CS 591 K1:
Data Stream Processing and Analytics

Spring 2021

High-availability & reconfiguration

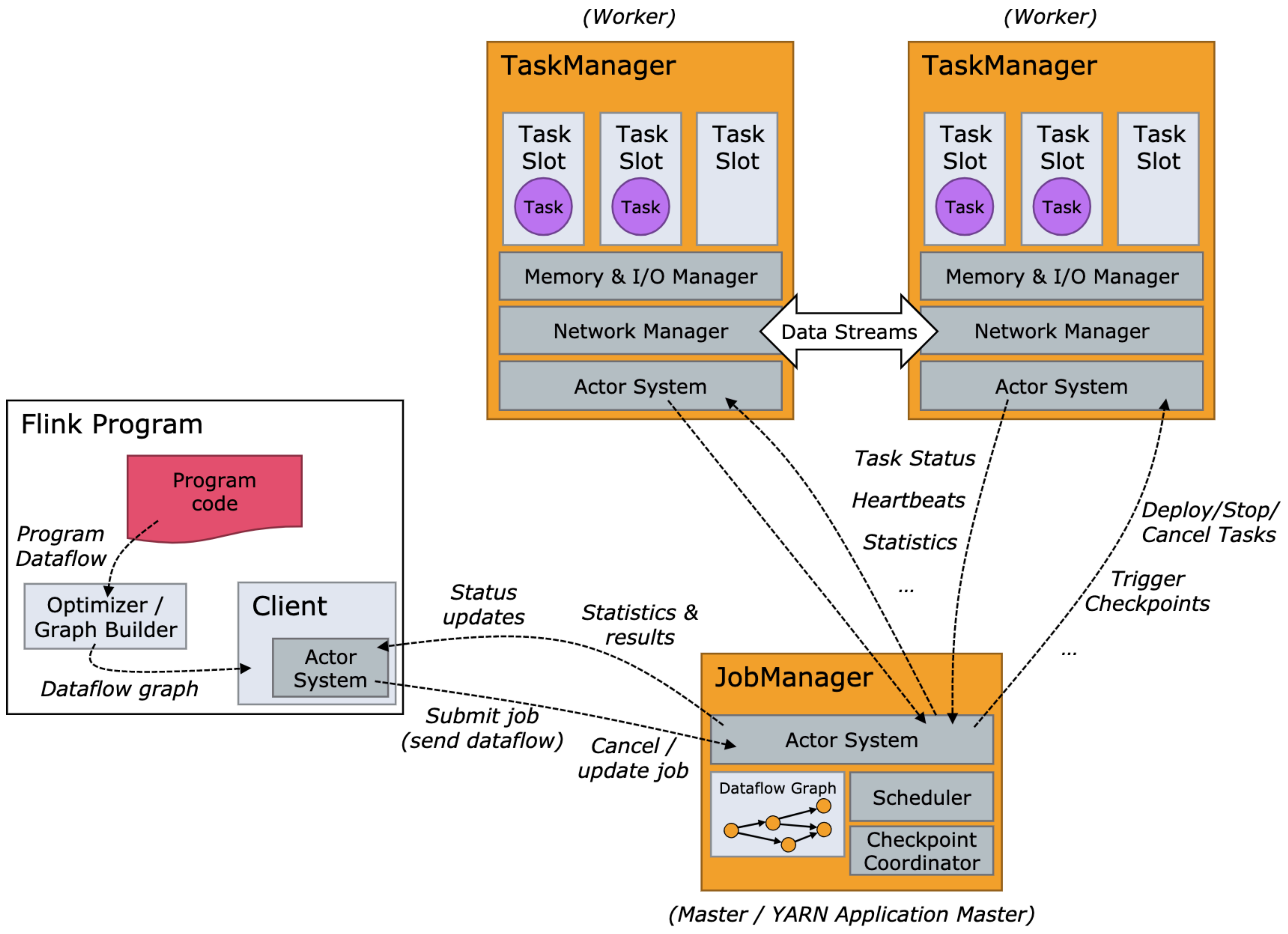
Vasiliki (Vasia) Kalavri
vkalavri@bu.edu

High-availability

Checkpointing guards the state from failures,
but what about process failure?

- To recover from failures, the system needs to
  - restart failed processes
  - restart the application and recover its state

Flink processes



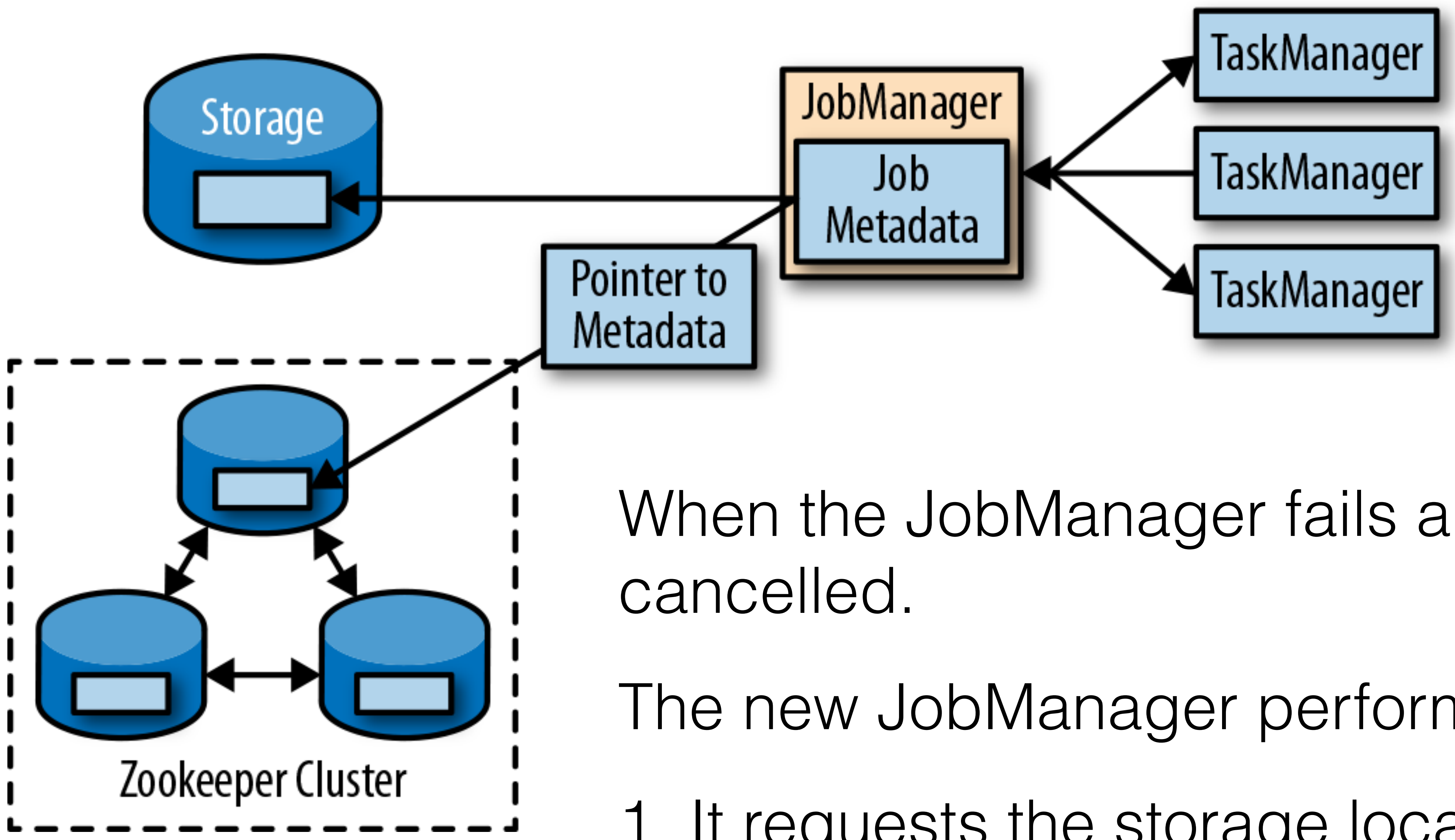
TaskManager failures

- Flink requires a sufficient number of processing slots in order to execute all tasks of an application.
- The JobManager cannot restart the application until enough slots become available.
  - Restart is automatic if there is a ResourceManager, e.g. in a YARN setup
  - A manual TaskManager re-start or a backup is required in standalone mode
- The **restart strategy** determines how often the JobManager tries to restart the application and how long it waits between restart attempts.

JobManager failures

- The JobManager is **a single point of failure** Flink applications
  - It keeps metadata about application execution, such as pointers to completed checkpoints.
- A high-availability mode migrates the responsibility and metadata for a job to another JobManager in case the original JobManager disappears.
- Flink relies on **Apache ZooKeeper** for high-availability
  - coordination and consensus services, e.g. leader election
- The JobManager writes the JobGraph and all required metadata, such as the application's JAR file, into a remote persistent storage system
- Zookeeper also holds state handles and checkpoint locations

Highly available Flink setup



When the JobManager fails all tasks are automatically cancelled.

The new JobManager performs the following steps:

1. It requests the storage locations from ZooKeeper to fetch the JobGraph, the JAR file, and the state handles of the last checkpoint from remote storage.

2. It requests processing slots.

3. It restarts the application and resets the state of all its tasks to the last completed checkpoint.

Restart strategies

To avoid repeating failures, Flink supports the following restart strategies:

- The **fixed-delay** strategy restarts an application a fixed number of times and waits a configured time between two restart attempts.
- The **failure-rate** strategy restarts an application as long as a configurable failure rate is not exceeded. The failure rate is specified as the maximum number of failures within a time interval.
  - e.g. you can configure that an application be restarted as long as it did not fail more than three times in the last ten minutes.
- The **no-restart** strategy does not restart an application, but fails it immediately.

Reconfiguration with Savepoints

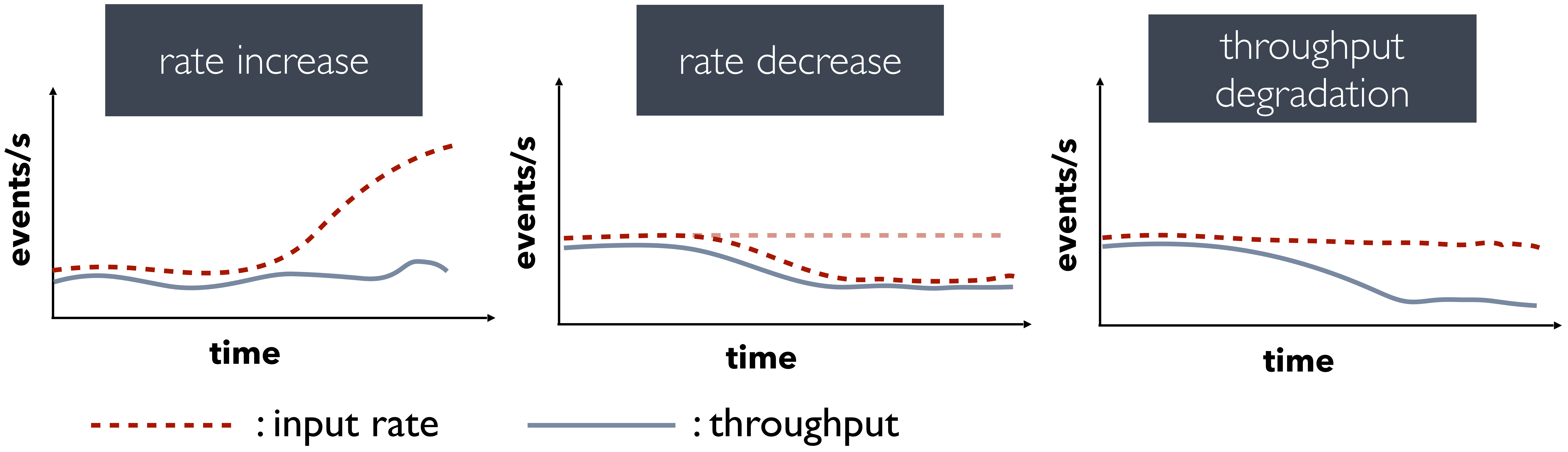
Reconfiguration cases

- Change parallelism
  - scale out to process increased load
  - scale in to save resources
- Fix bugs or change business logic
- Optimize execution plan
- Change operator placement
  - skew and straggler mitigation
- Migrate to a different cluster or software version

Why is it necessary?

Streaming applications are long-running

- Workload will change
- Conditions might change
- State is accumulated over time



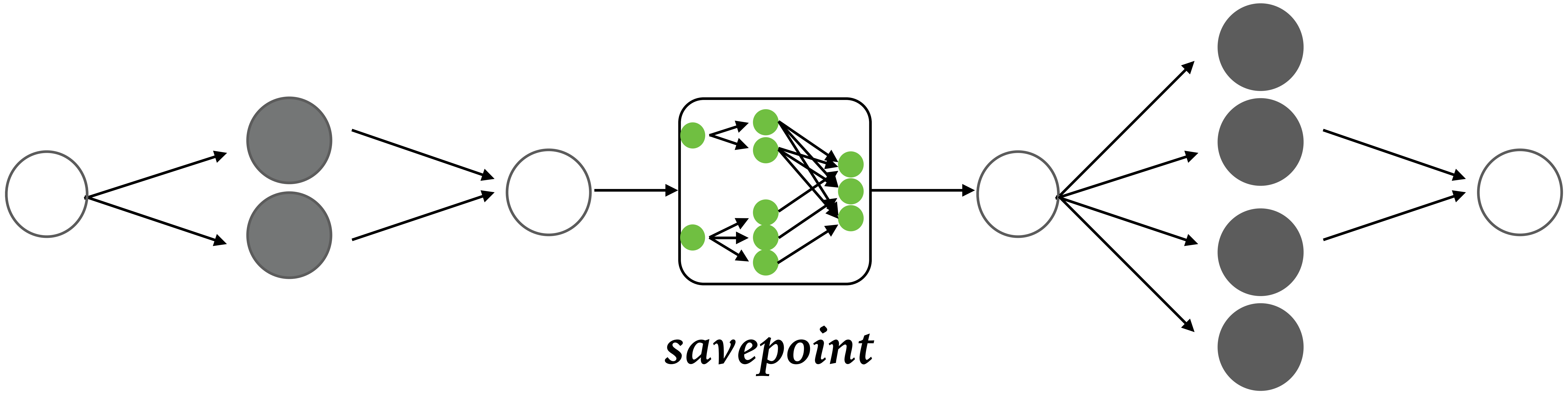
Challenges of reconfiguration

- Ensure result correctness
    - reconfiguration mechanism often relies on fault-tolerance mechanism
- State re-partitioning and migration
    - minimize communication
    - keep duration short
    - minimize performance disruption, e.g. latency spikes
    - avoid introducing load imbalance
- Resource management
    - utilization, isolation
- Automation
    - continuous monitoring
    - bottleneck detection
    - stability, accuracy

Reconfiguring Flink applications

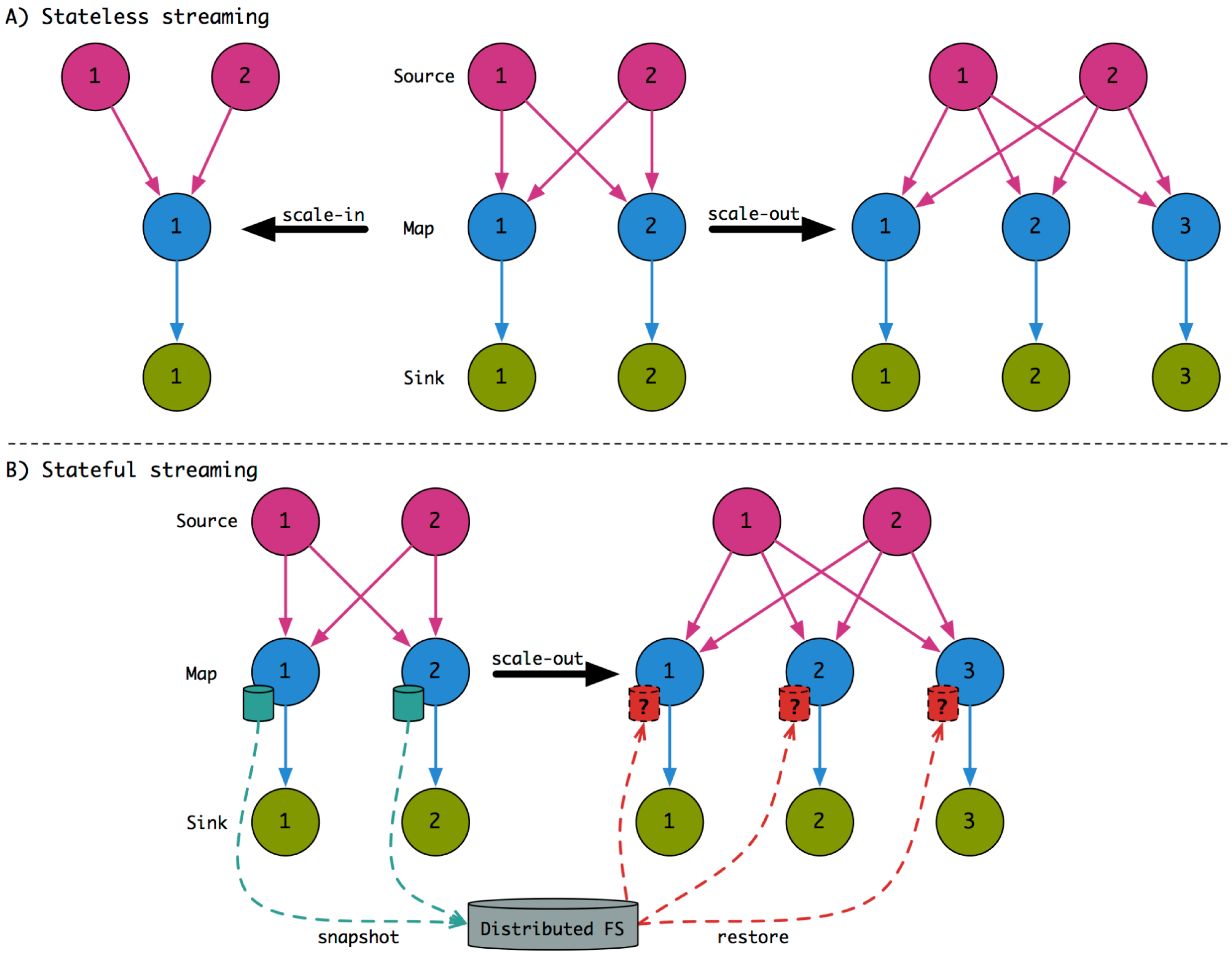
Savepoints: user-triggered checkpoints

- A consistent and complete snapshot of an application's state
  - Checkpoints are automatically created and removed by Flink.
  - Savepoints are never automatically removed.



Scaling from a Savepoint

- To decrease or increase the parallelism of an application:
  - Take a savepoint
  - Cancel the application
  - Restart it with an adjusted parallelism
- The state is automatically *redistributed* to the new set of parallel tasks
- For **exactly-once results**, we need to prevent a checkpoint to complete after the savepoint!
  - Use the integrated savepoint-and-cancel command

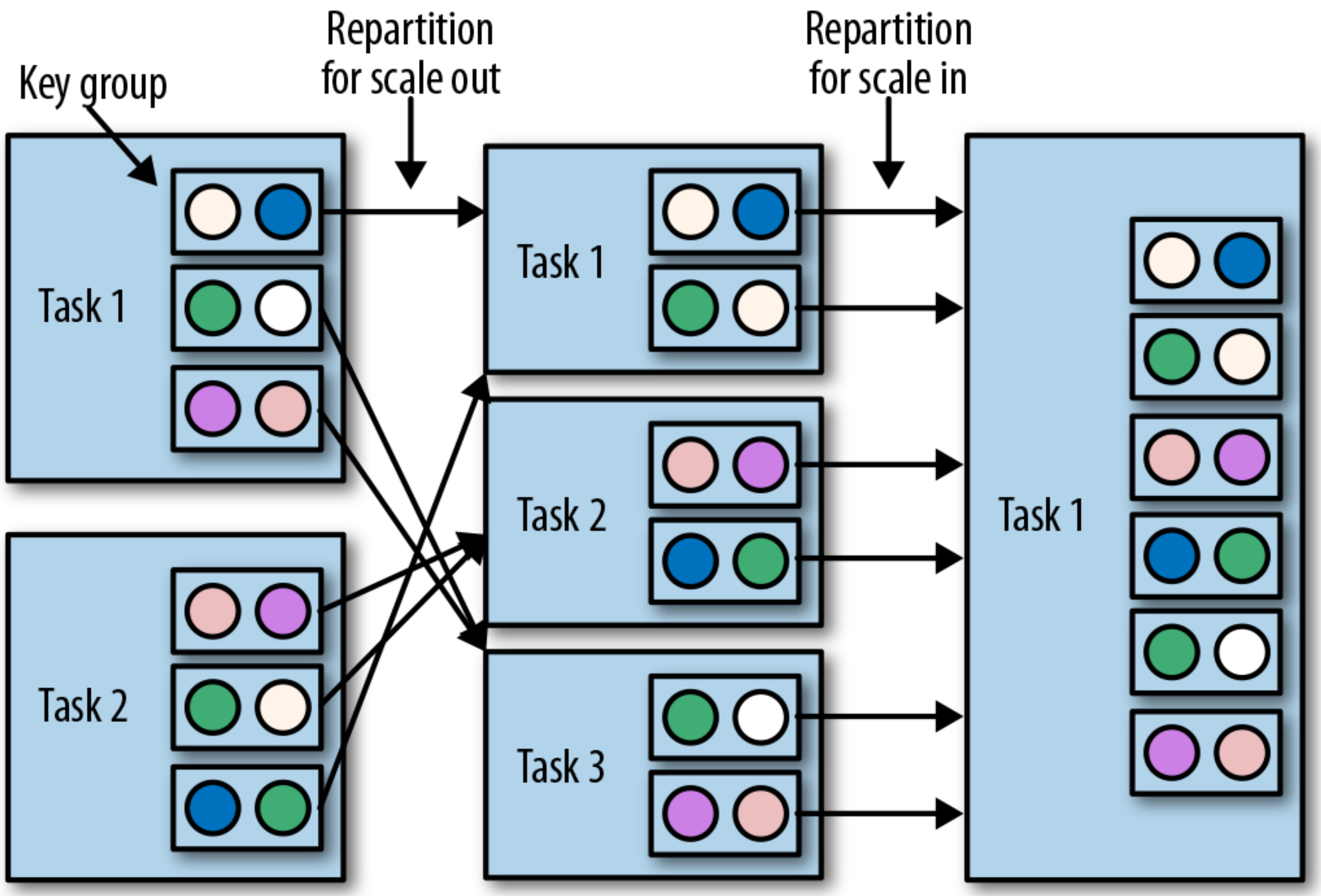


A) Stateless streaming
Source
Map
Sink
scale-in
scale-out
1
2
3
B) Stateful streaming
Source
Map
Sink
scale-out
?
Distributed FS
snapshot
restore

Scaling stateful operators

- When scaling stateful operators, state needs to be repartitioned and assigned to more or fewer parallel tasks
- Scaling different types of state
    - Operators with **keyed state** are scaled by repartitioning keys
    - Operators with **operator list state** are scaled by redistributing the list entries.
    - Operators with **operator broadcast state** are scaled up by copying the state to new tasks.

Scaling keyed state



State re-distribution

Naive approaches

- Read all the previous subtask state from the checkpoint in all sub-tasks and filter out the matching keys for each sub-task
  - Sequential read pattern
  - Tasks read unnecessary data and the distributed file system receives high load of read requests
- Track the state location for each key in the checkpoint, so that tasks locate and read the matching keys only
  - Avoids reading irrelevant data
  - Requires a materialized index for all keys, i.e. a key-to-read-offset mapping, which can potentially grow very large
  - Large amount of random I/O

Reconfiguring keyed stateful operators requires **preserving the key semantics**:

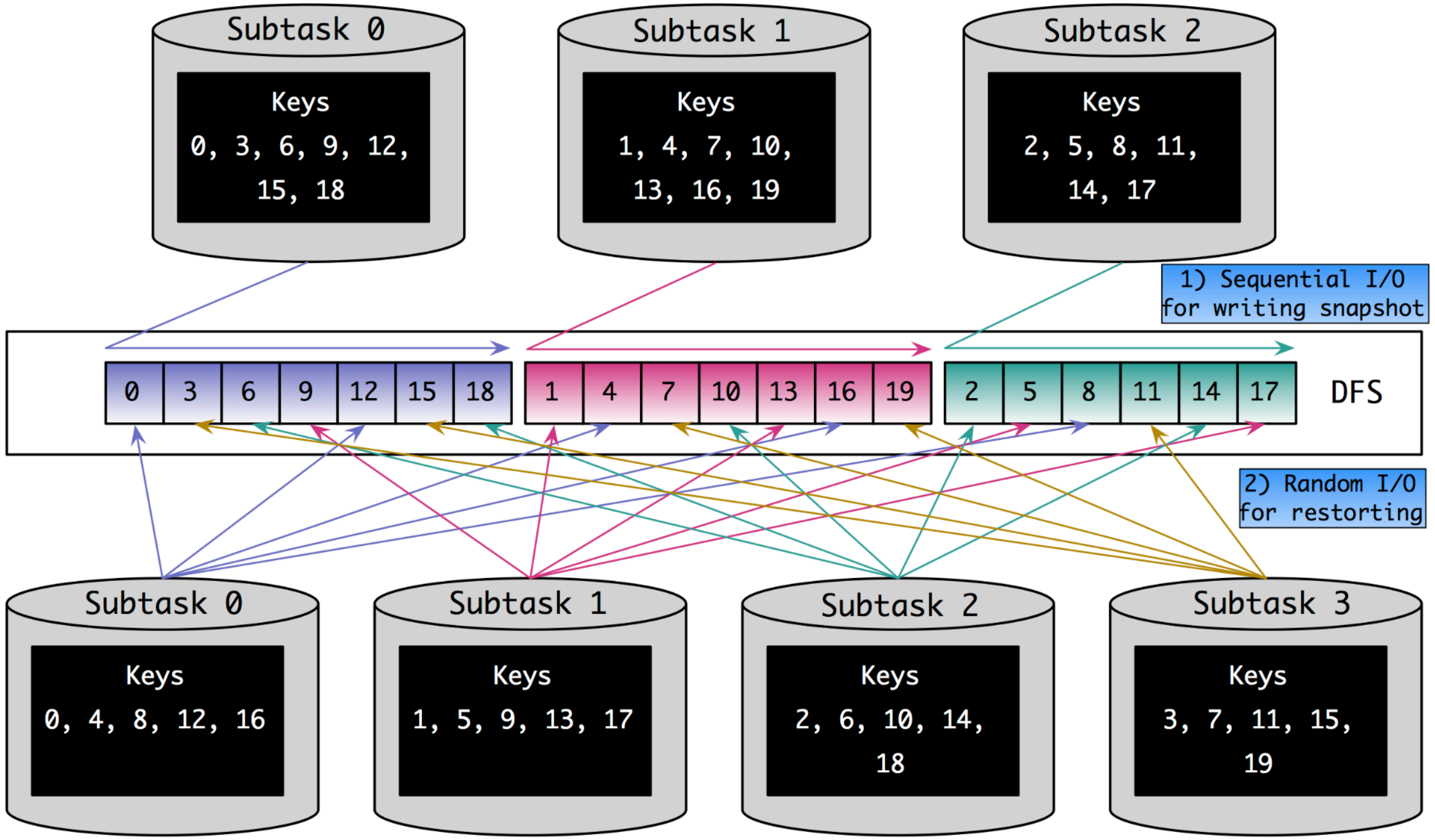
- Existing state for a particular key and all future events with this key must be routed to the same parallel instance
- Some kind of hashing is typically used
- Maintaining routing tables or an index for all key mappings is usually impractical
- Skewed load is challenging to handle with hashing

State redistribution objectives

- Load balance
  - memory: load in terms of maintained state
  - computation: load in terms of computation
  - communication: load in terms of flow size in the input channel of each parallel task
- Partitioning function performance
  - space required to implement routing
  - lookup cost
- Migration performance
  - re-assignment computation cost
  - state movement cost

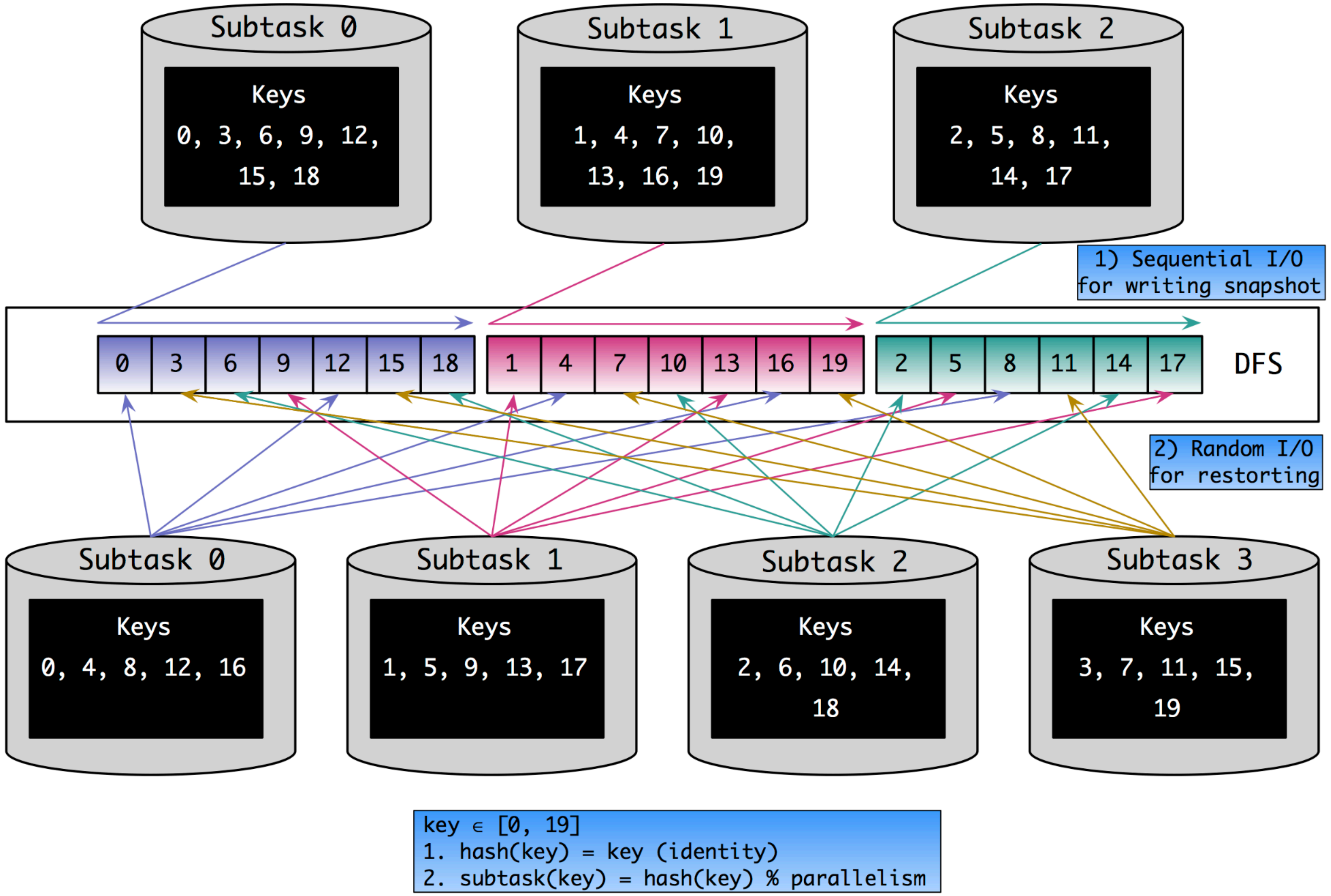
Uniform hashing

- Evenly distributes keys across parallel tasks
- Fast to compute, no routing state
- High migration cost
  - When a new node is added, state is shuffled across existing and new nodes
  - Random I/O and high network communication
- Not suitable for adaptive applications

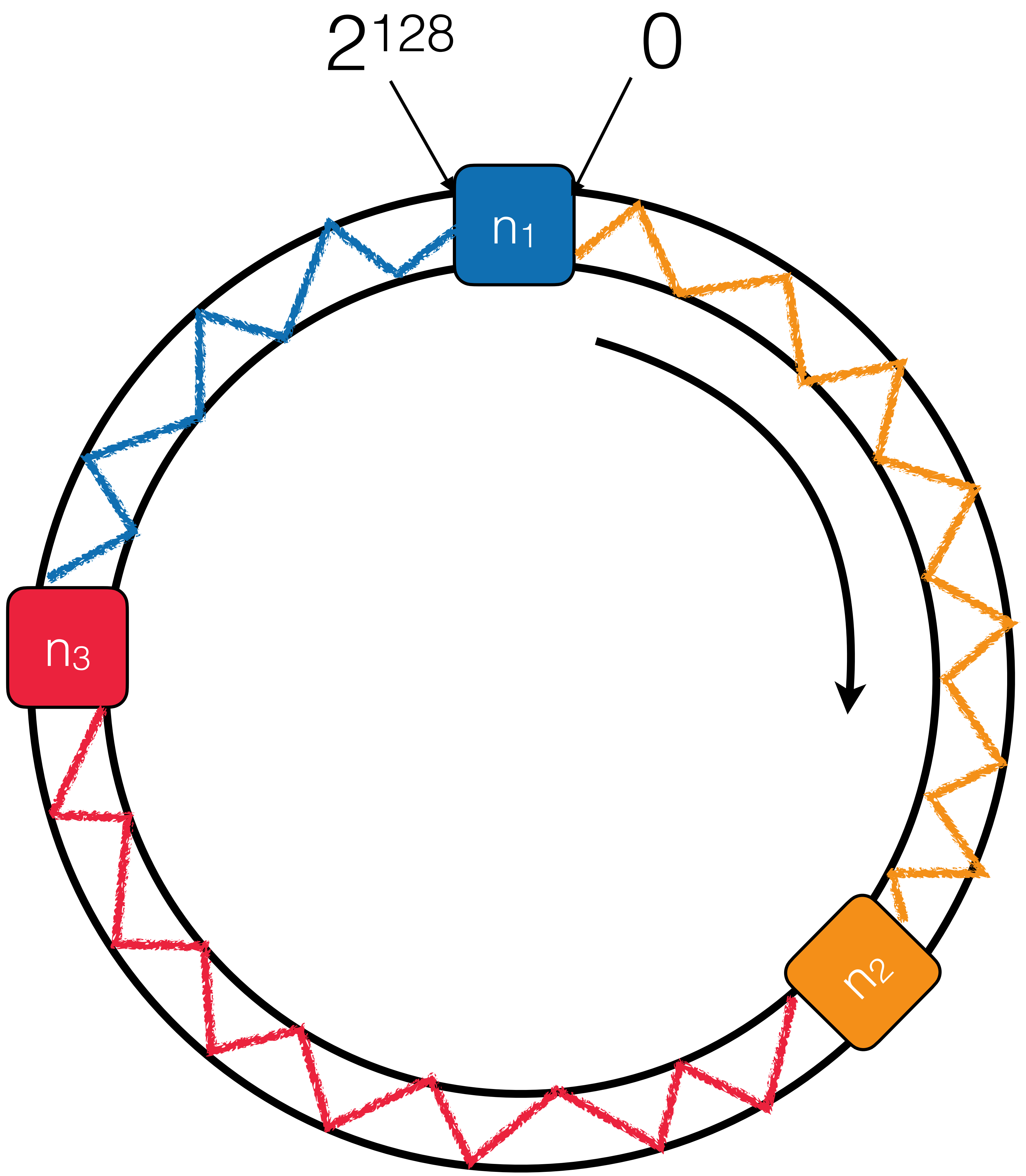


key ∈ [0, 19]
1. hash(key) = key (identity)
2. subtask(key) = hash(key) % parallelism

Can we do better?

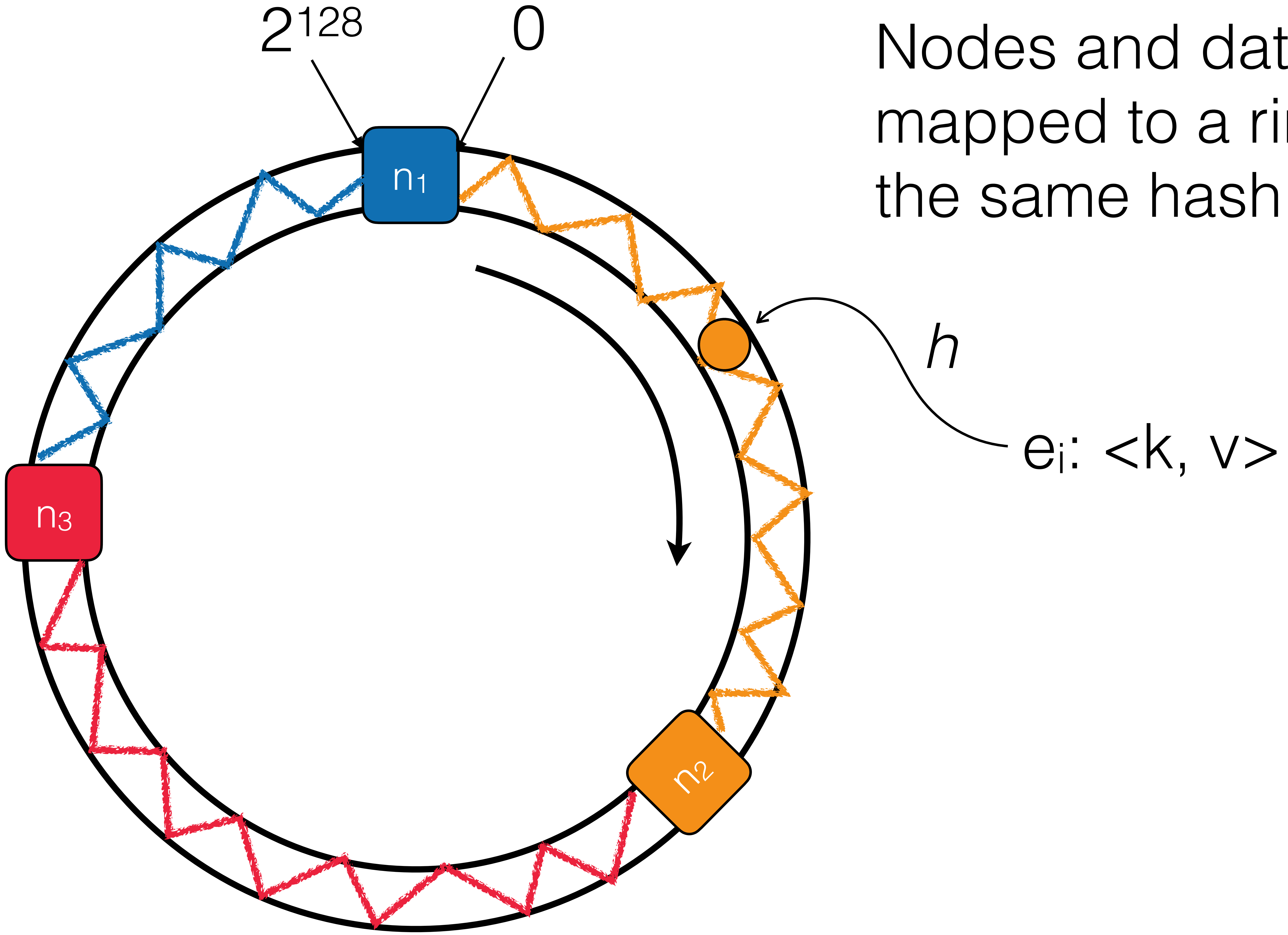


Consistent hashing



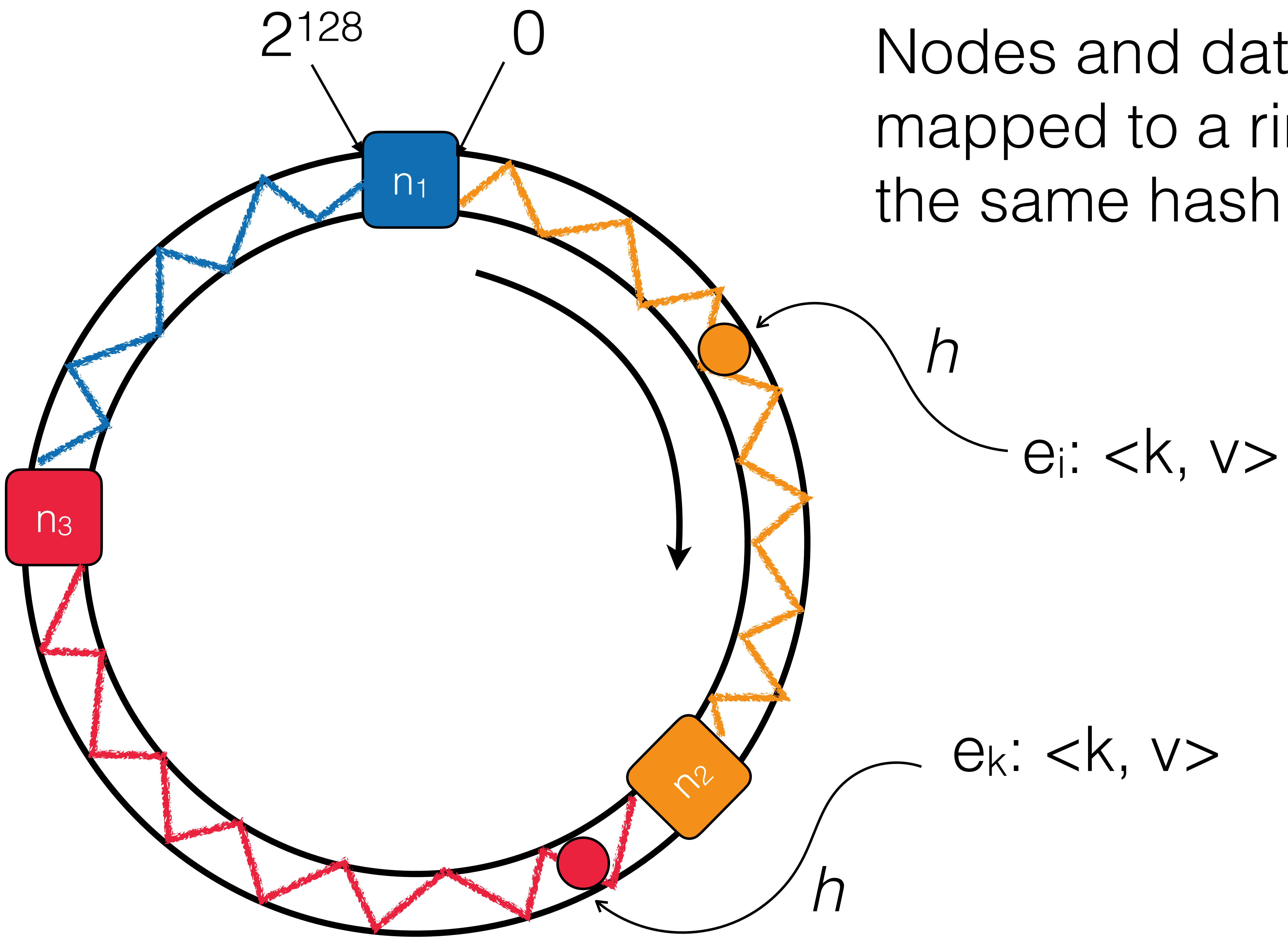
Nodes and data are mapped to a ring using the same hash function.

Consistent hashing



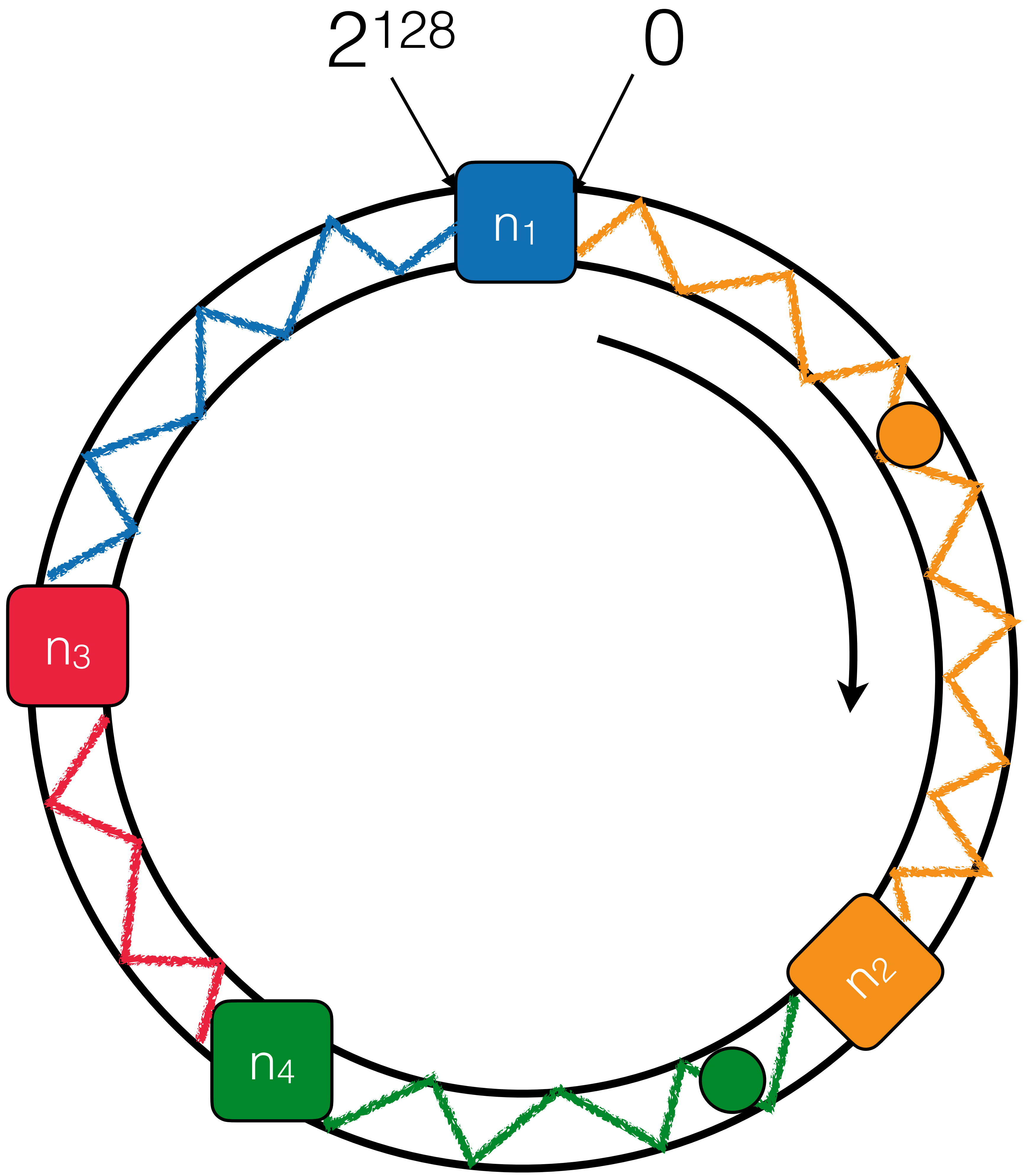
Nodes and data are mapped to a ring using the same hash function.

Consistent hashing



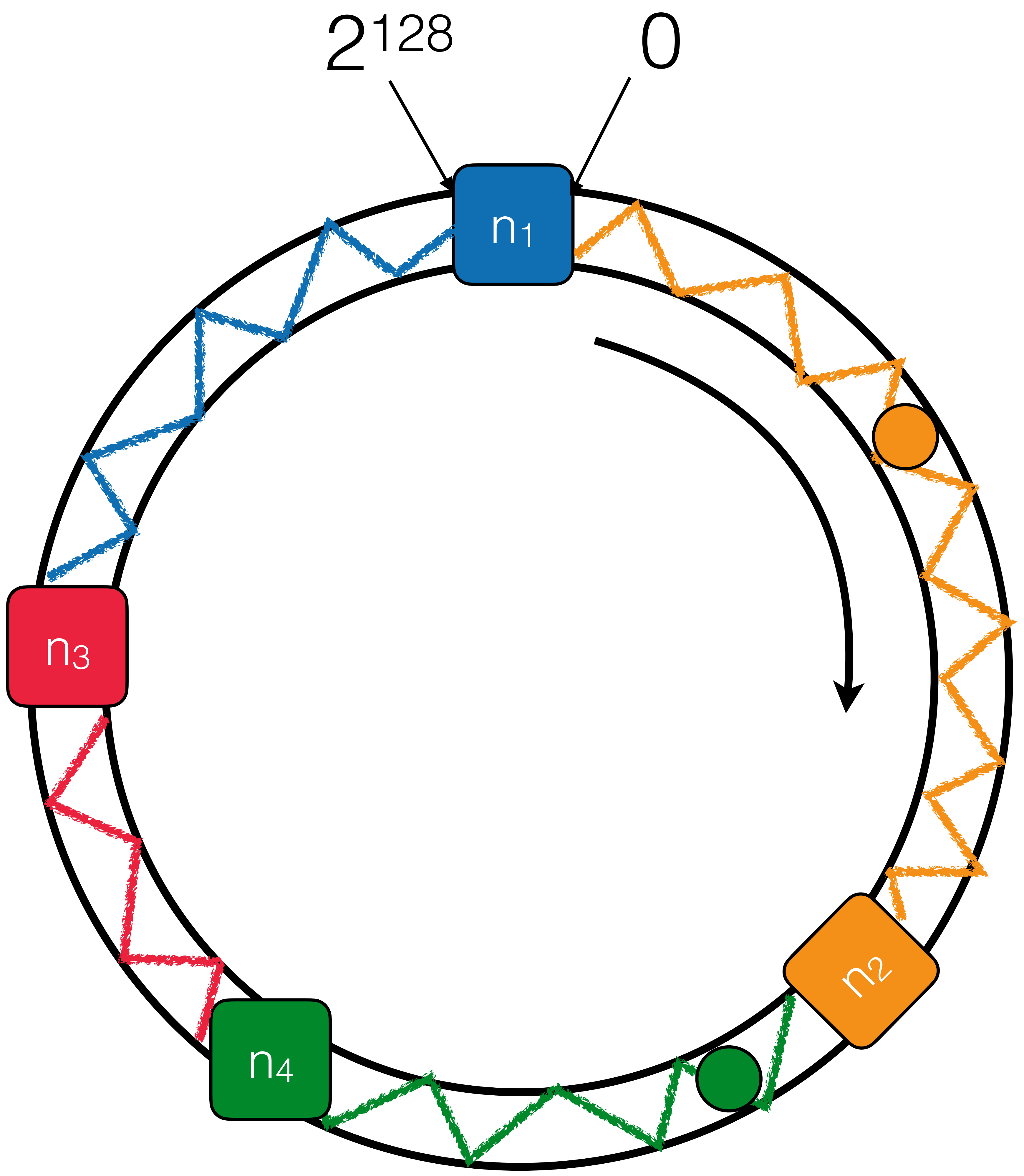
Nodes and data are mapped to a ring using the same hash function.

Consistent hashing



When a new node joins, no data is moved across old nodes.

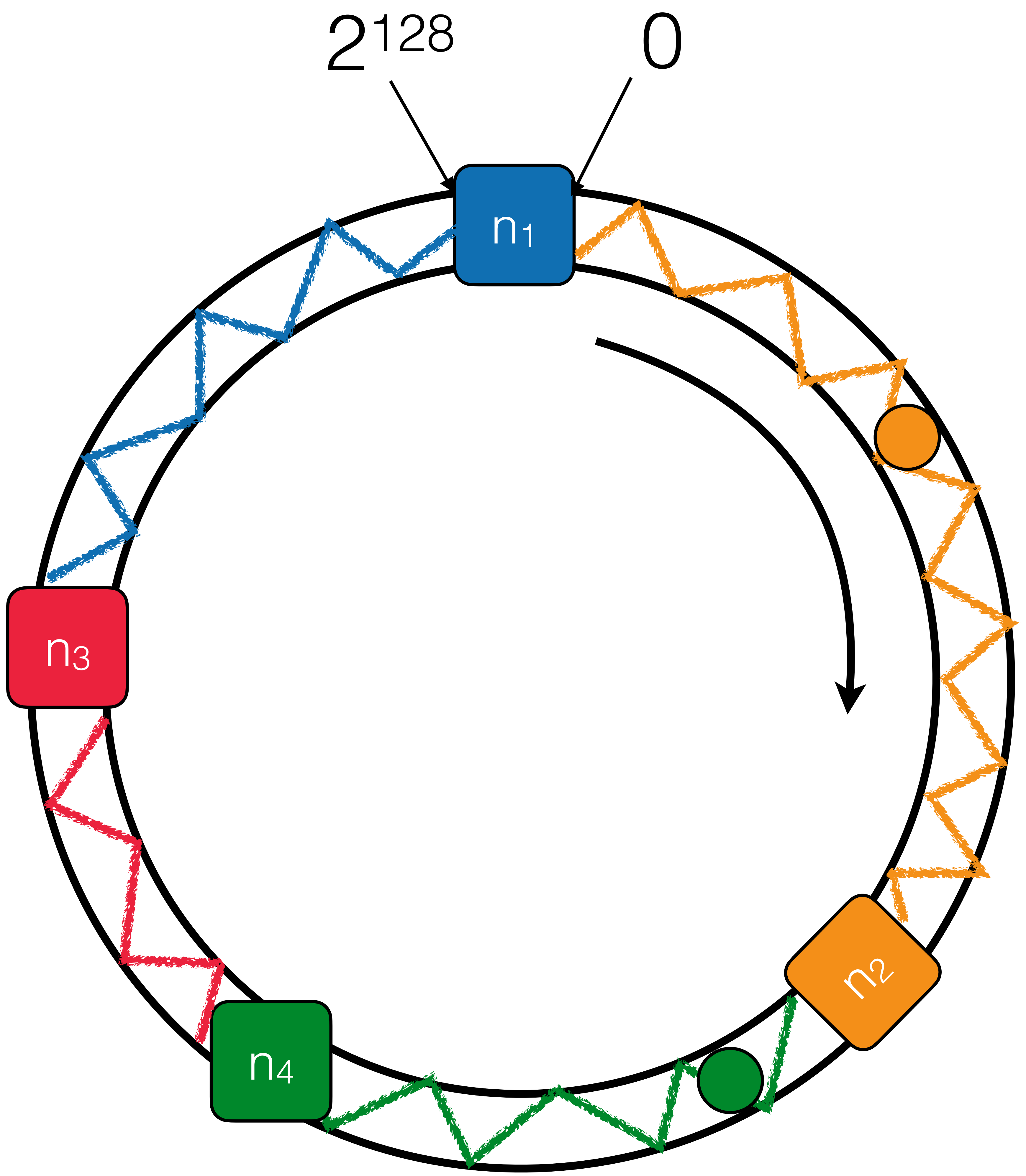
Consistent hashing



When a new node joins, no data is moved across old nodes.

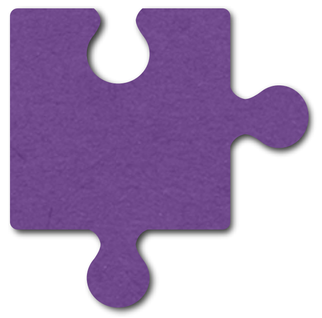
In practice, each node is mapped to multiple points on the ring using multiple hash functions.

Consistent hashing



When a new node joins, no data is moved across old nodes.

In practice, each node is mapped to multiple points on the ring using multiple hash functions.



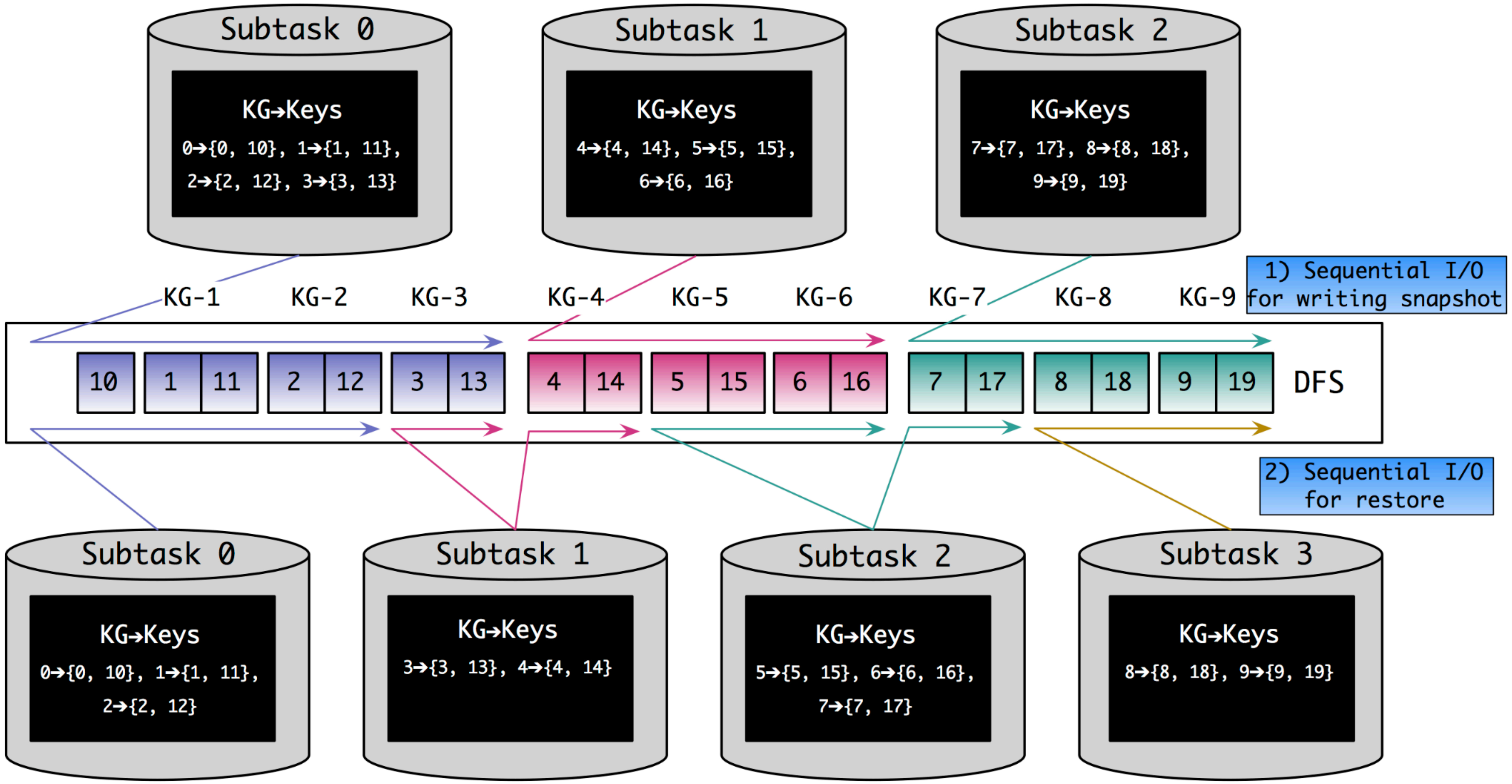
Why?

Consistent hashing

- It ensures state is not moved across nodes that are present before and after the migration
- When a new node joins, it becomes responsible for data items from multiple of the existing nodes
- When a node leaves, its data items are distributed over the existing nodes
- On average *M/N* partitions are moved when the *Nth* node is inserted or removed from a system with *M* partitions

Apache Flink Key-groups

- State is mapped into key-groups
- Key-groups are mapped to subtasks as *ranges*
  - On restore, reads are sequential within each key-group, and often across multiple key-groups
  - The metadata of key-group-to-subtask assignments are small. No need to maintain explicit lists of key-groups, only range boundaries.
- The maximum parallelism parameter of an operator defines the number of key groups into which the keyed state of the operator is split.
  - The number of key groups limits the maximum number of parallel tasks to which keyed state can be scaled.
  - Trade-off between flexibility in rescaling and the maximum overhead involved in indexing and restoring the state



key ∈ [0, 19], number_of_key_groups = 10
1. hash(key) = key (identity)
2. key_group(key) = hash(key) % number_of_key_groups
3. subtask(key) = key_group(key) * parallelism / number_of_key_groups

Setting the max parallelism

```
val env = StreamExecutionEnvironment.getExecutionEnvironment

// set the maximum parallelism for this application
env.setMaxParallelism(512)

val alerts: DataStream[(String, Double, Double)] =
    keyedSensorData
    .flatMap(new TemperatureAlertFunction(1.1))
    // set the maximum parallelism for this operator
    .setMaxParallelism(1024)
```

Lecture references

- A Deep Dive into Rescalable State in Apache Flink: https://flink.apache.org/features/2017/07/04/flink-rescalable-state.html
- Buğra Gedik. **Partitioning functions for stateful data parallelism in stream processing**. (VLDB Journal 23, 4, 2014).