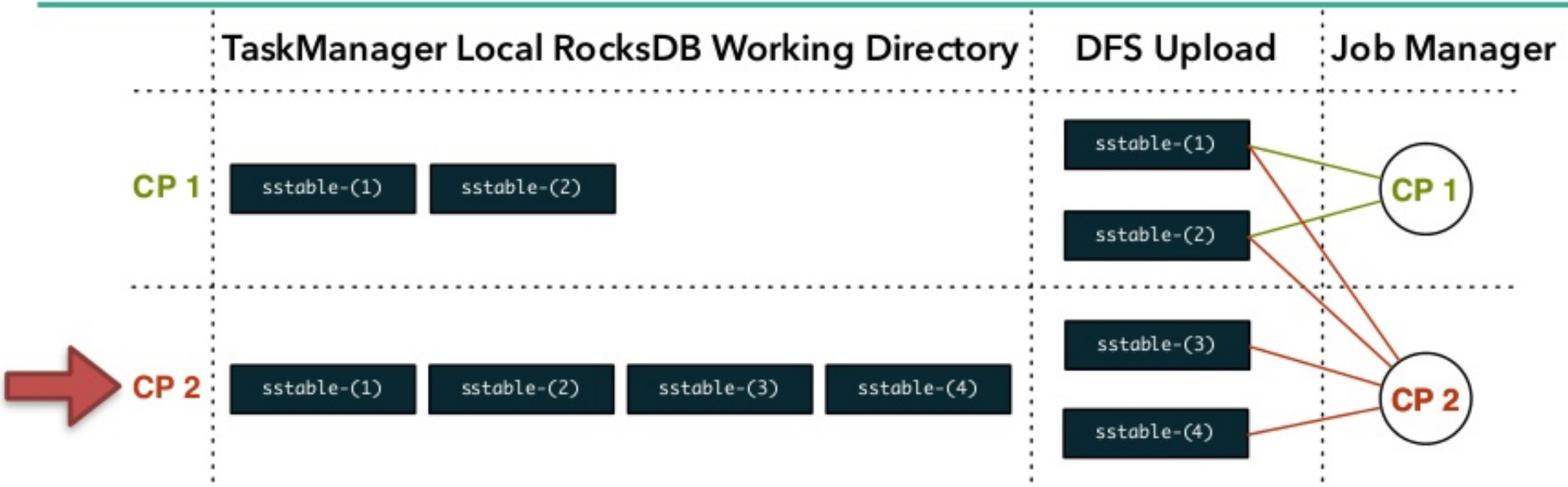
- Illustrate interactions (simplified) between:
  - Local RocksDB instance
  - Remote Checkpoint Directory (DFS)
  - Job Manager (Checkpoint Coordinator)
- In this example: 2 latest checkpoints retained

Incremental Checkpoints - Example

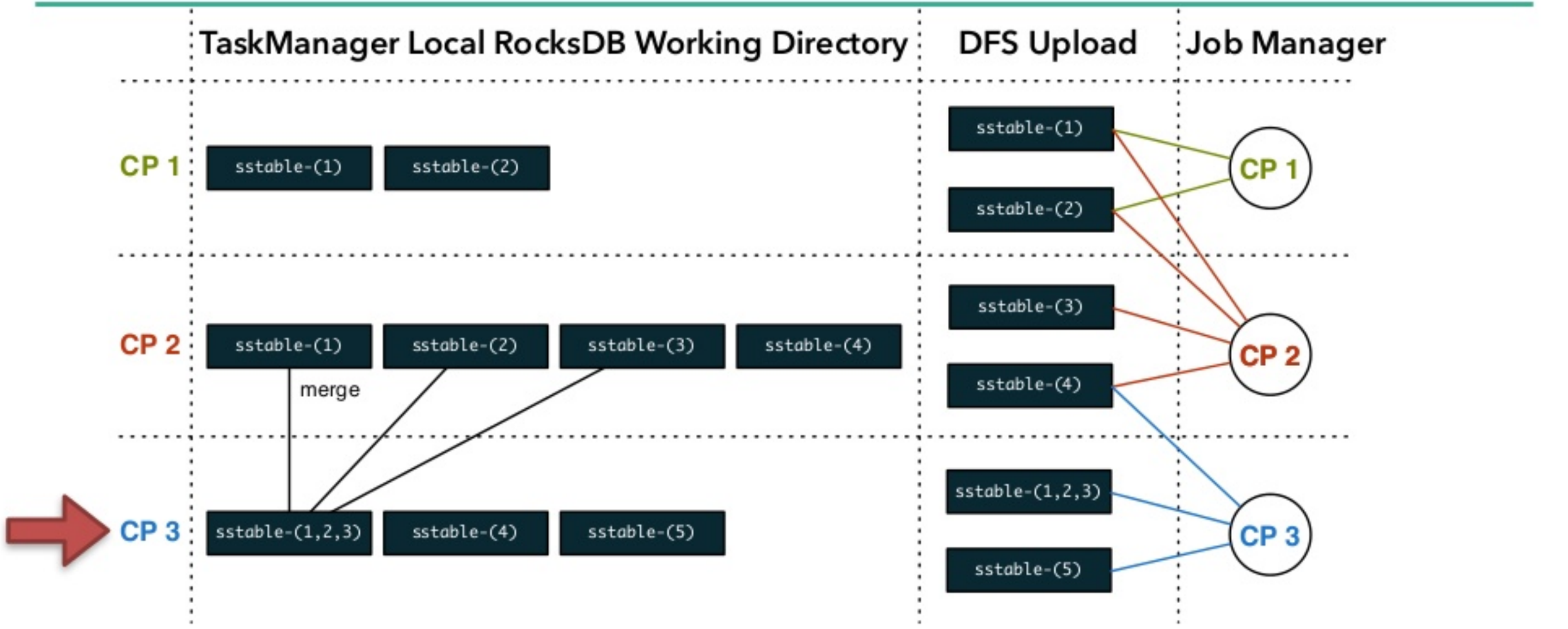
Incremental Checkpoints - Example

Incremental Checkpoints - Example

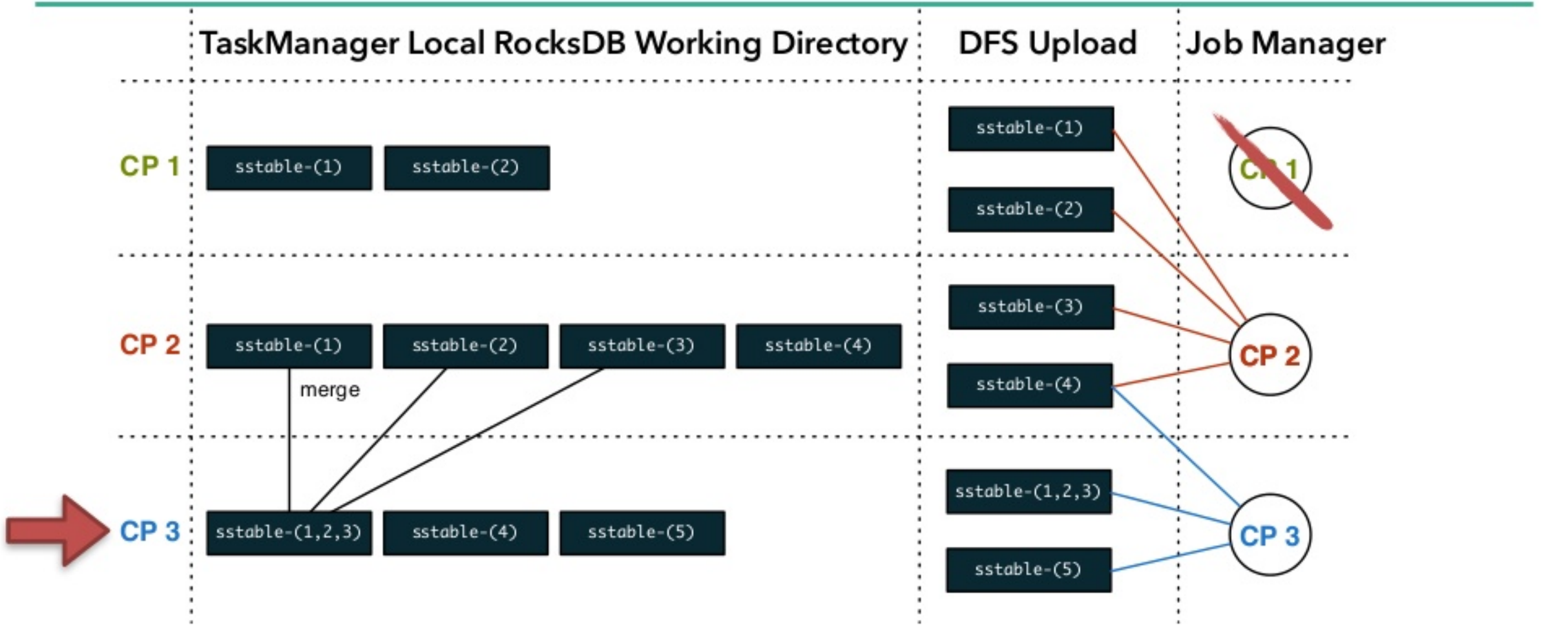
Incremental Checkpoints - Example



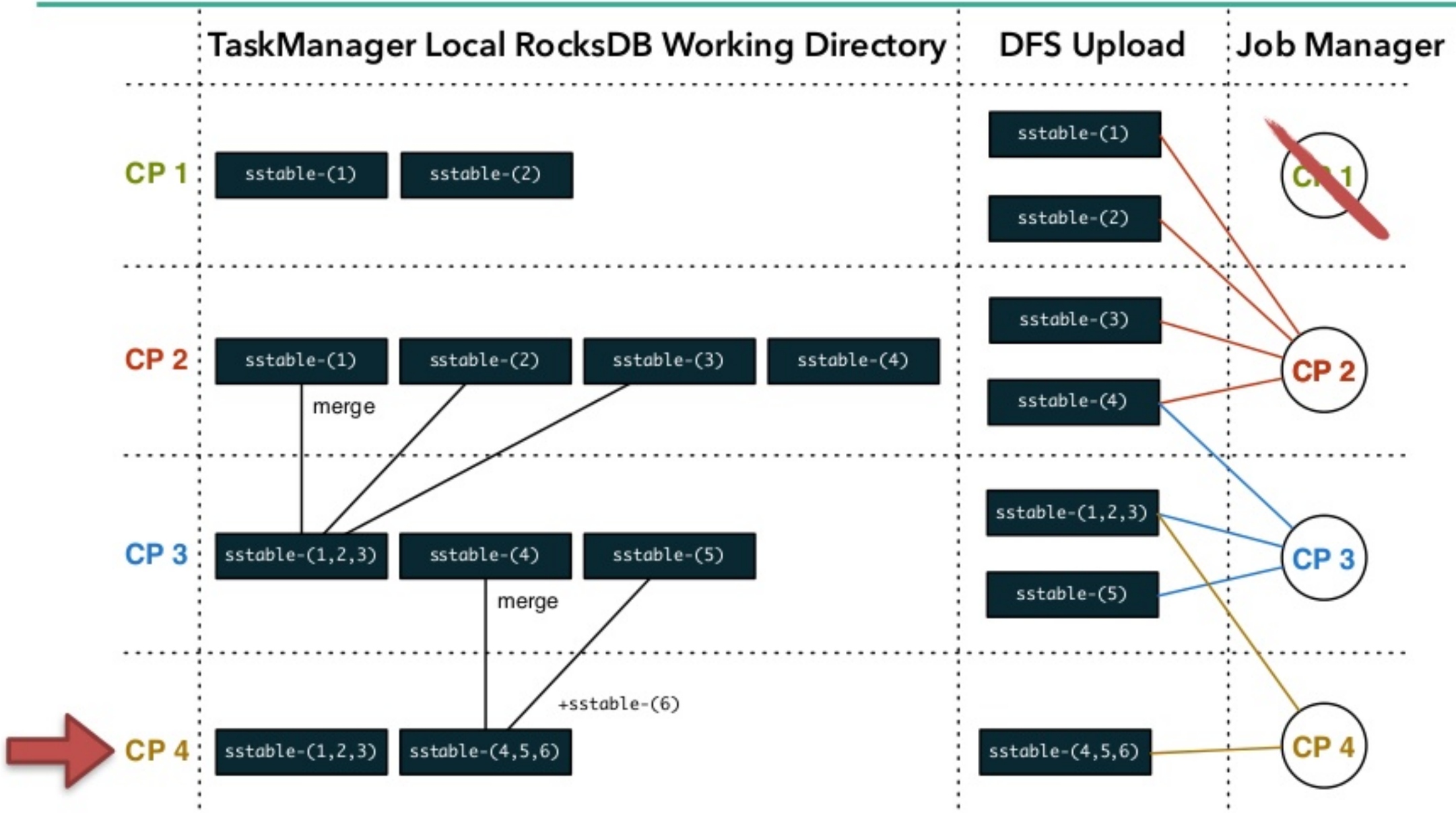
Incremental Checkpoints - Example



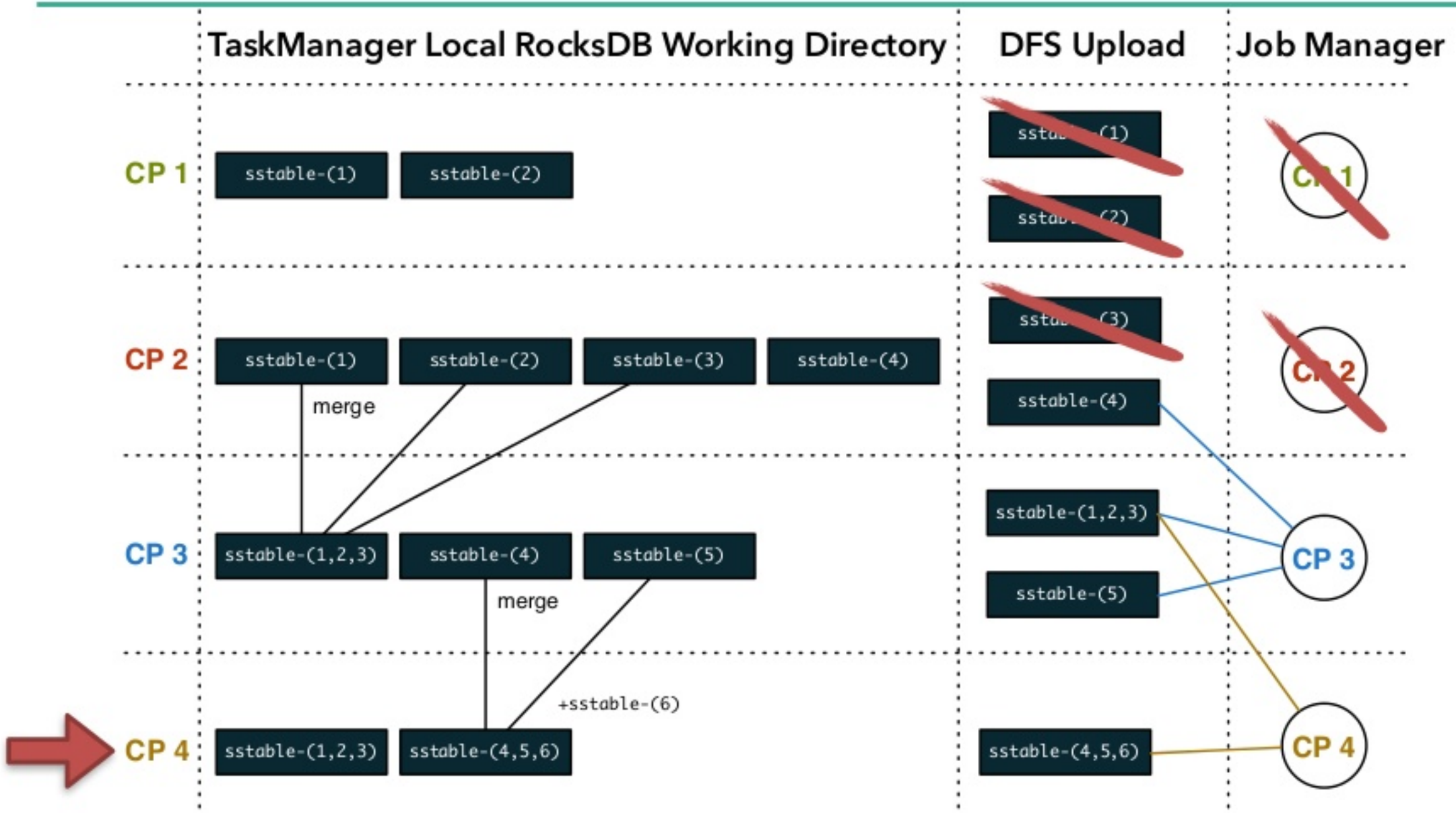
Incremental Checkpoints - Example



Incremental Checkpoints - Example



Incremental Checkpoints - Example

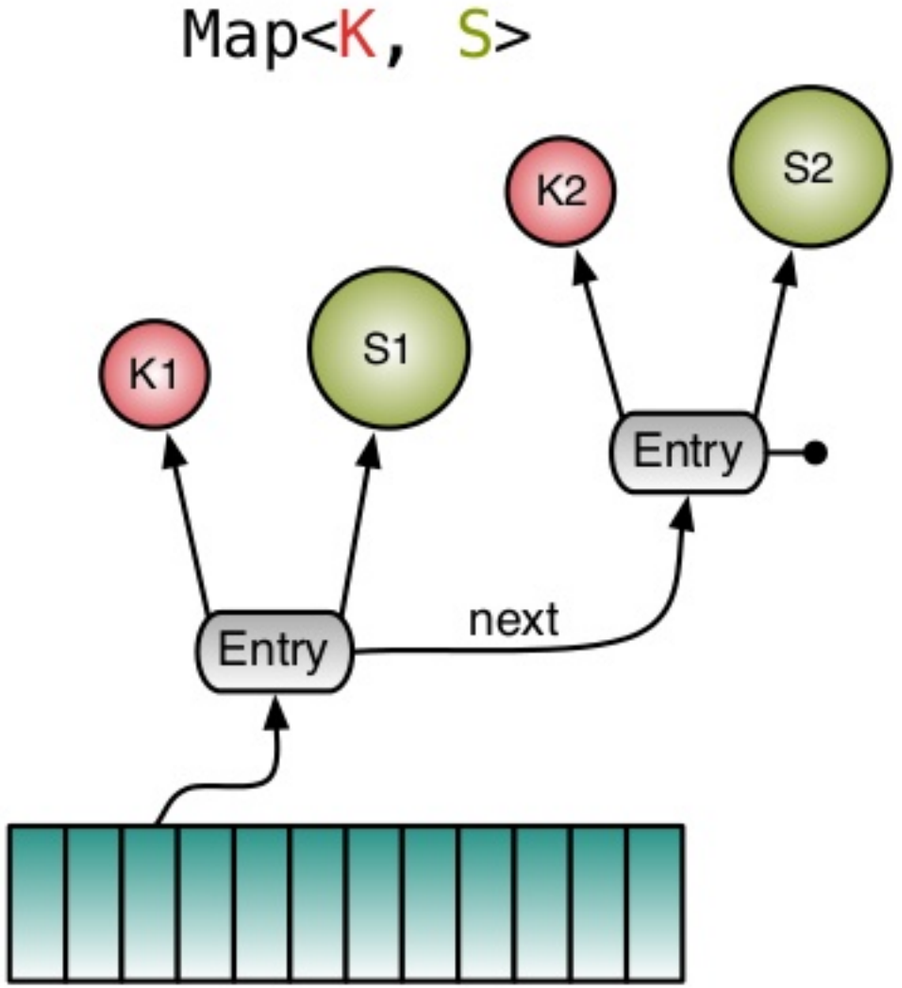


Summary RocksDB

- Asynchronous snapshots „for free“, immutable copy is already there.
- Trivially supports multiple concurrent snapshots.
- Detection of incremental changes easy: observe creation and deletion of SSTables.
- Bounded checkpoint history through compaction.

Asynchronous Checkpoints with the Heap Keyed-State Backend

Heap Backend: Chained Hash Map



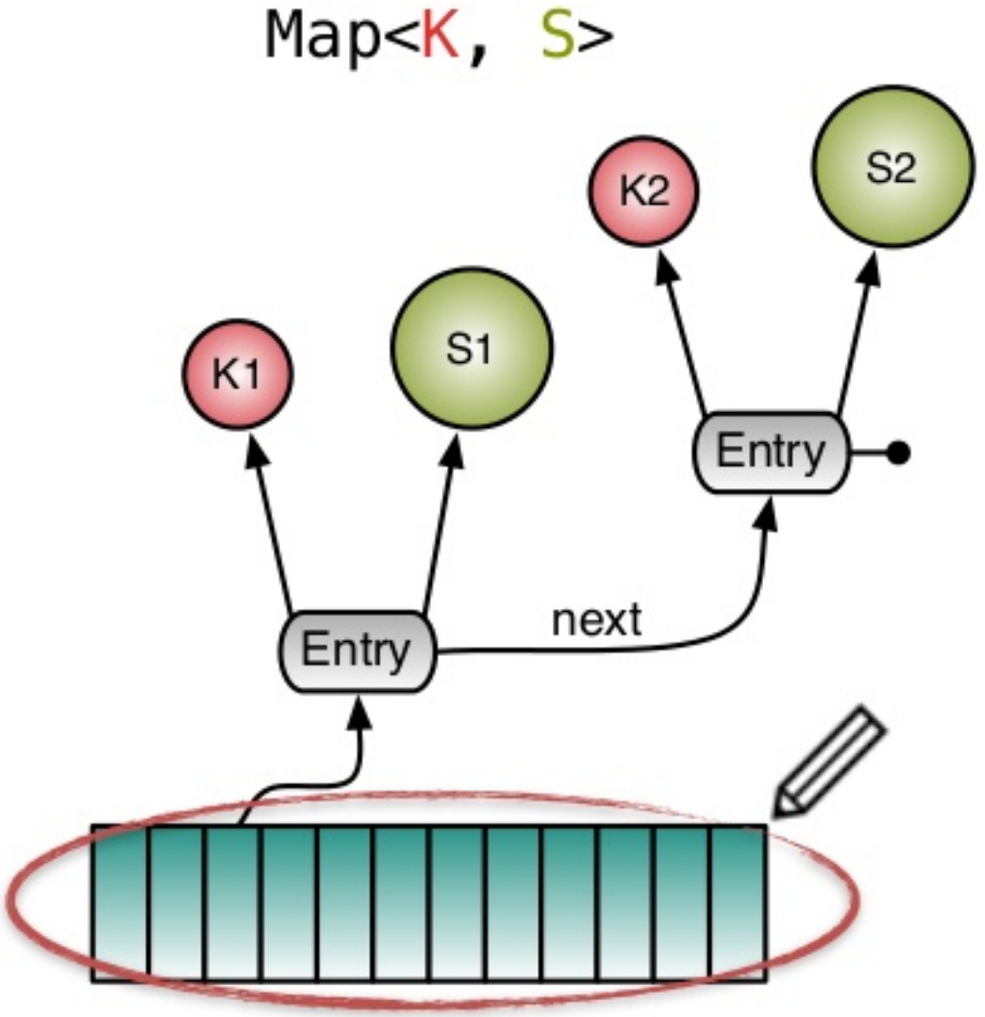

```
Map<K, S> {

    Entry<K, S>[] table;
}
```

```
Entry<K, S> {

    final K key;
    S state;
    Entry next;
}
```

Heap Backend: Chained Hash Map



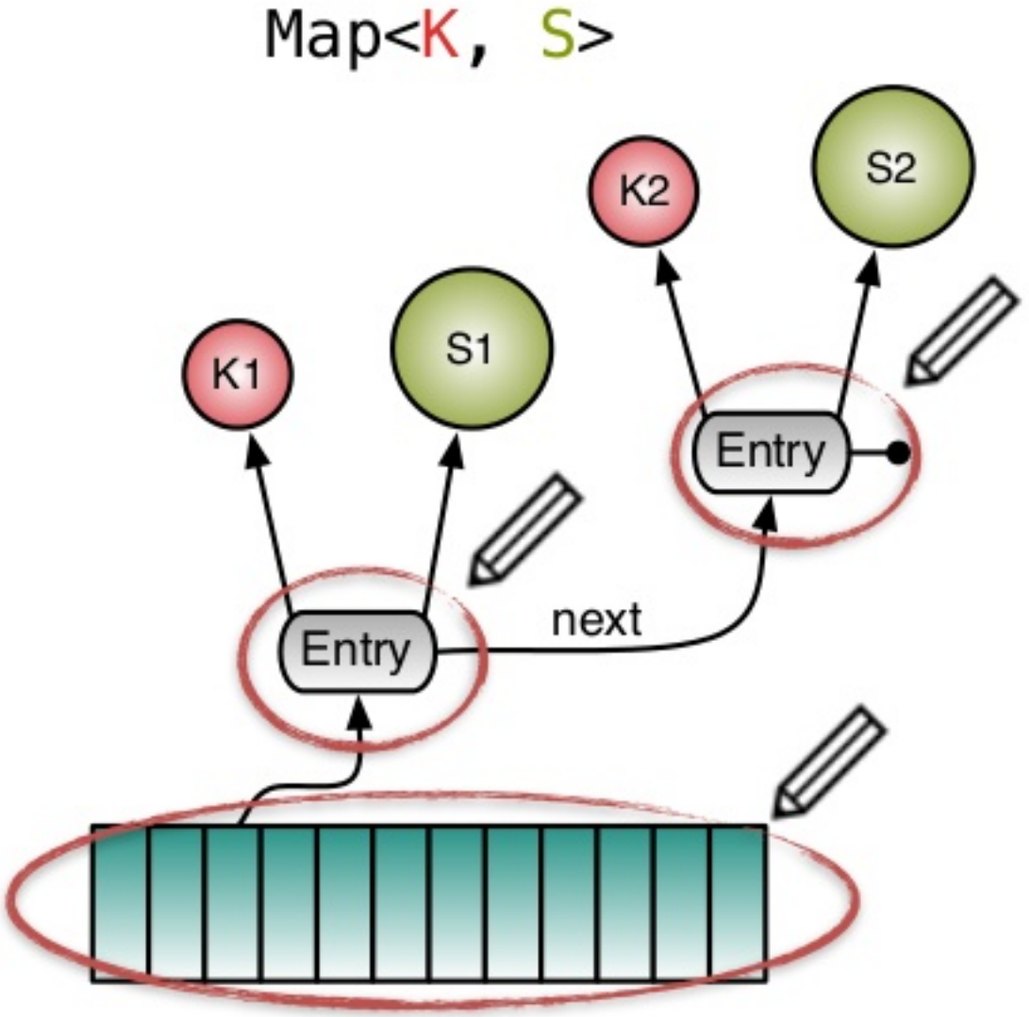

```
Map<K, S> {

    Entry<K, S>[] table;
}
```

```
Entry<K, S> {

    final K key;
    S state;
    Entry next;
}
```

Heap Backend: Chained Hash Map



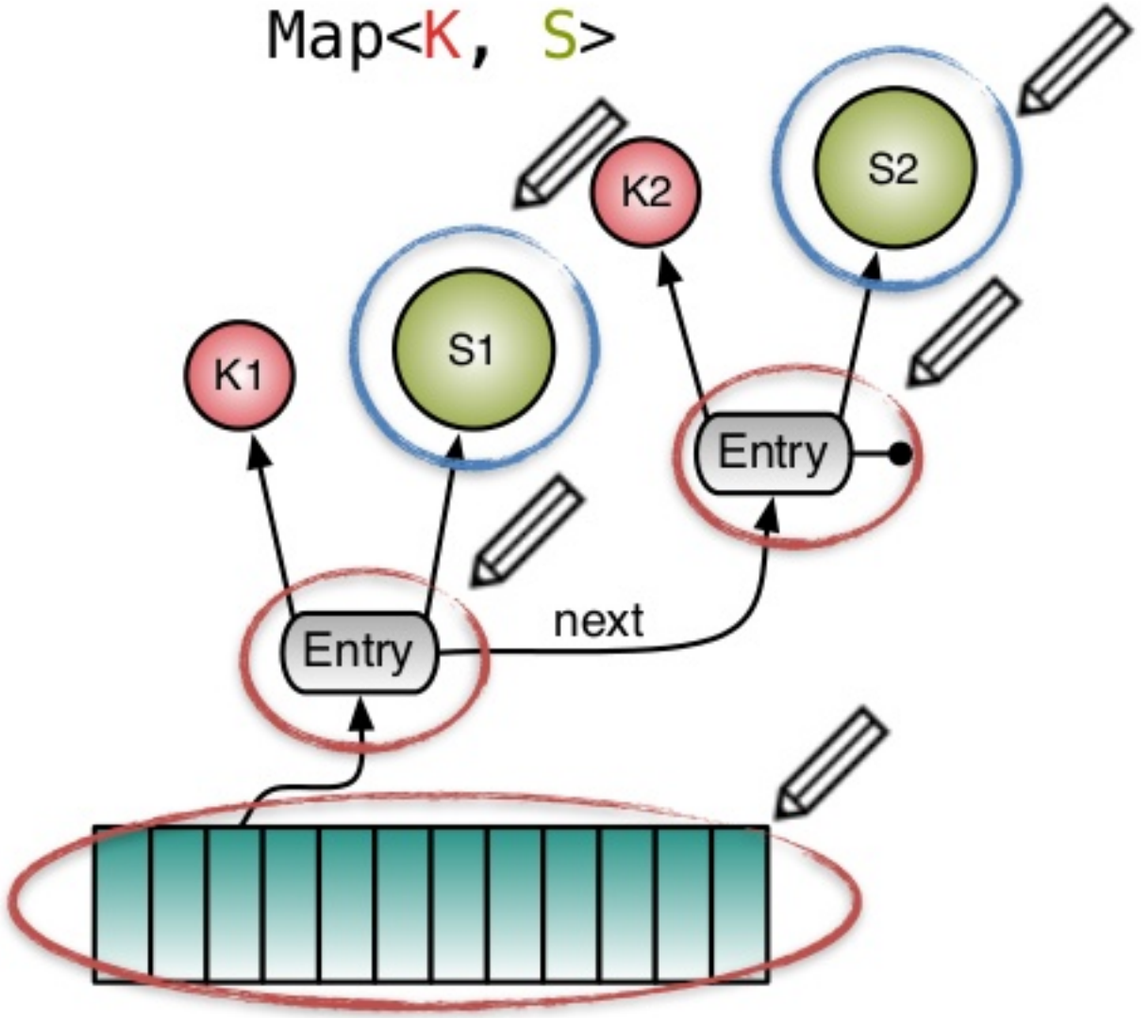

```
Map<K, S> {

    Entry<K, S>[] table;
}
```

```
Entry<K, S> {

    final K key;
    S state;
    Entry next;
}
```

Heap Backend: Chained Hash Map



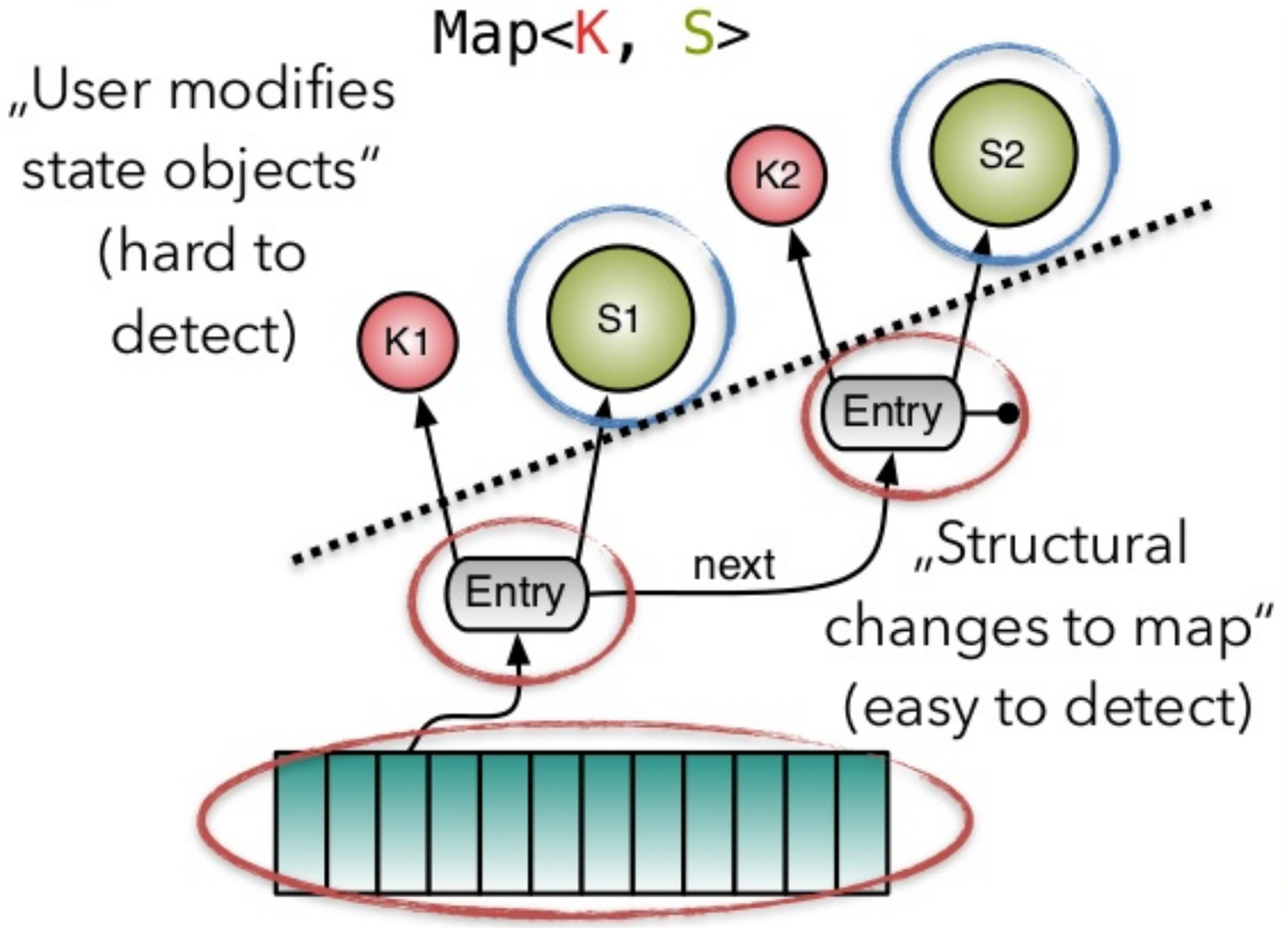

```
Map<K, S> {

    Entry<K, S>[] table;
}
```

```
Entry<K, S> {

    final K key;
    S state;
    Entry next;
}
```

Heap Backend: Chained Hash Map



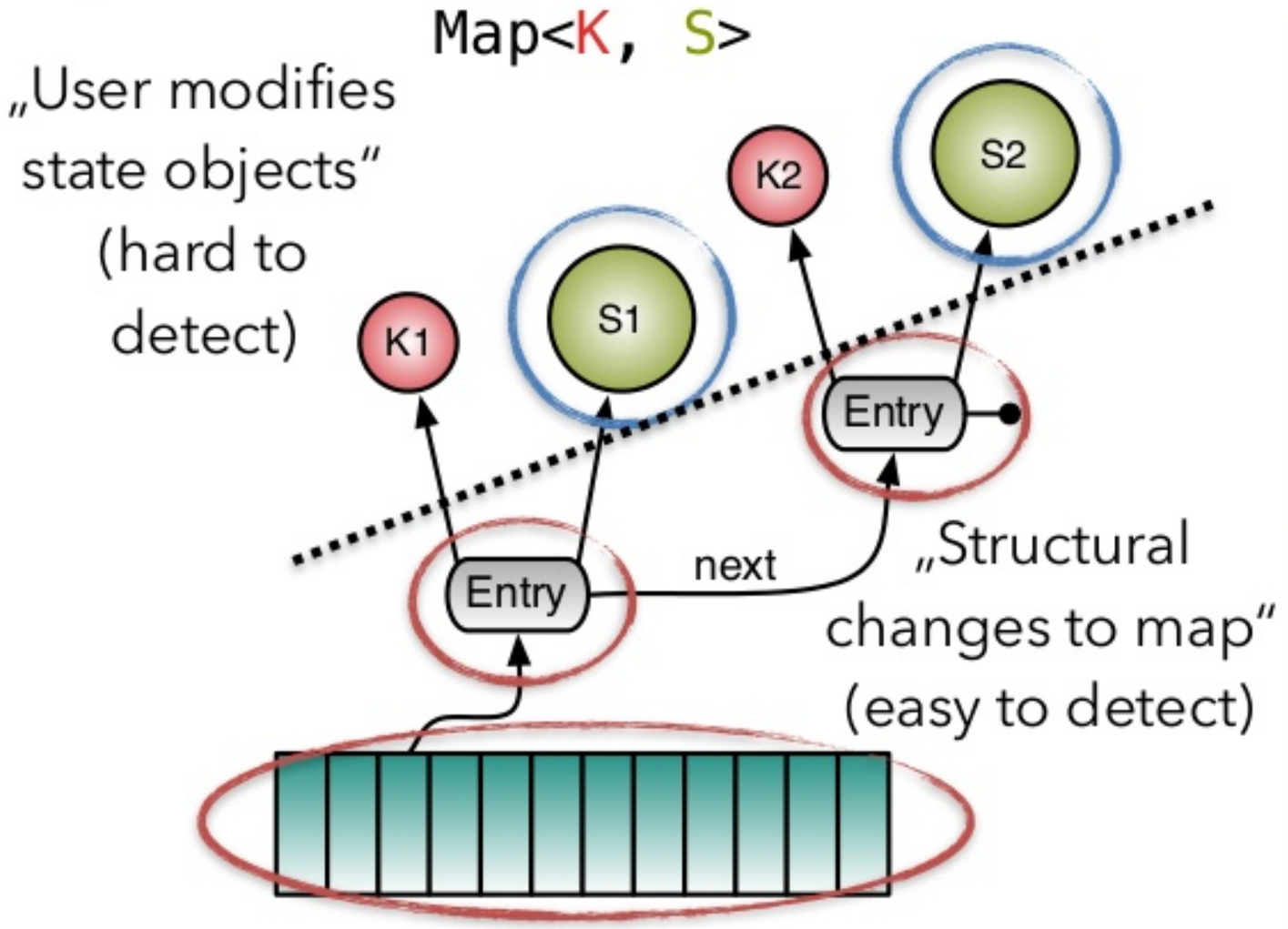

```
Map<K, S> {

    Entry<K, S>[] table;
}
```

```
Entry<K, S> {

    final K key;
    S state;
    Entry next;
}
```

Copy-on-Write Hash Map




```
Map<K, S> {

    Entry<K, S>[] table;

    int mapVersion;
    int requiredVersion;

    OrderedSet<Integer> snapshots;
}
```

```
Entry<K, S> {

    final K key;
    S state;
    Entry next;

    int stateVersion;
    int entryVersion;
}
```

Copy-on-Write Hash Map - Snapshots

Create Snapshot:

1. Flat array-copy of table array
2. snapshots.add(mapVersion);
3. ++mapVersion;
4. requiredVersion = mapVersion;

Release Snapshot:

1. snapshots.remove(snapVersion);
2. requiredVersion = snapshots.getMax();

```
Map<K, S> {

   Entry<K, S>[] table;

   int mapVersion;
   int requiredVersion;

   OrderedSet<Integer> snapshots;
}

Entry<K, S> {

   final K key;
   S state;
   Entry next;

   int stateVersion;
   int entryVersion;
}
```

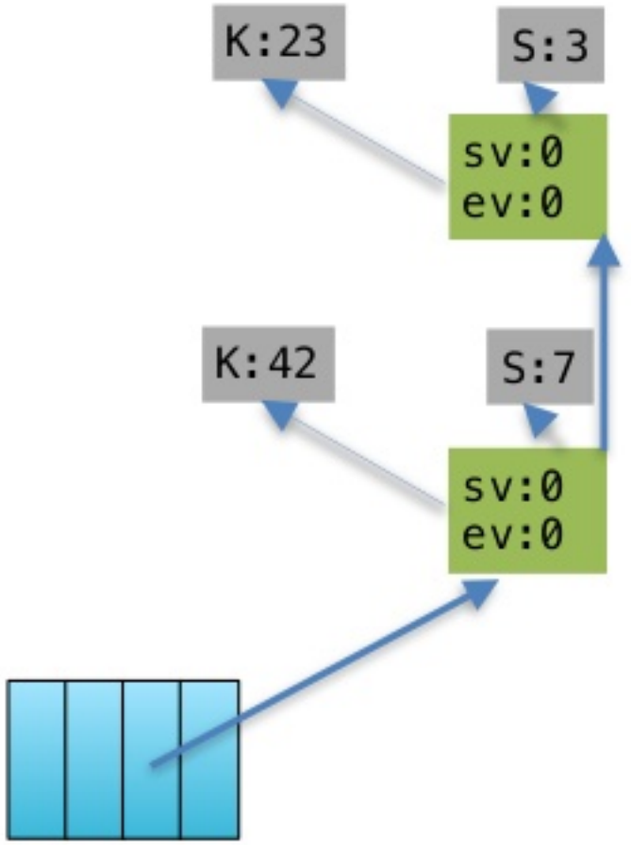
Copy-on-Write Hash Map - 2 Golden Rules

1. *Whenever a map entry* `e` *is modified and* `e.entryVersion < map.requiredVersion`*, first copy the entry and redirect pointers that are reachable through snapshot to the copy. Set* `e.entryVersion = map.mapVersion`*. Pointer redirection can trigger recursive application of rule 1 to other entries.*

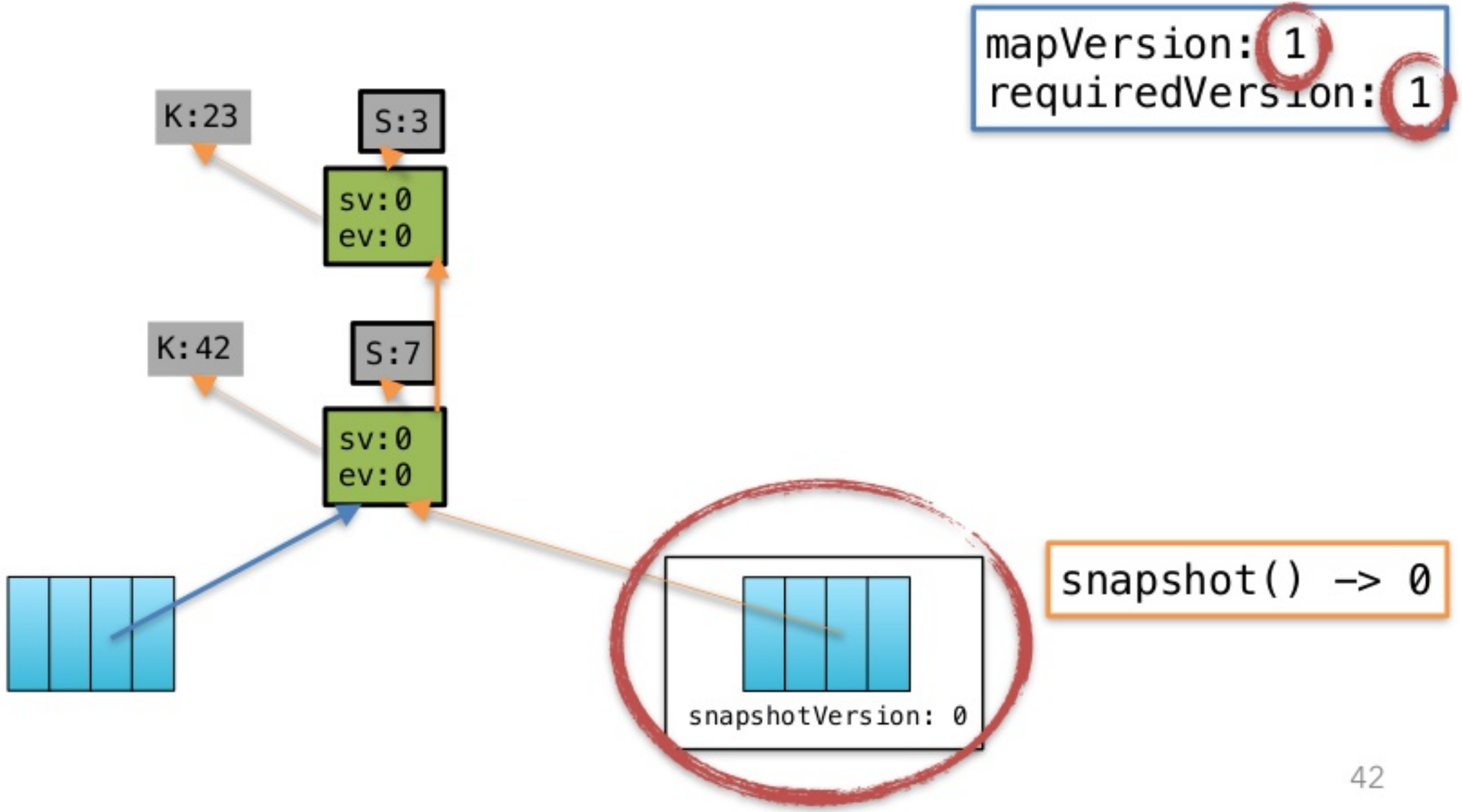
2. *Before returning the state object* `s` *of entry* `e` *to a caller, if* `e.stateVersion < map.requiredVersion`*, create a deep copy of* `s` *and redirect the pointer in* `e` *to the copy. Set* `e.stateVersion = map.mapVersion`. Then return the copy to the caller. *Applying rule 2 can trigger rule 1.*

Copy-on-Write Hash Map - Example

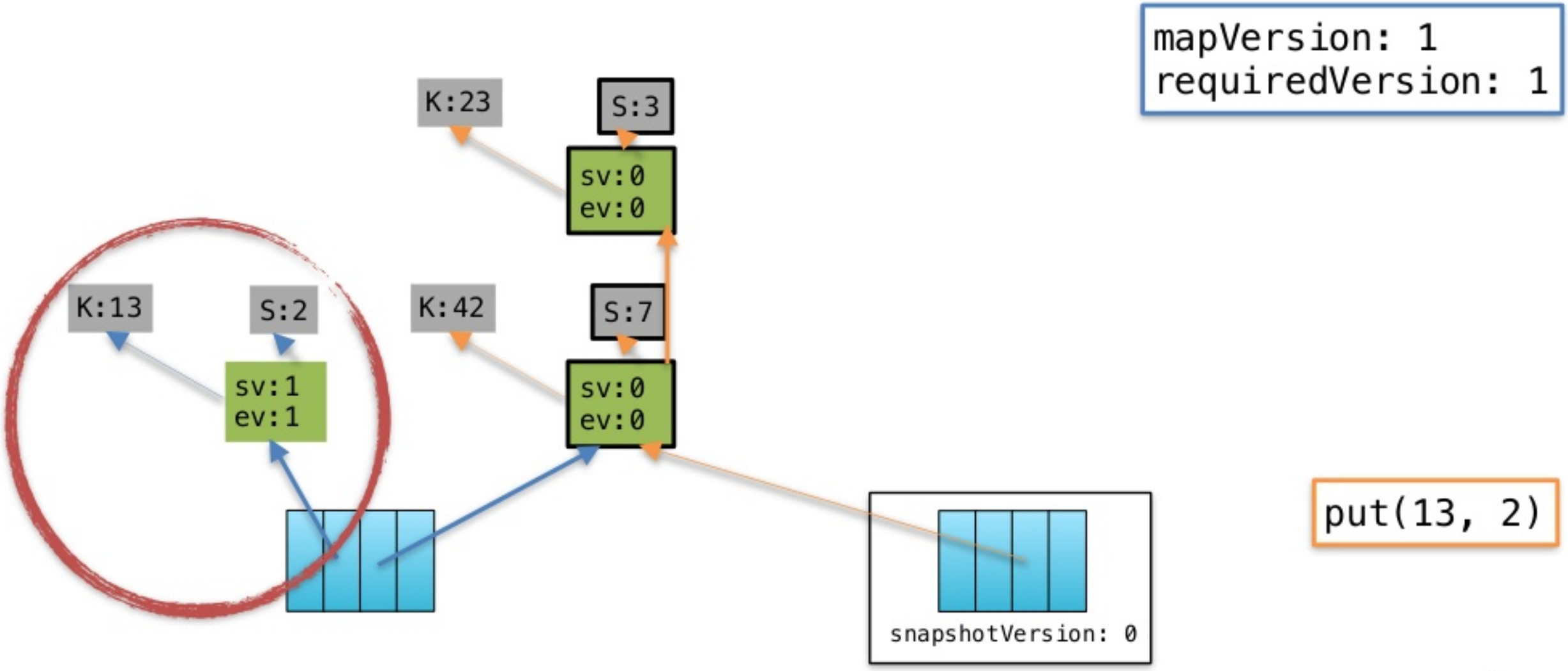
```
mapVersion: 0
requiredVersion: 0
```



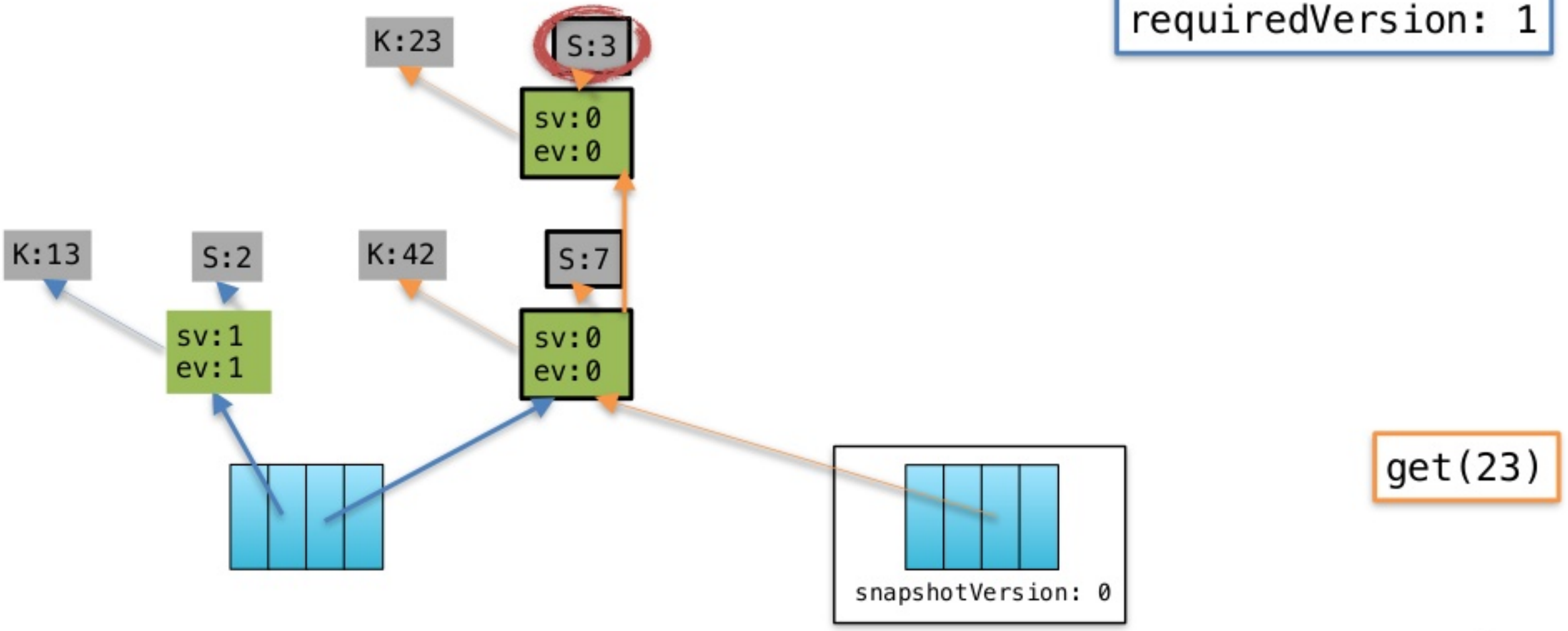
Copy-on-Write Hash Map - Example



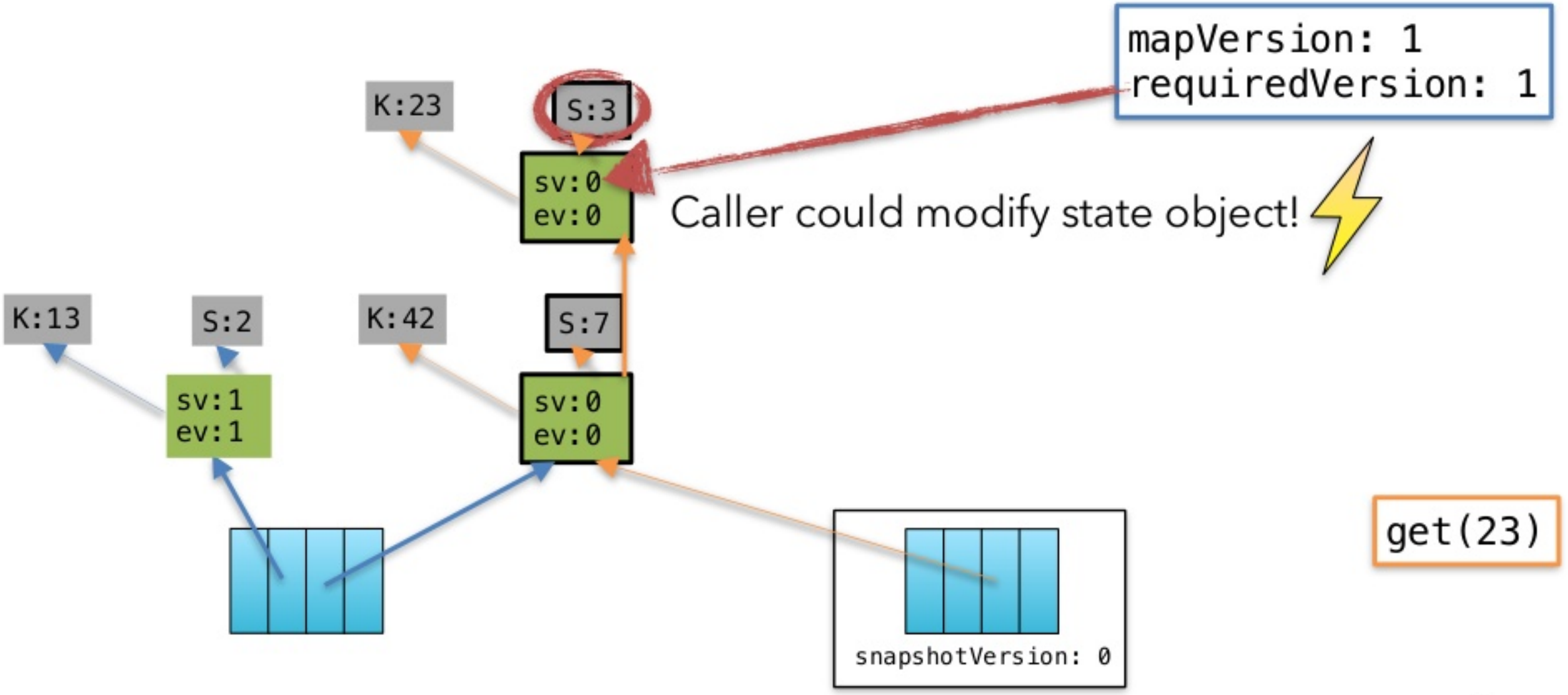
Copy-on-Write Hash Map - Example



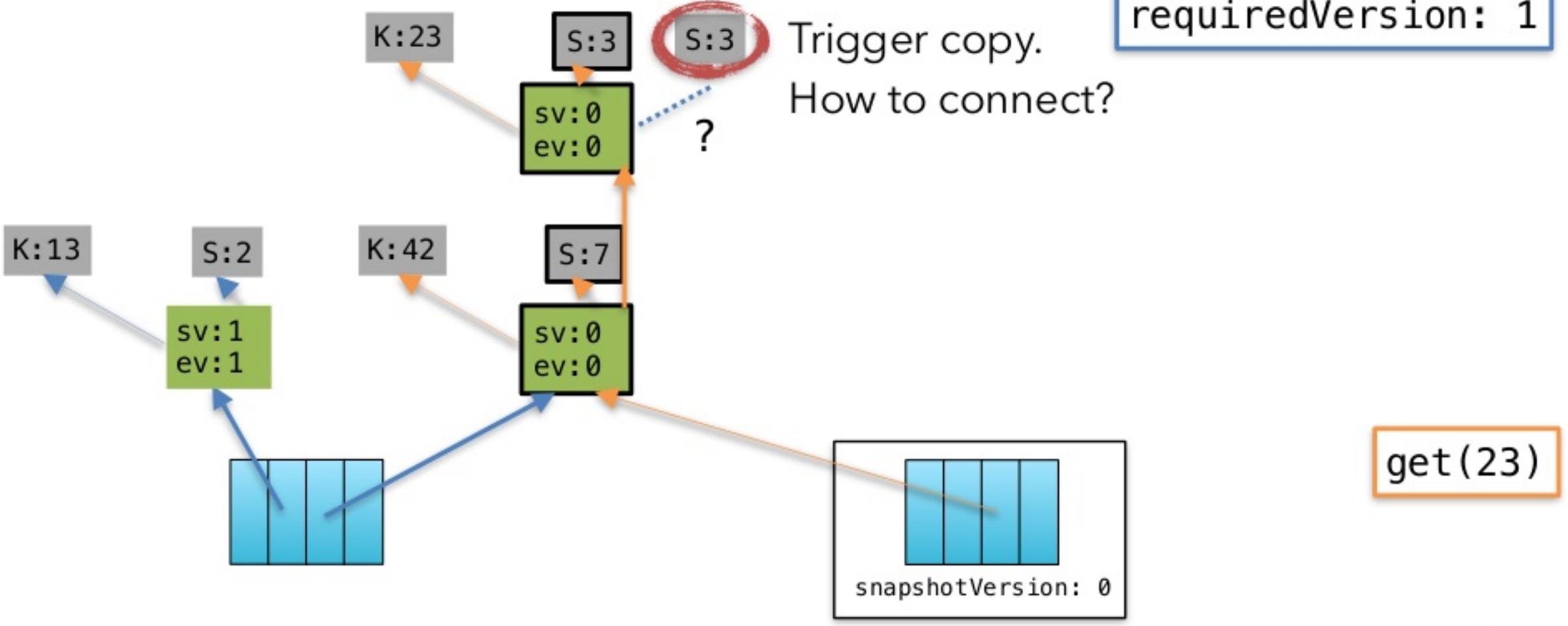
Copy-on-Write Hash Map - Example



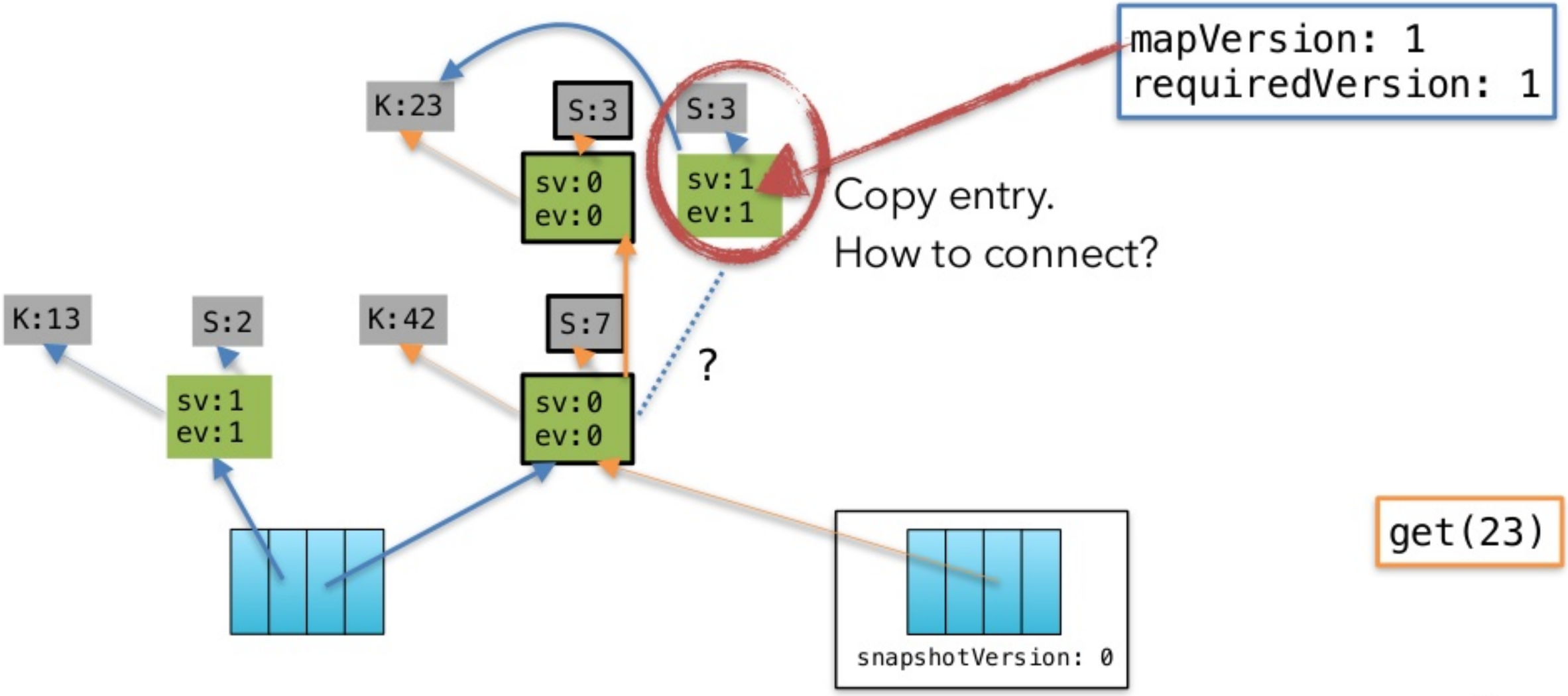
Copy-on-Write Hash Map - Example



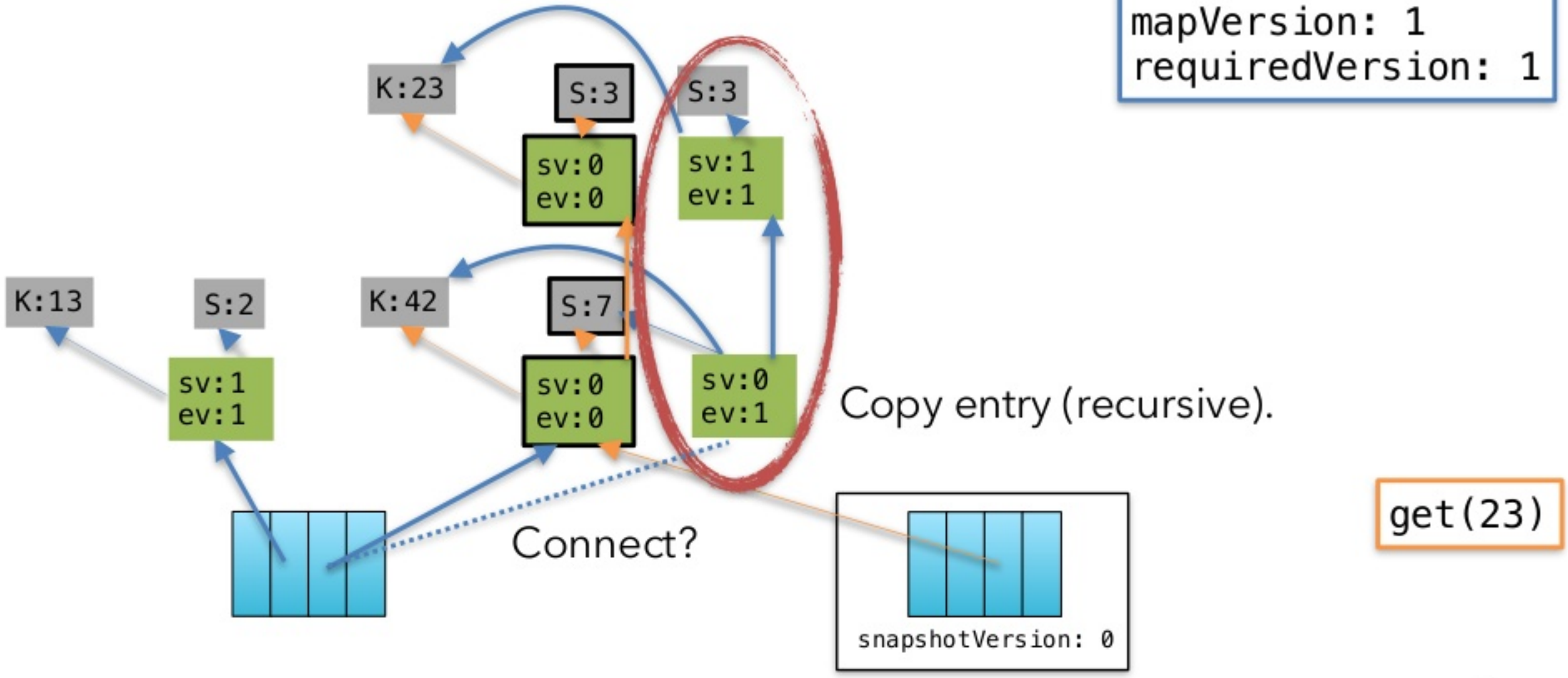
Copy-on-Write Hash Map - Example



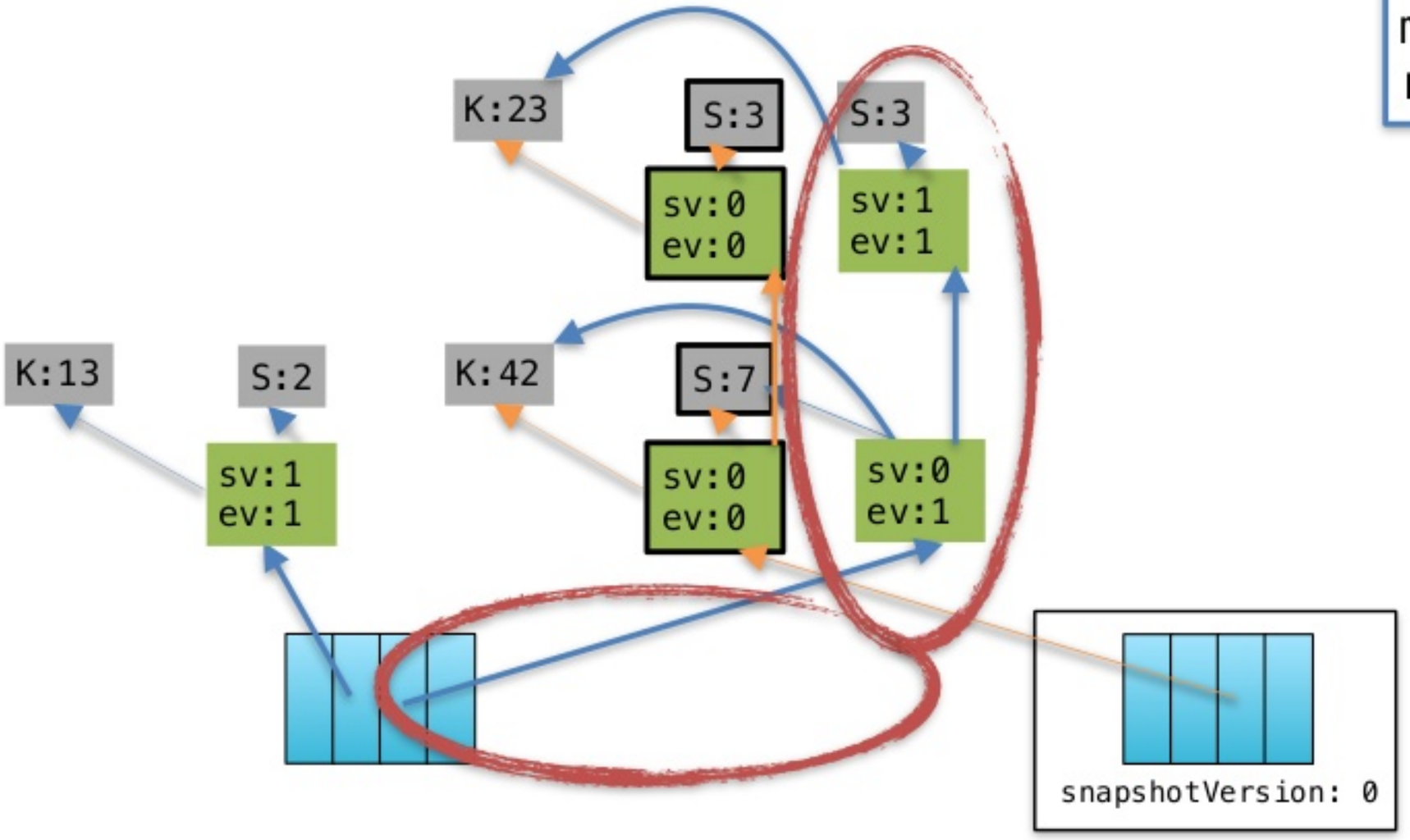
Copy-on-Write Hash Map - Example



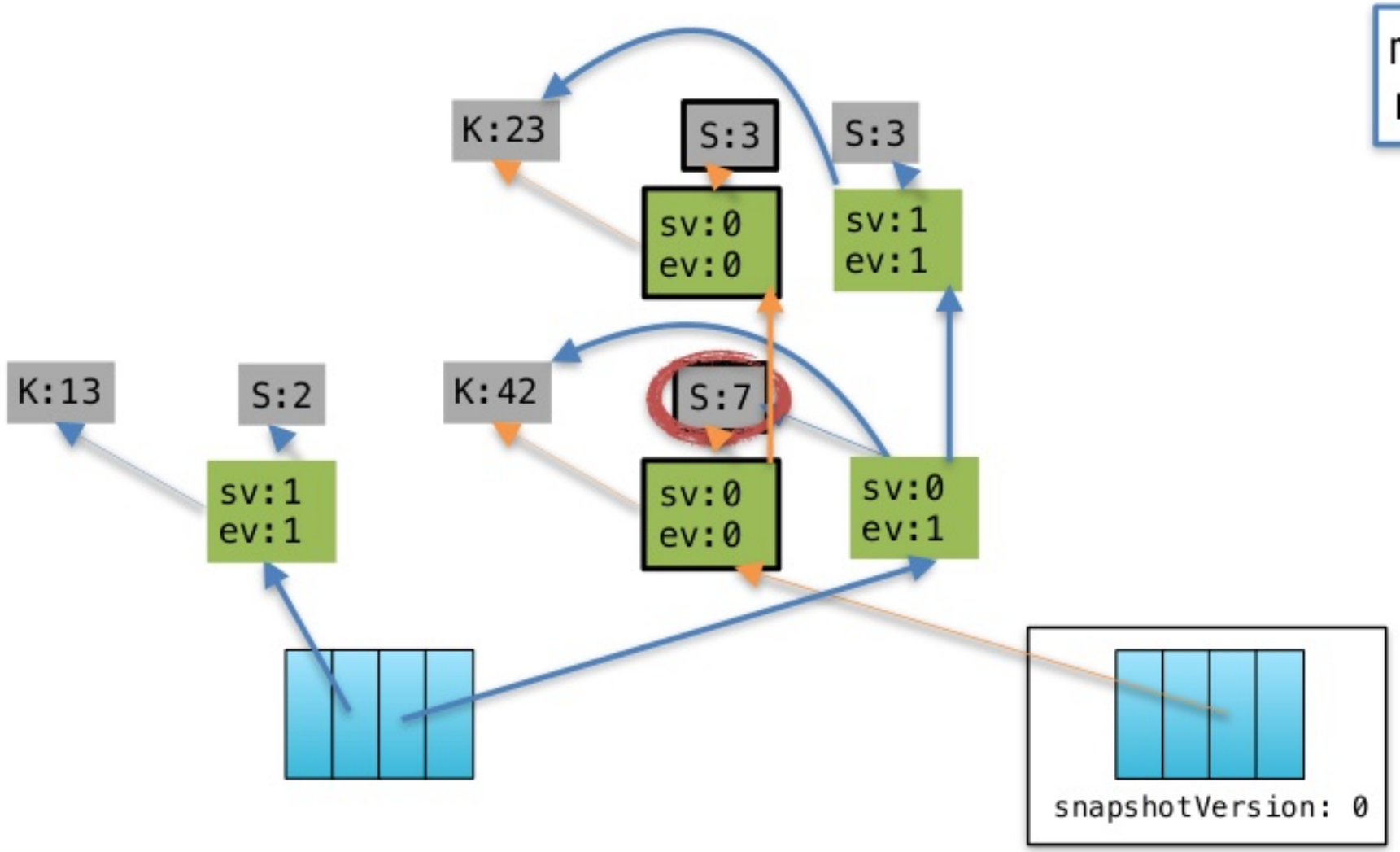
Copy-on-Write Hash Map - Example



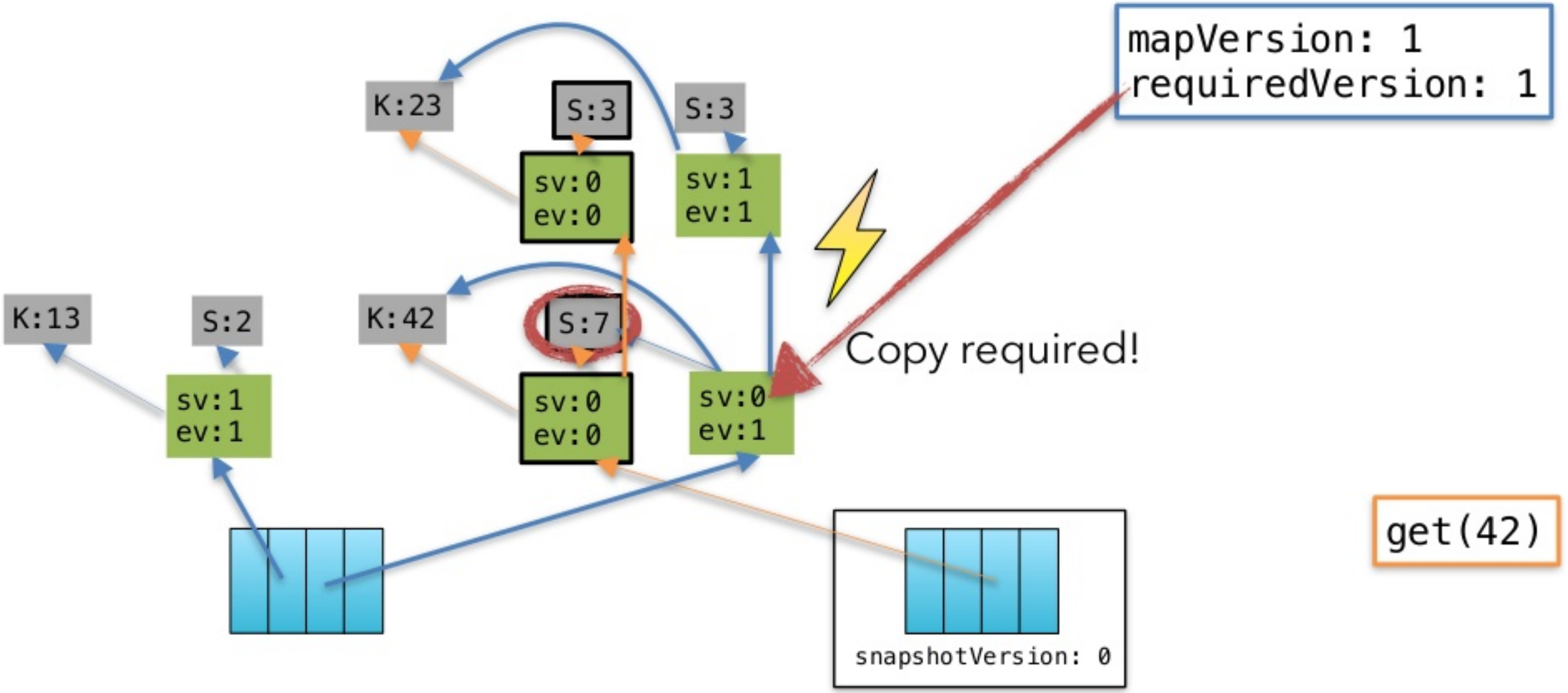
Copy-on-Write Hash Map - Example



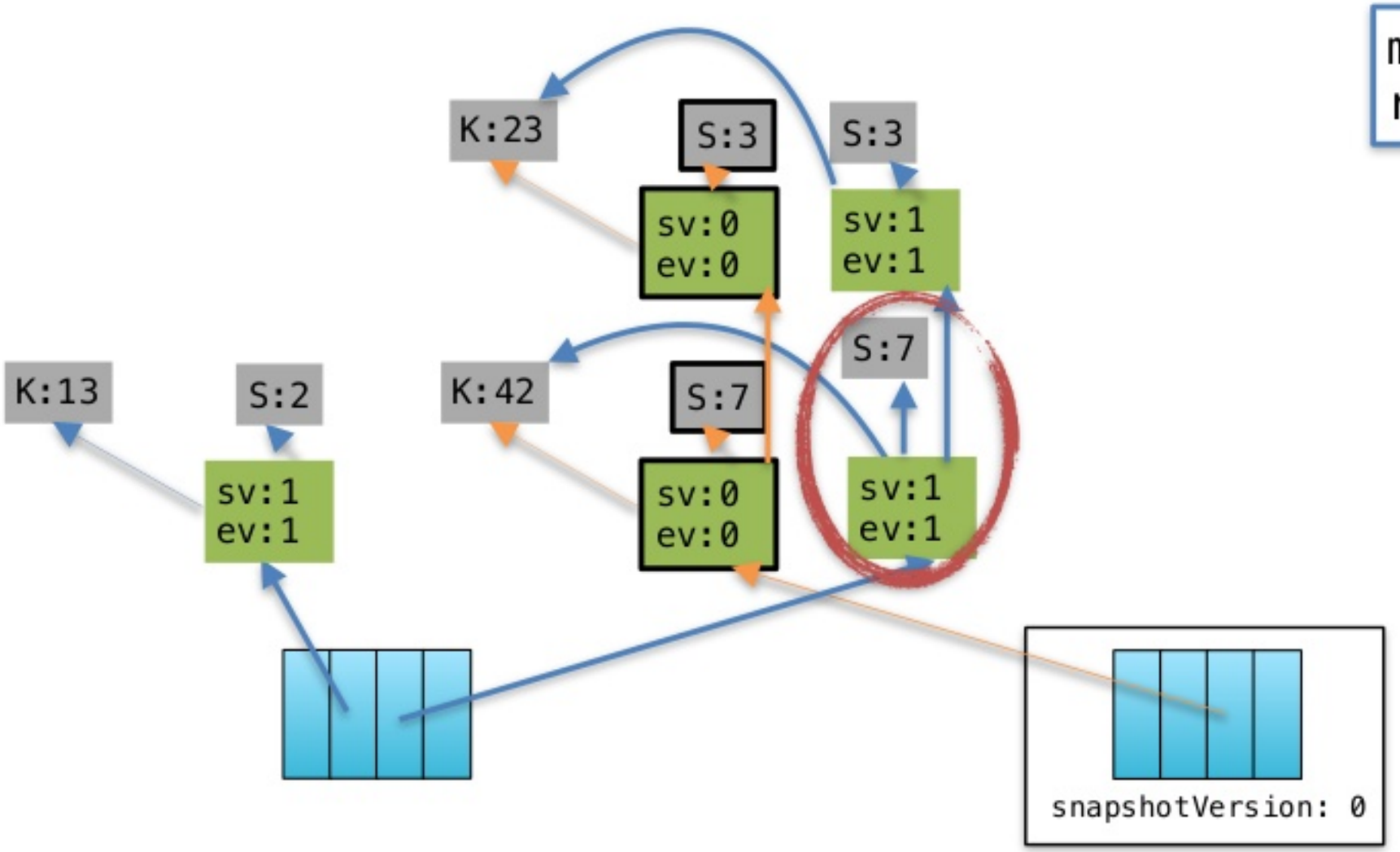
Copy-on-Write Hash Map - Example



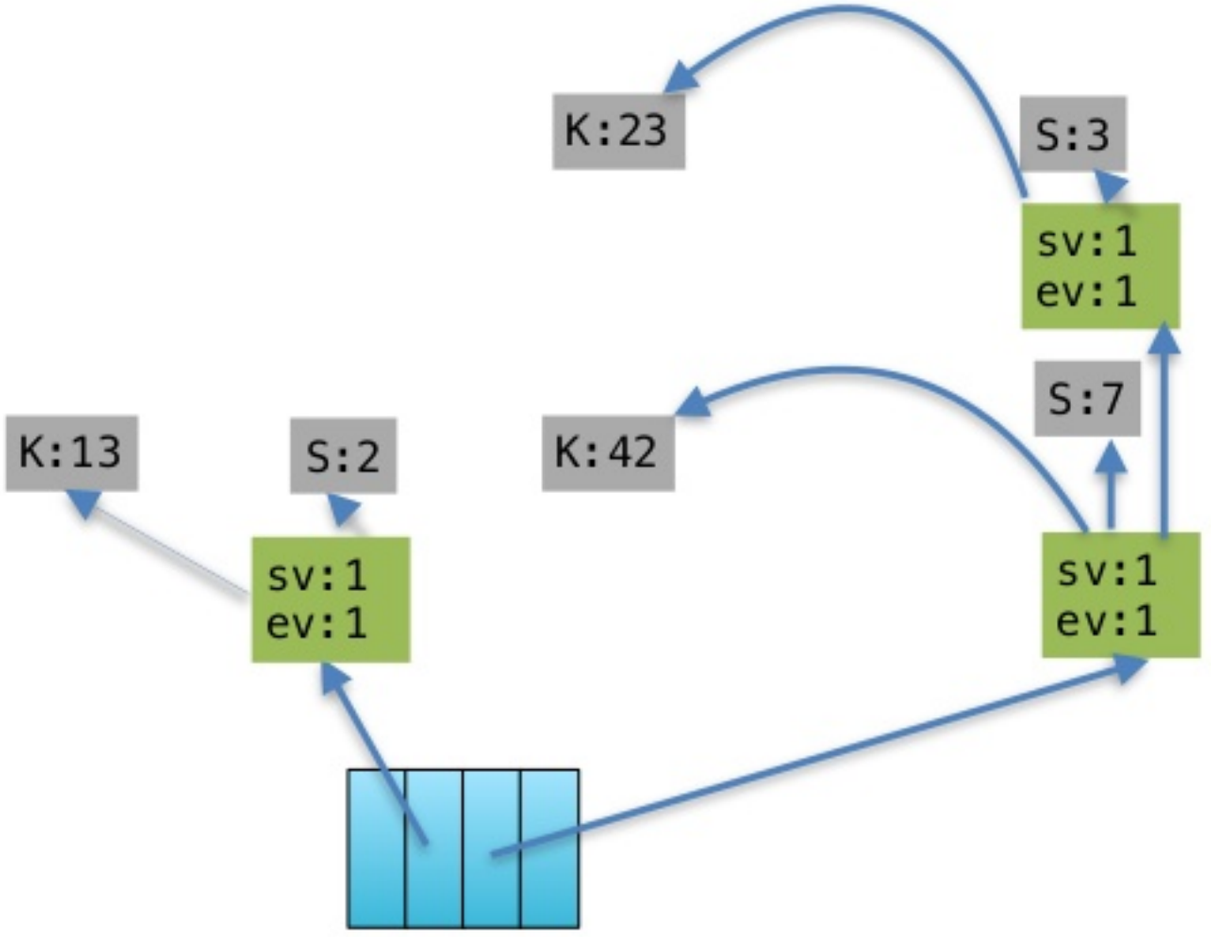
Copy-on-Write Hash Map - Example



Copy-on-Write Hash Map - Example



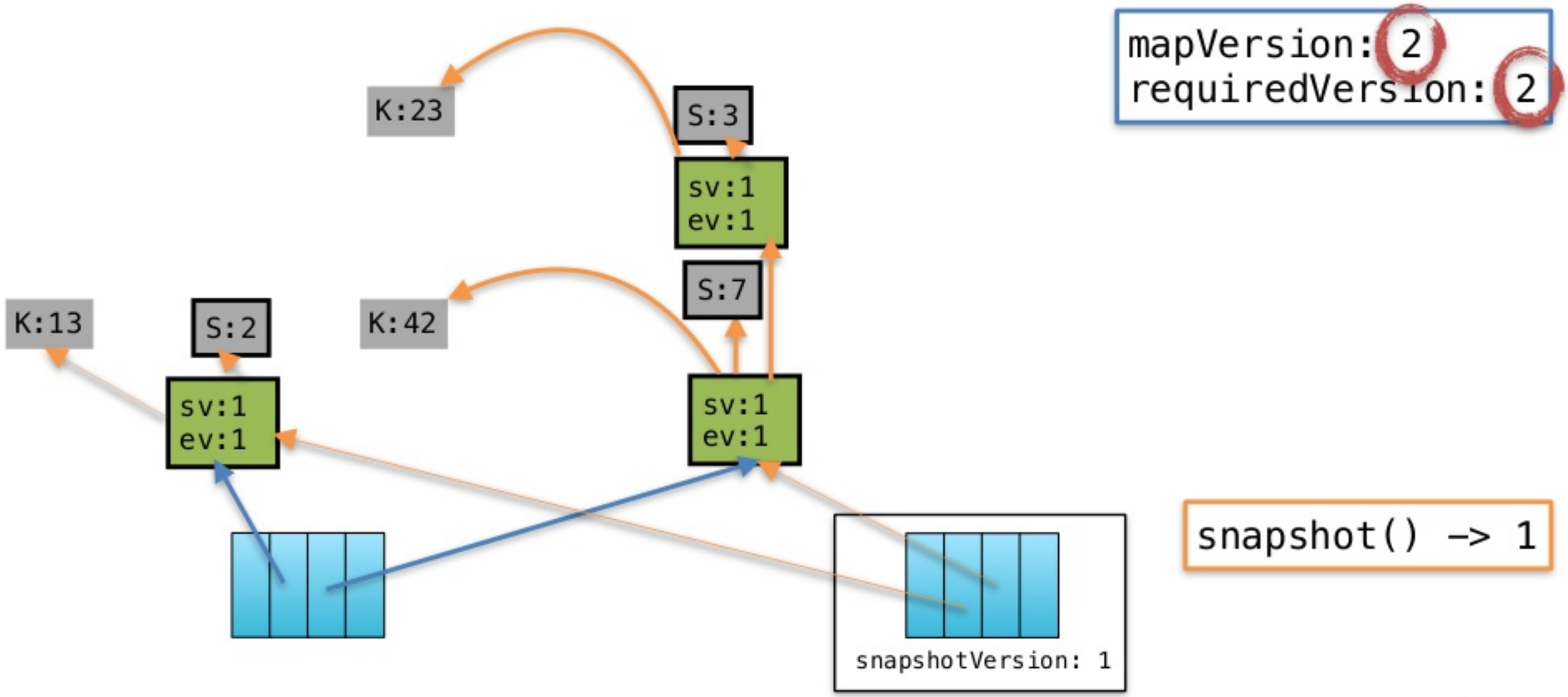
Copy-on-Write Hash Map - Example



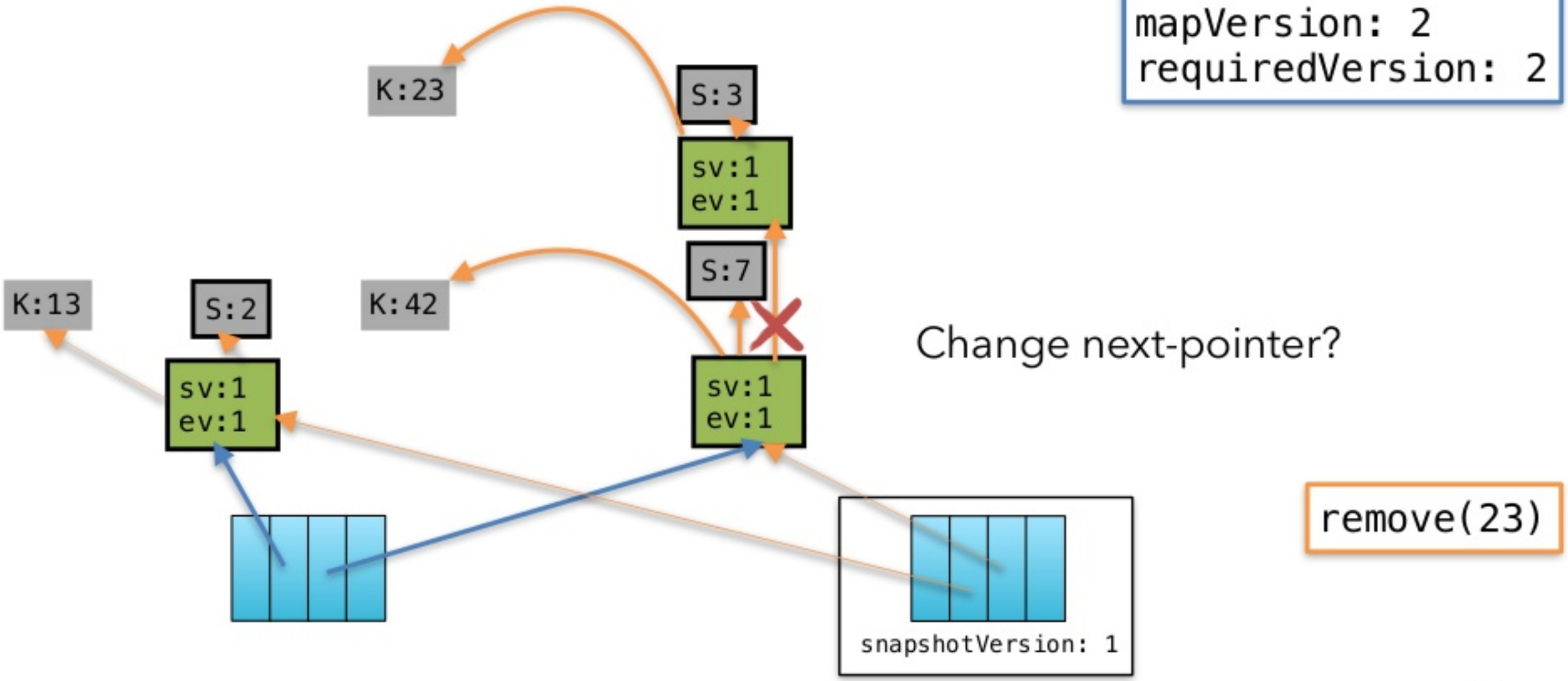

```
mapVersion: 1
requiredVersion: 0
```

```
release(0)
```

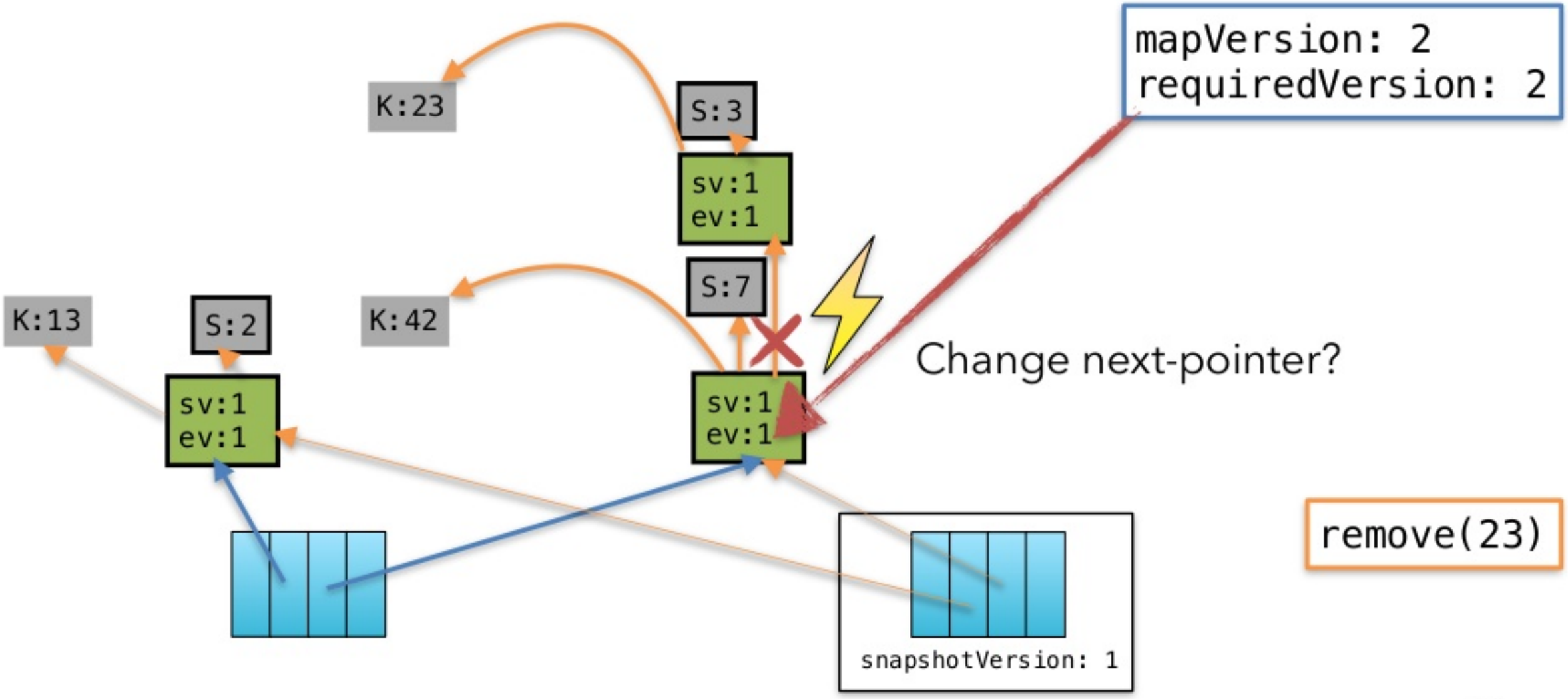
Copy-on-Write Hash Map - Example



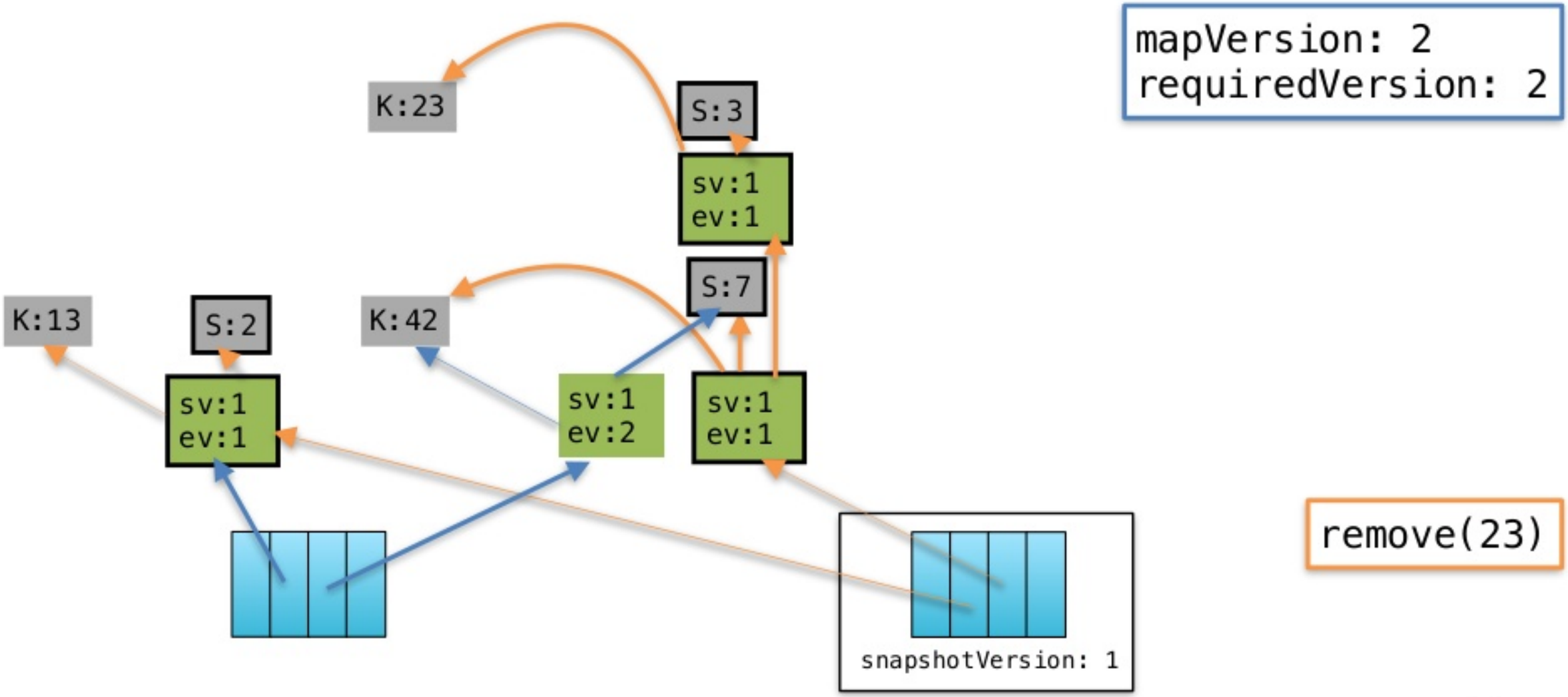
Copy-on-Write Hash Map - Example



Copy-on-Write Hash Map - Example

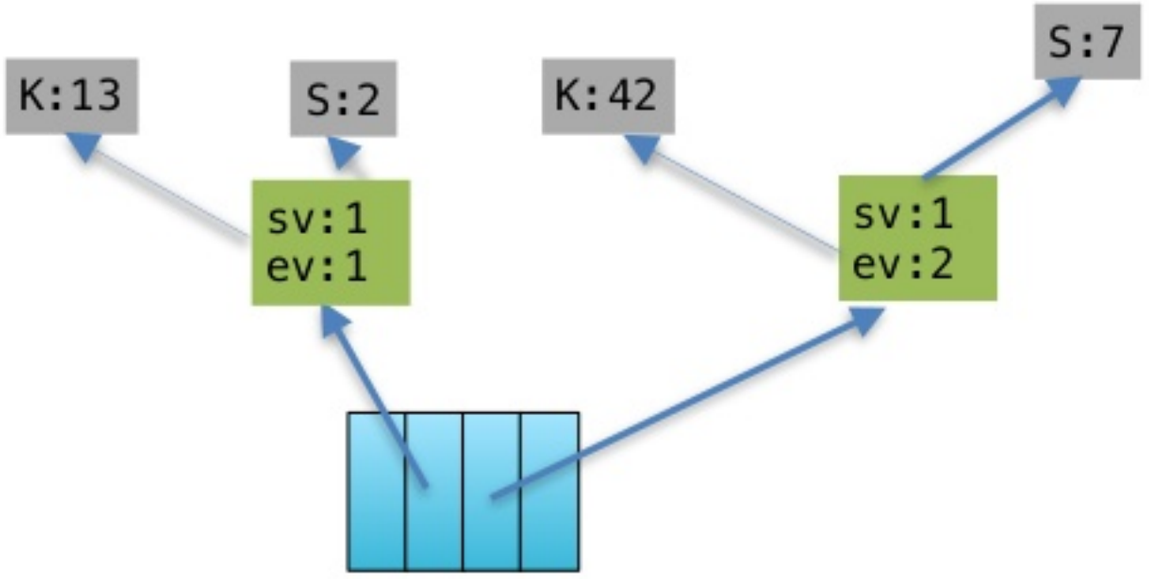


Copy-on-Write Hash Map - Example



Copy-on-Write Hash Map - Example

```
mapVersion: 2
requiredVersion: 0
```




```
release(1)
```

Copy-on-Write Hash Map Benefits

- Fine granular, lazy copy-on-write.
- At most one copy per object, per snapshot.
- No thread synchronisation between readers and writer after array-copy step.
- Supports multiple concurrent snapshots.
- Also implements efficient incremental rehashing.
- Future: Could serve as basis for incremental snapshots.

Questions?