東京大学
THE UNIVERSITY OF TOKYO

SEMem: deployment of MPI-based in-memory storage for Hadoop on supercomputers

Thanh-Chung Dao and Shigeru Chiba

The University of Tokyo, Japan

Running Hadoop on modern supercomputers

- Hadoop assumes every compute node has a local disk drive
- Modern supercomputers do not have local disk drives
  - It only has a central file server using e.g. Lustre
  - For example, K computer, Cray Titan, and IBM Sequoia

From Fujitsu

Why supercomputers do not have local disk drives

- Local disk
  - Not scalable
  - Hard to maintain
  - Physical space is limited
- It cannot be shared among all users
- SSD is available on some supercomputers
  - But data should be erased after a job finishes

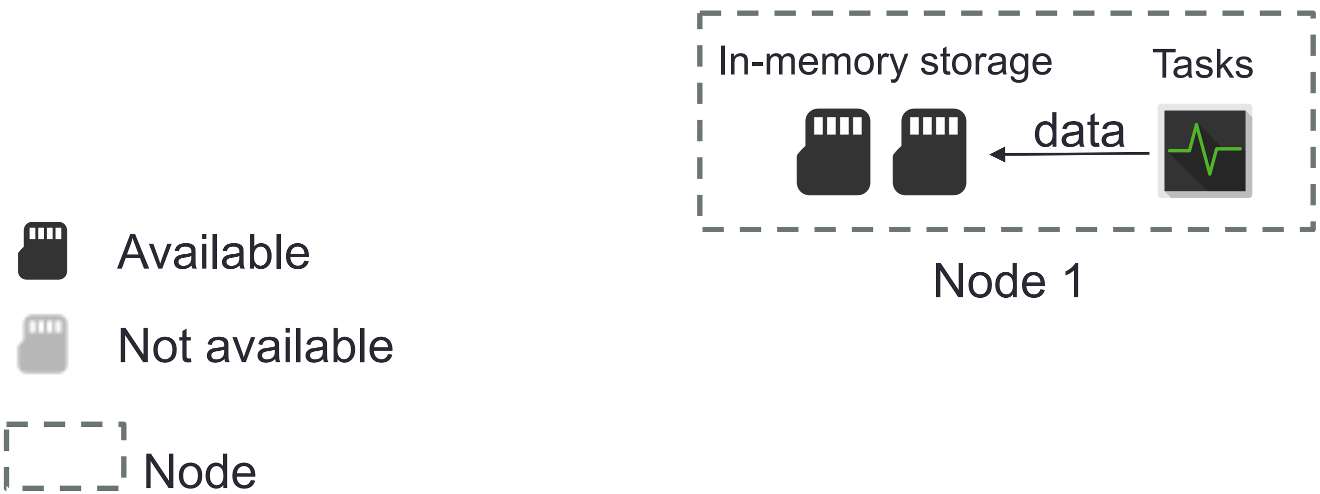
Using in-memory storage to provide efficient virtual local disks

- Research question:
  How to deploy in-memory storage on supercomputers
  - Choose the best deployment strategy in context of MapReduce
  - Using in-memory storage is natural approach to avoid expensive disk I/O to central file server
    - Data is kept in memory
  - Memcached-like separate in-memory server is also an option
    - Typical deployment of Memcached software is installing its daemon on dedicated nodes

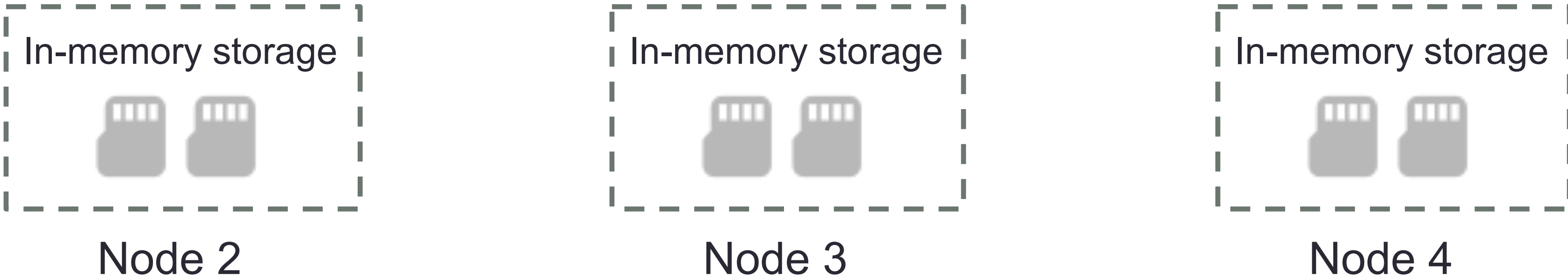
Our approach: SEMem in-memory file system

- Users can choose three deployment strategies
  - RamDisk: data is stored only in local memory
  - Every-node: data can be stored in remote memory
  - Dedicated-node: data is stored on dedicated nodes
- Our in-memory storage, SEMem:
  - Easily configurable to select appropriate deployment strategy
  - Tightly integrated with Hadoop
  - Using MPI communication [Dao & Chiba, CCGRID'16]

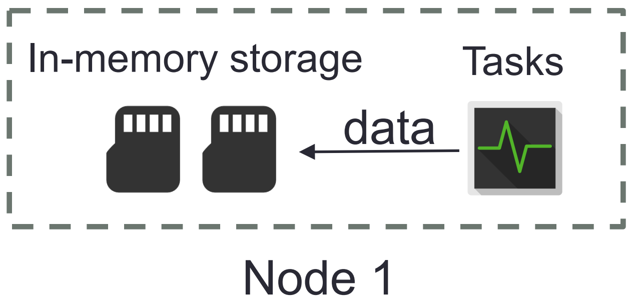
RamDisk: data is stored only in local memory

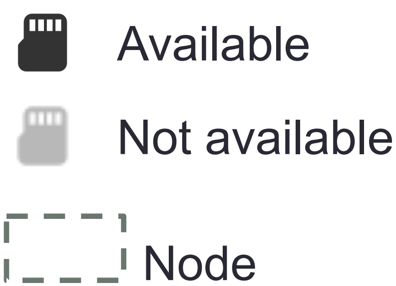


RamDisk: data is stored only in local memory



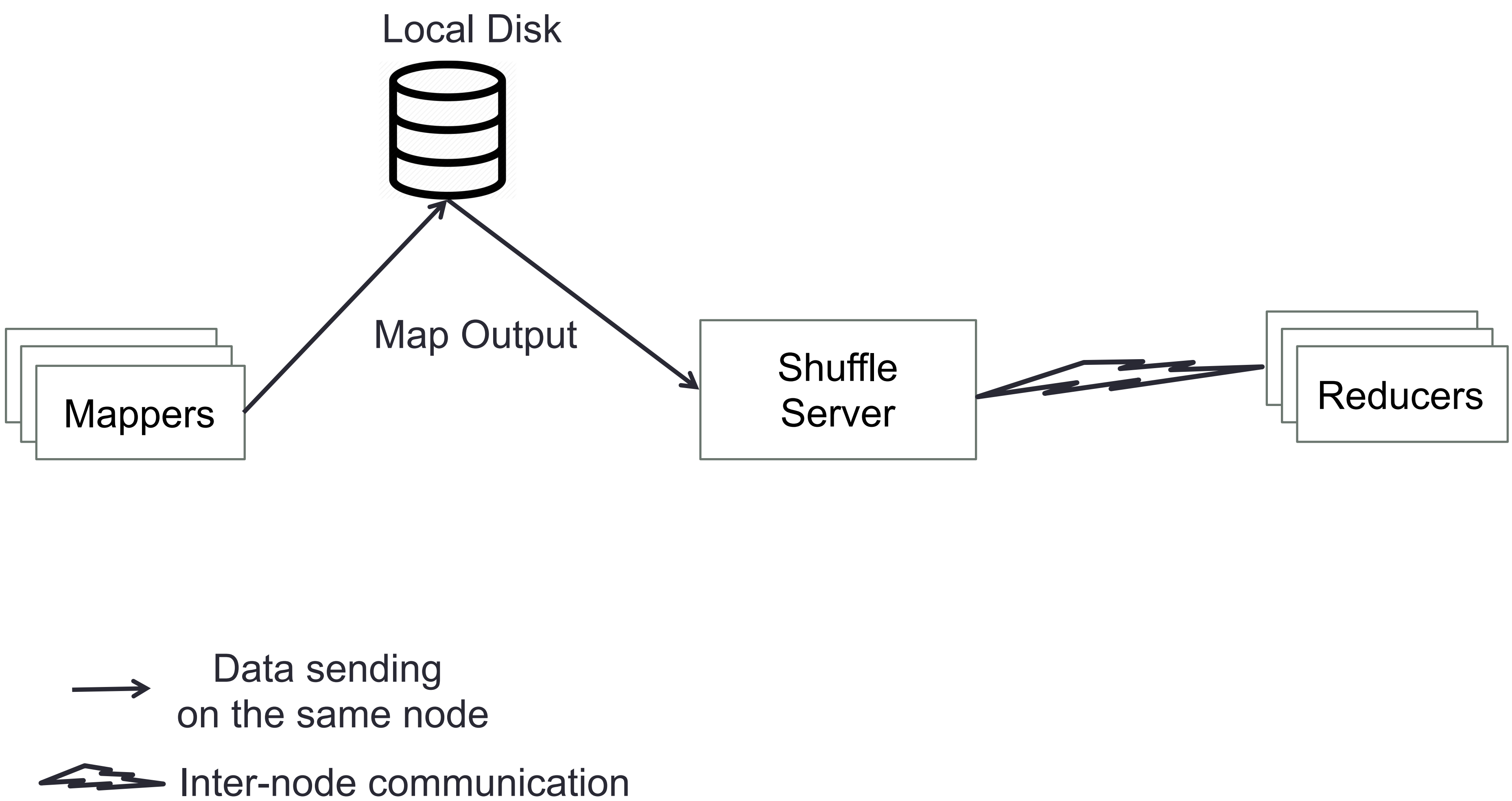
Node 1 cannot use memory of Node 2, 3, & 4



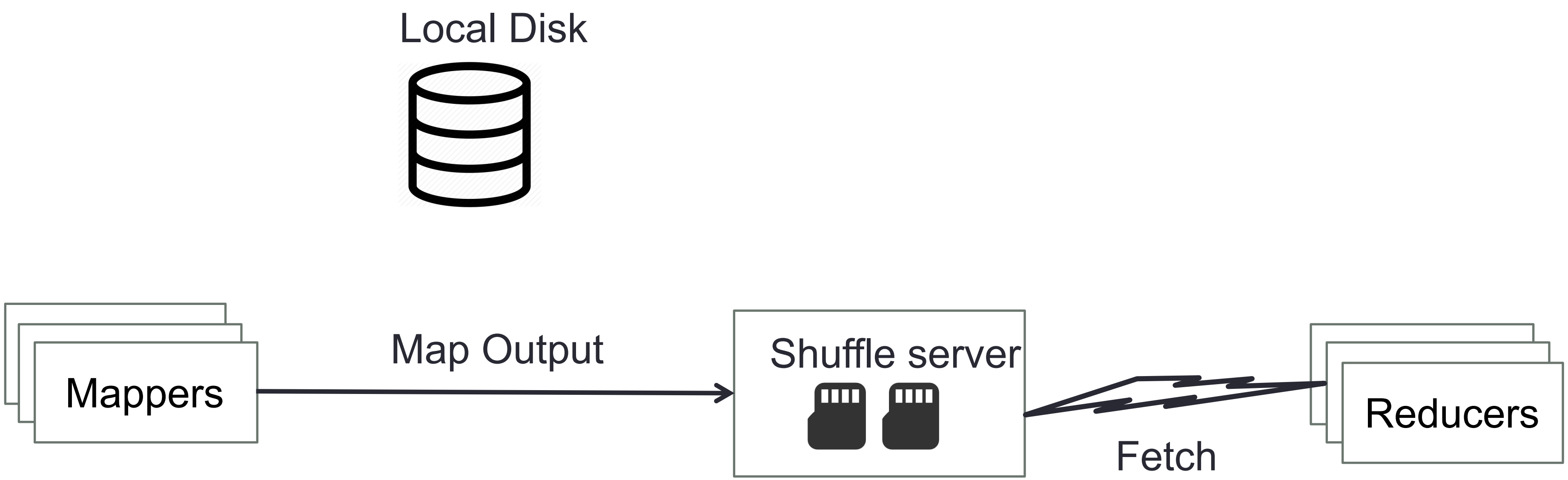


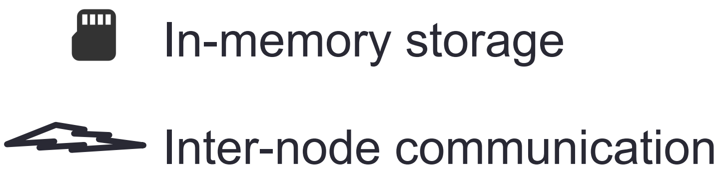
- Out of memory can happen
  - ❖ Since each node has limited amount of memory

The original Hadoop workflow



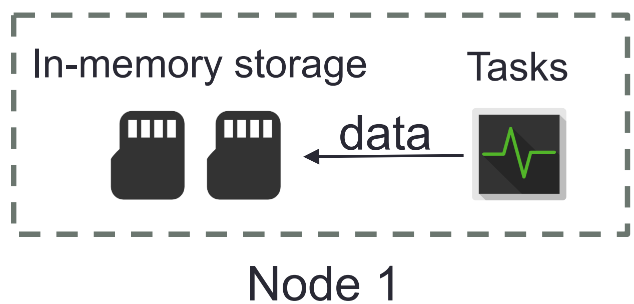
RamDisk deployment on Hadoop workflow

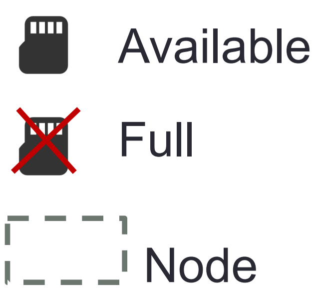




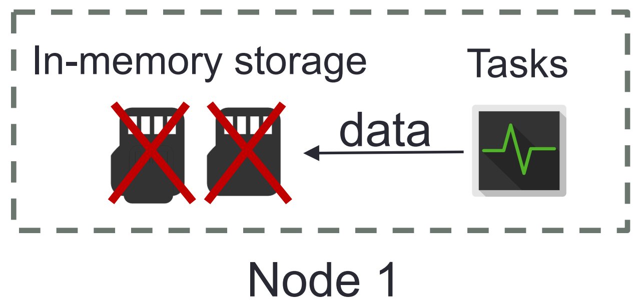
- Mappers are modified to send their output directly to shuffle server
- In-memory storage is set up at shuffle server

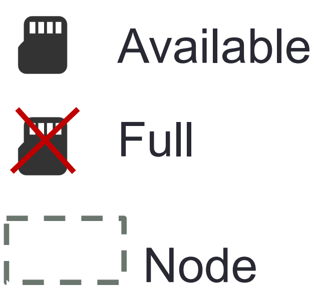
Every-node: deployed on every node and data can be stored in remote memory



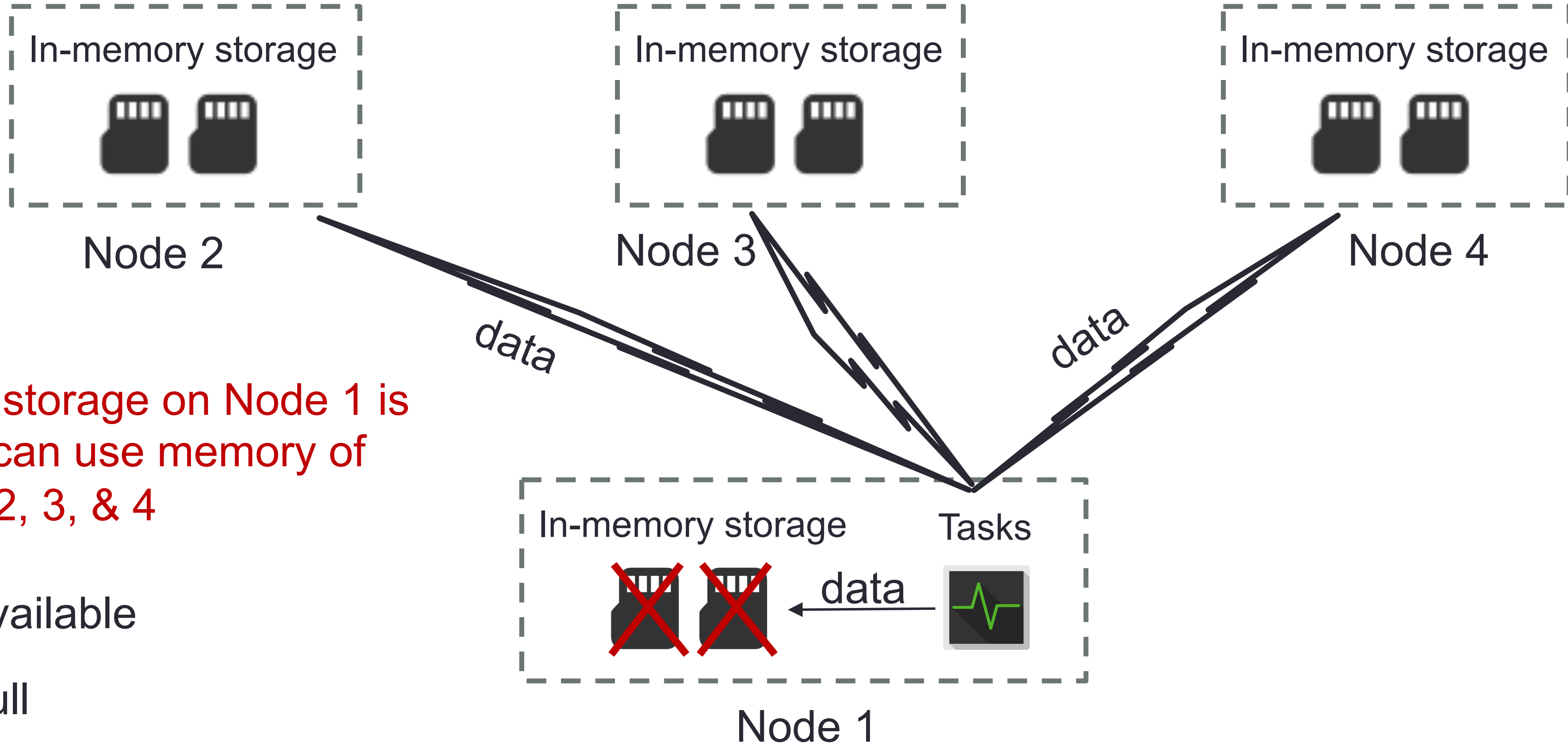


Every-node: deployed on every node and data can be stored in remote memory

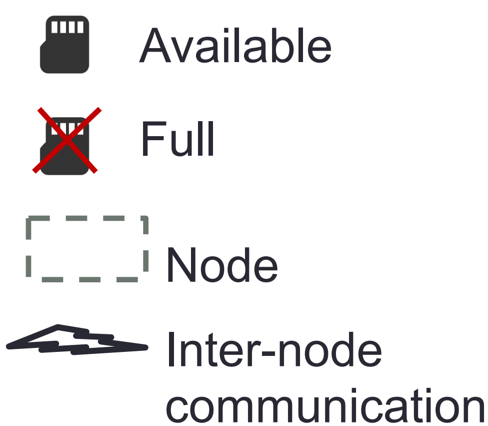




Every-node: deployed on every node and data can be stored in remote memory

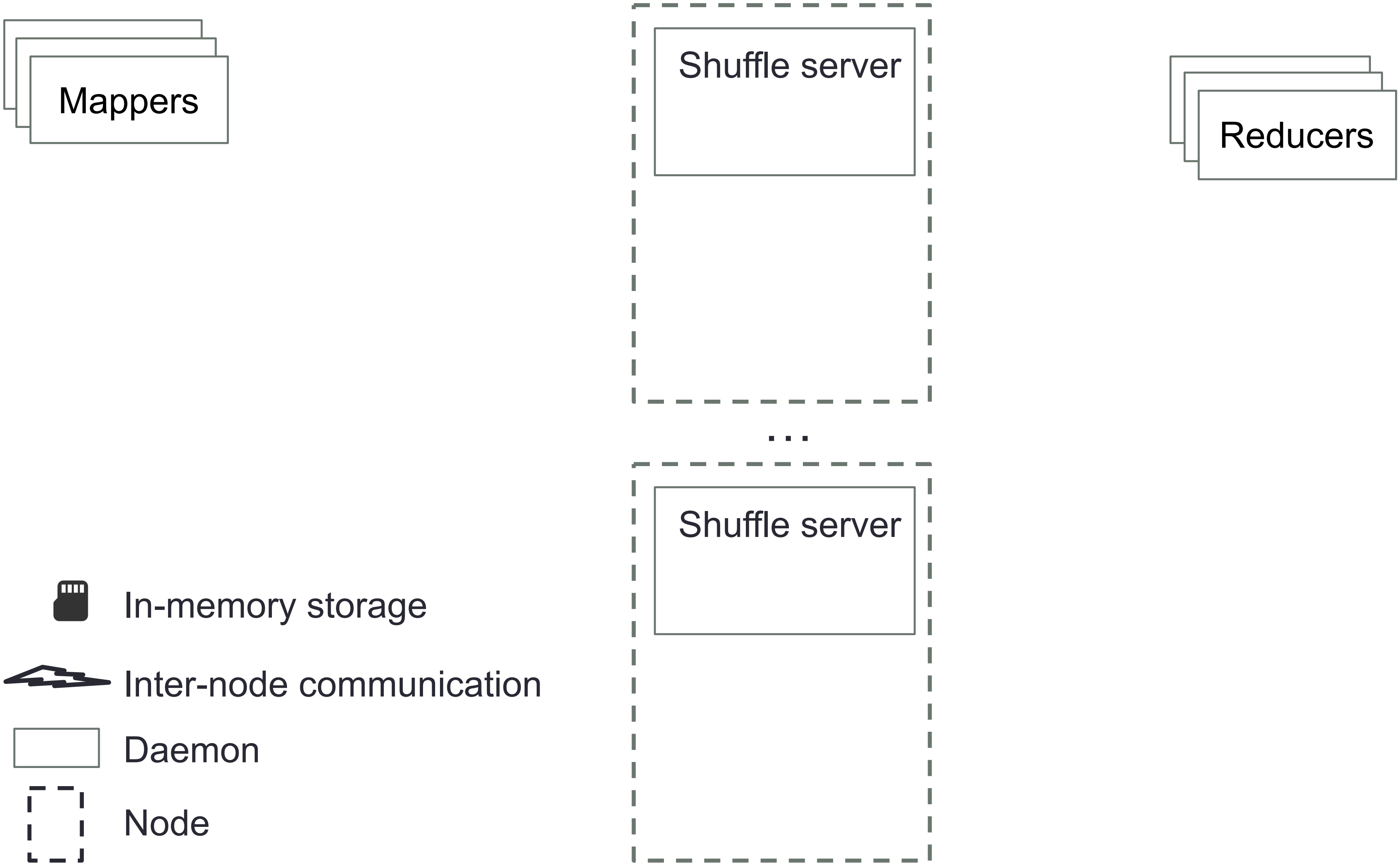


When storage on Node 1 is full, it can use memory of Node 2, 3, & 4

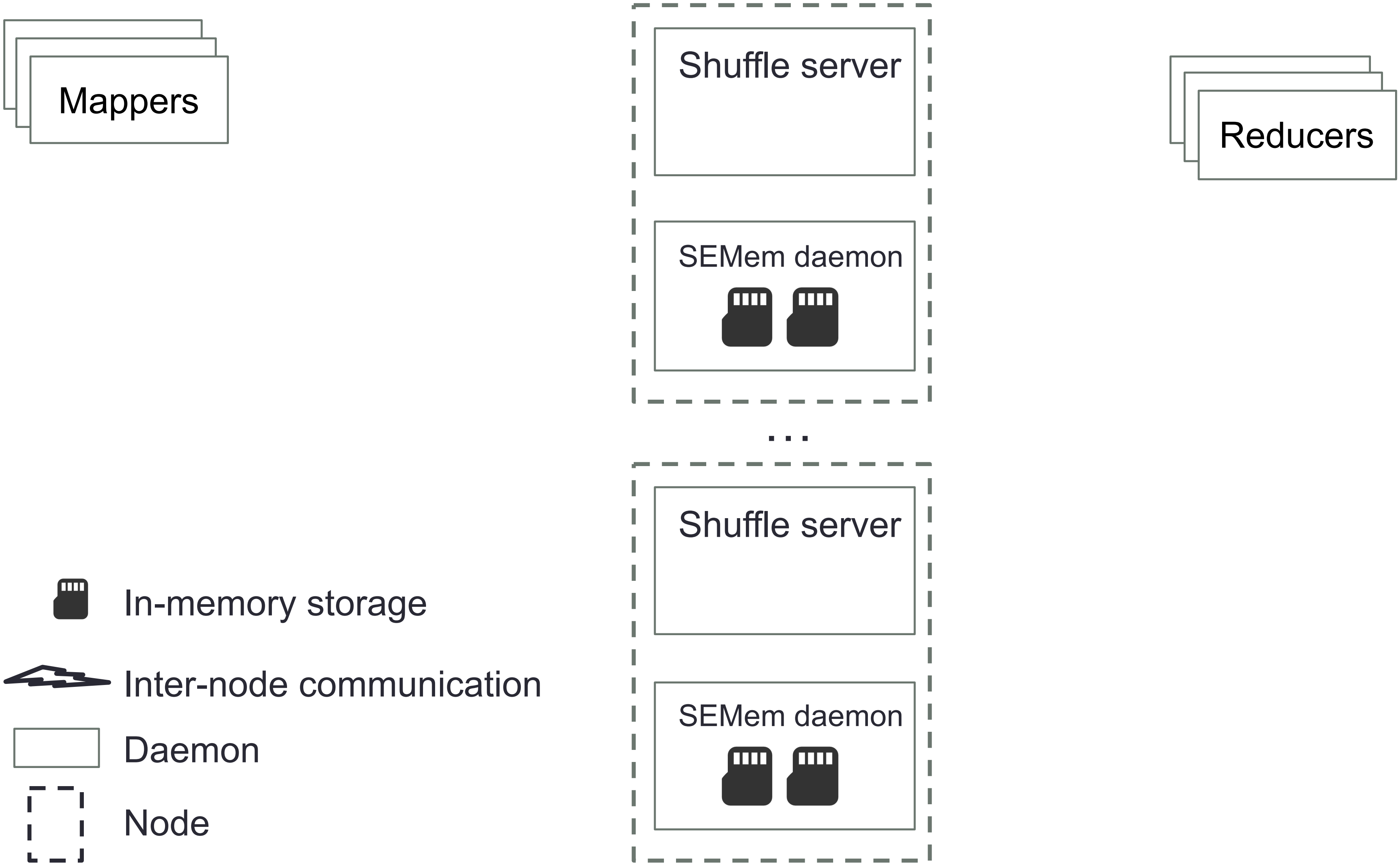


- More complex in-memory management is required

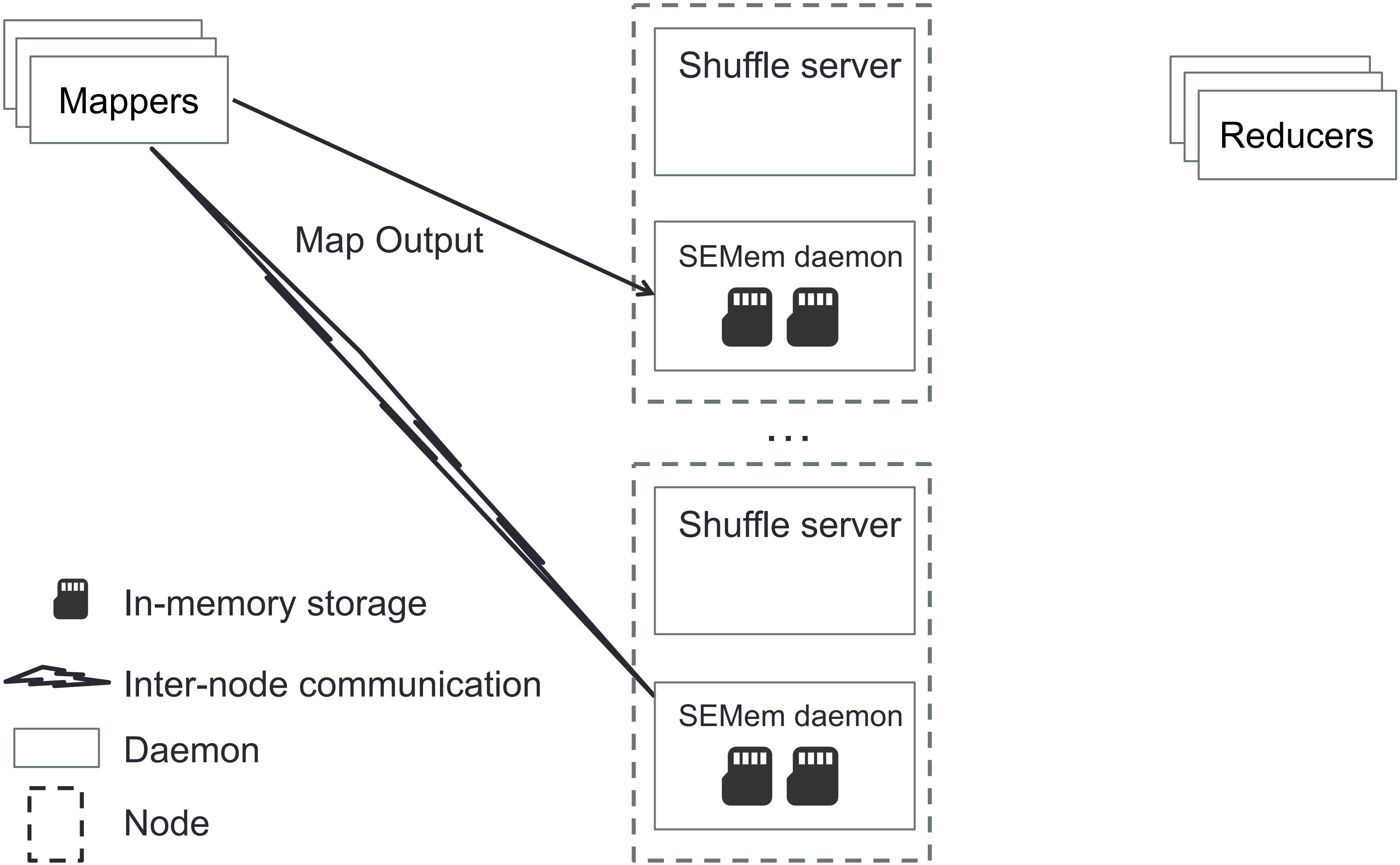
Every-node deployment on Hadoop workflow



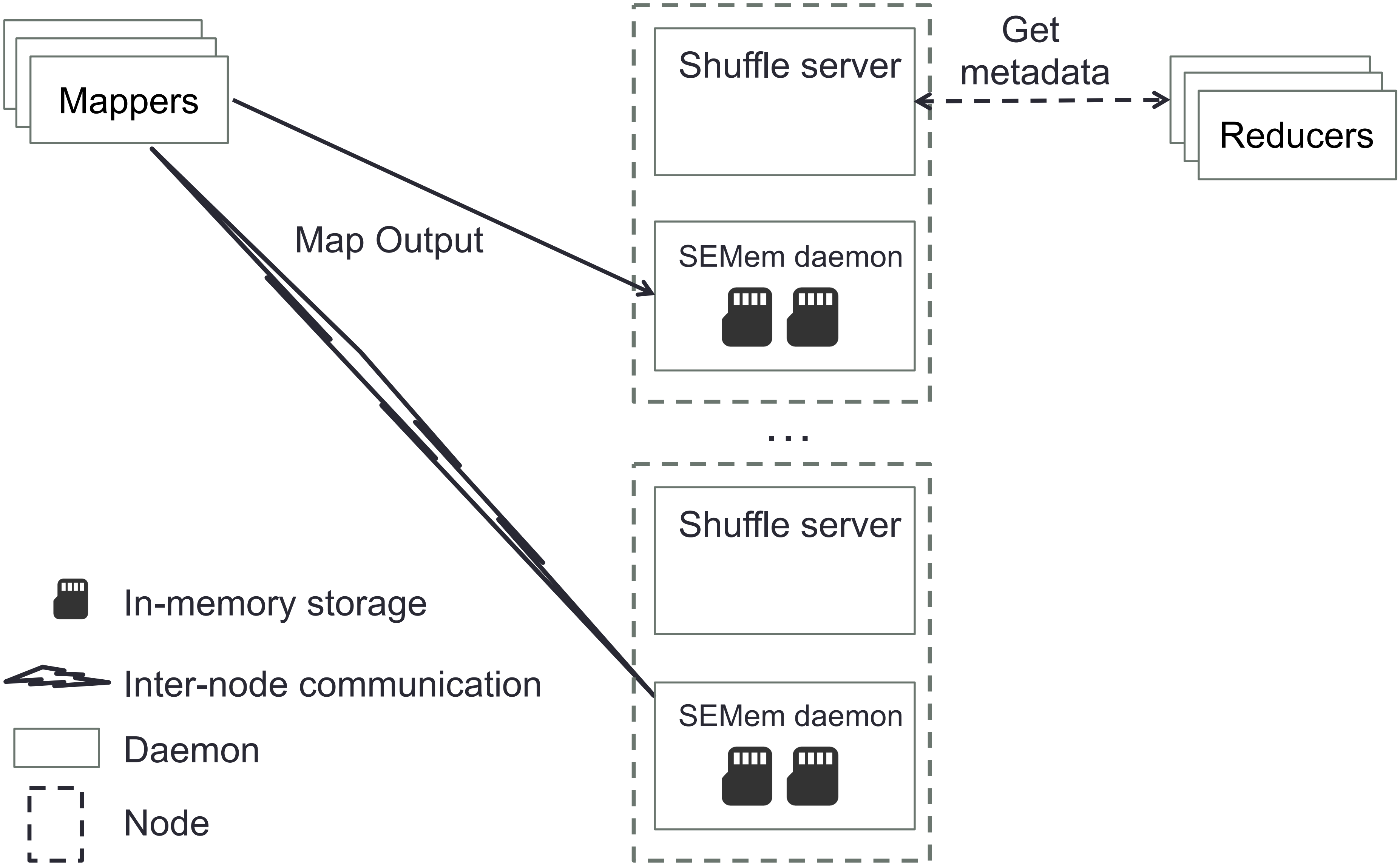
Every-node deployment on Hadoop workflow



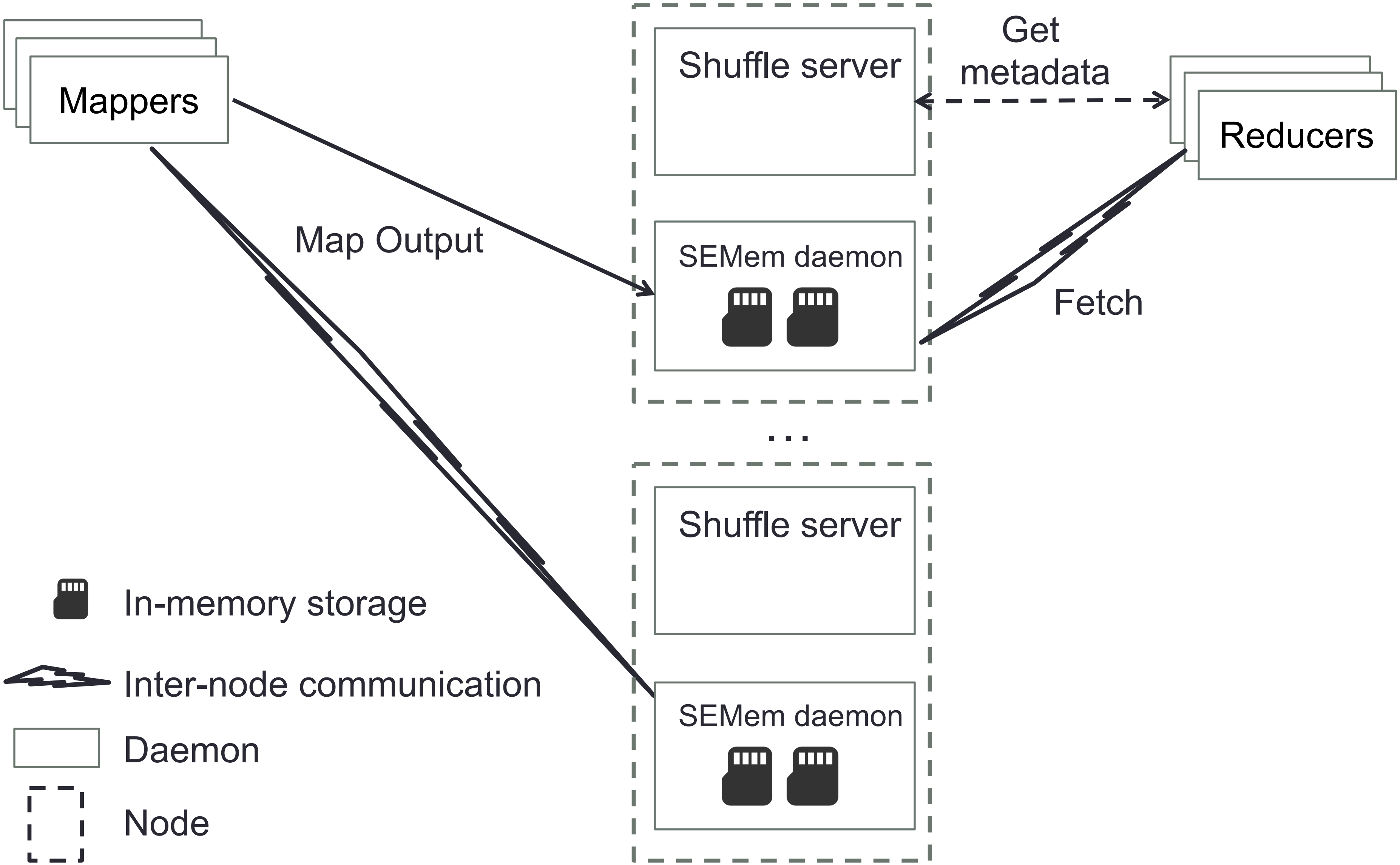
Every-node deployment on Hadoop workflow



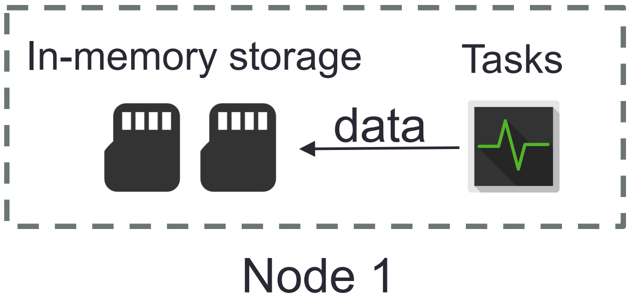
Every-node deployment on Hadoop workflow

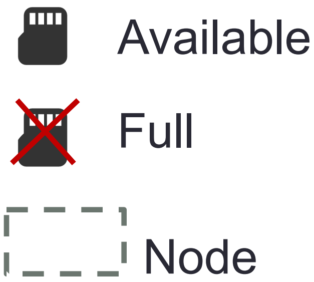


Every-node deployment on Hadoop workflow

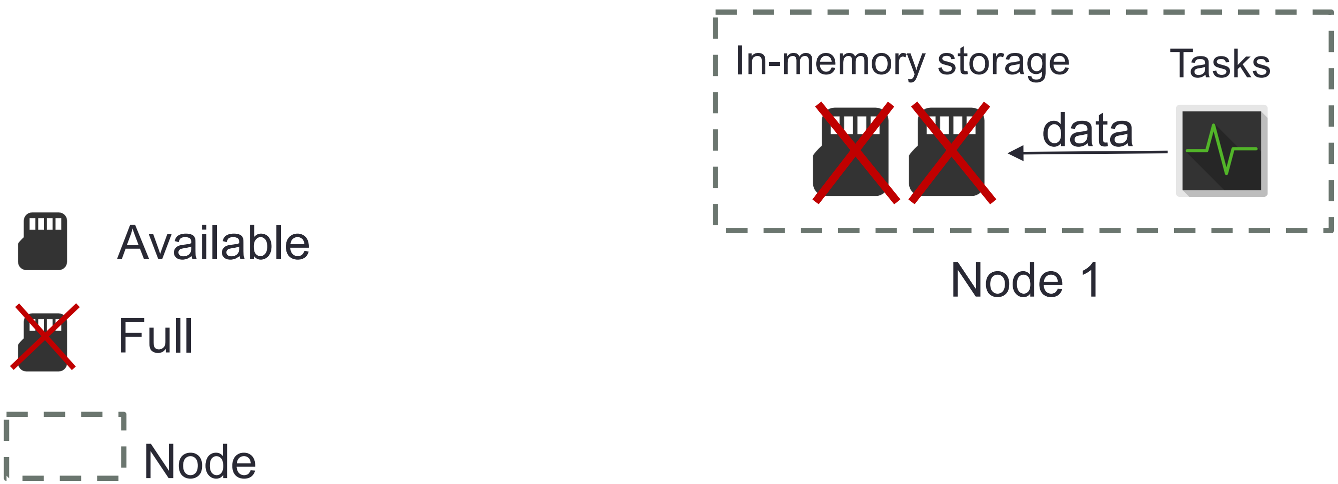


Dedicated-node: deployed only on dedicated nodes that are used only for storage

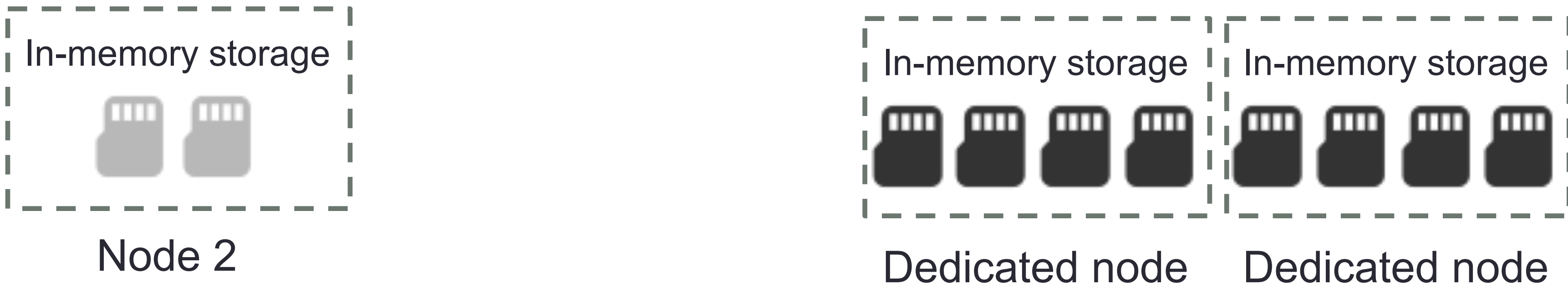


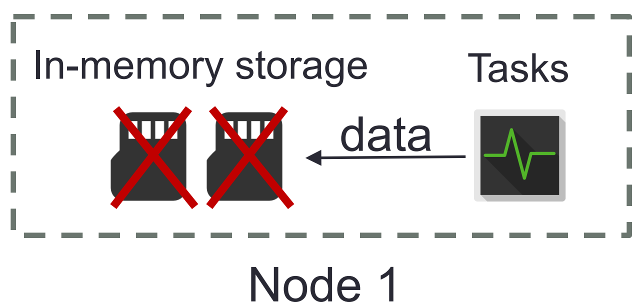


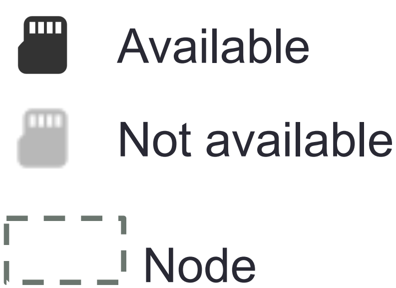
Dedicated-node: deployed only on dedicated nodes that are used only for storage



Dedicated-node: deployed only on dedicated nodes that are used only for storage

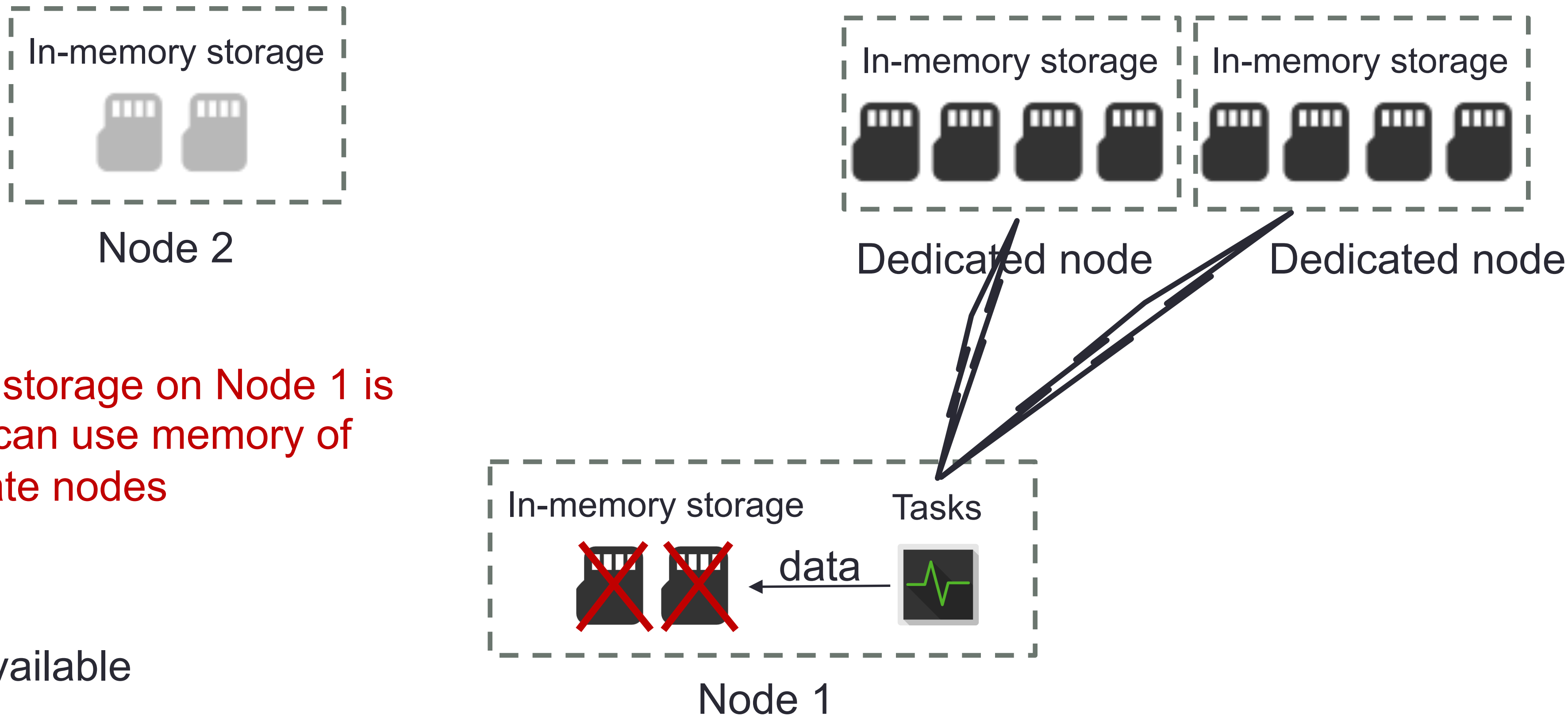




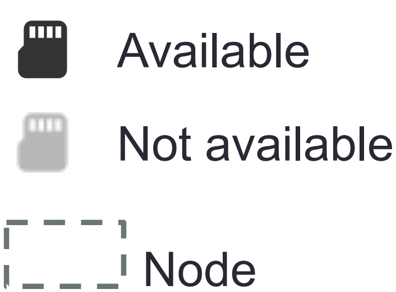


- Dedicated nodes
  - There is no computation task

Dedicated-node: deployed only on dedicated nodes that are used only for storage

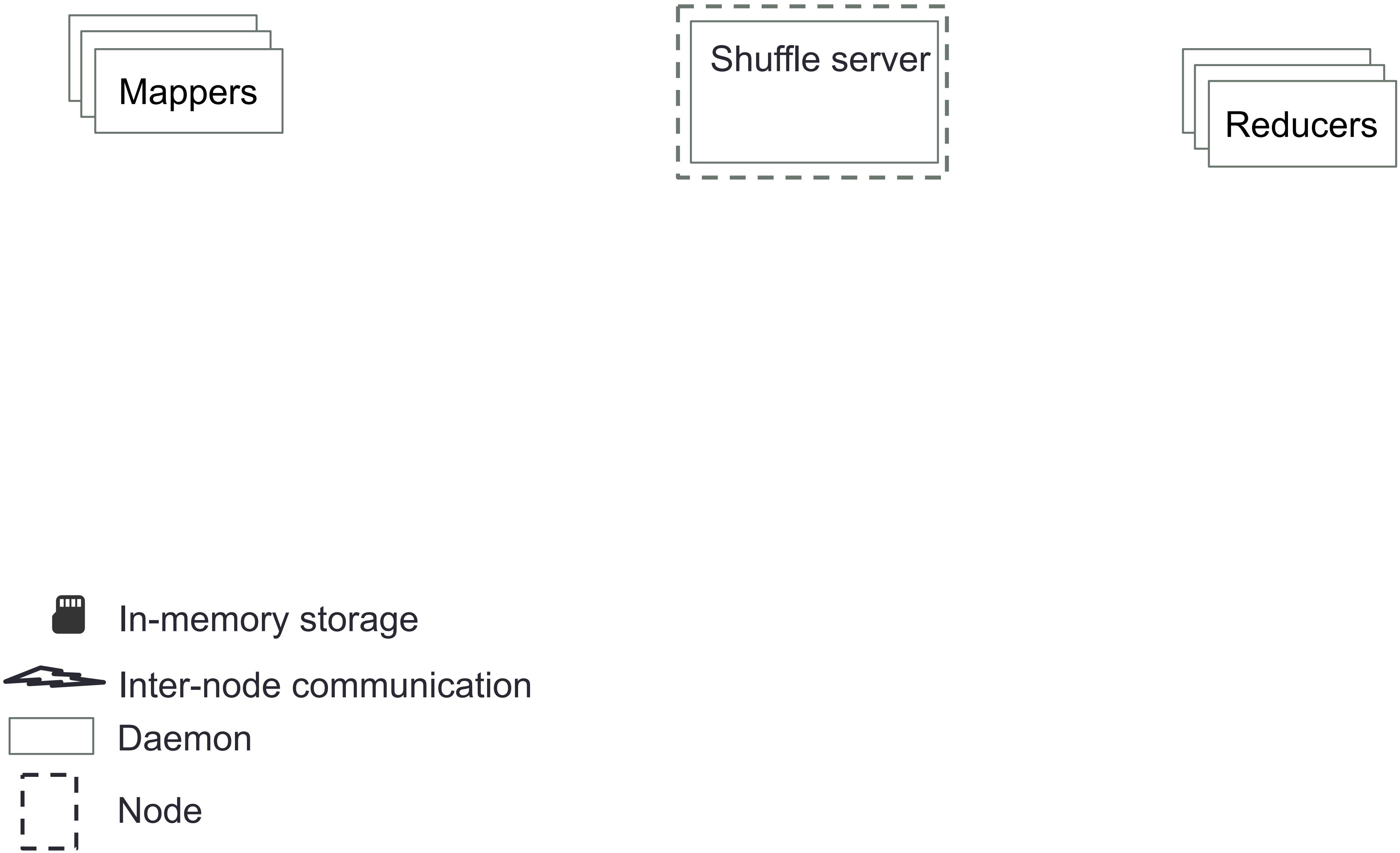


When storage on Node 1 is full, it can use memory of dedicate nodes

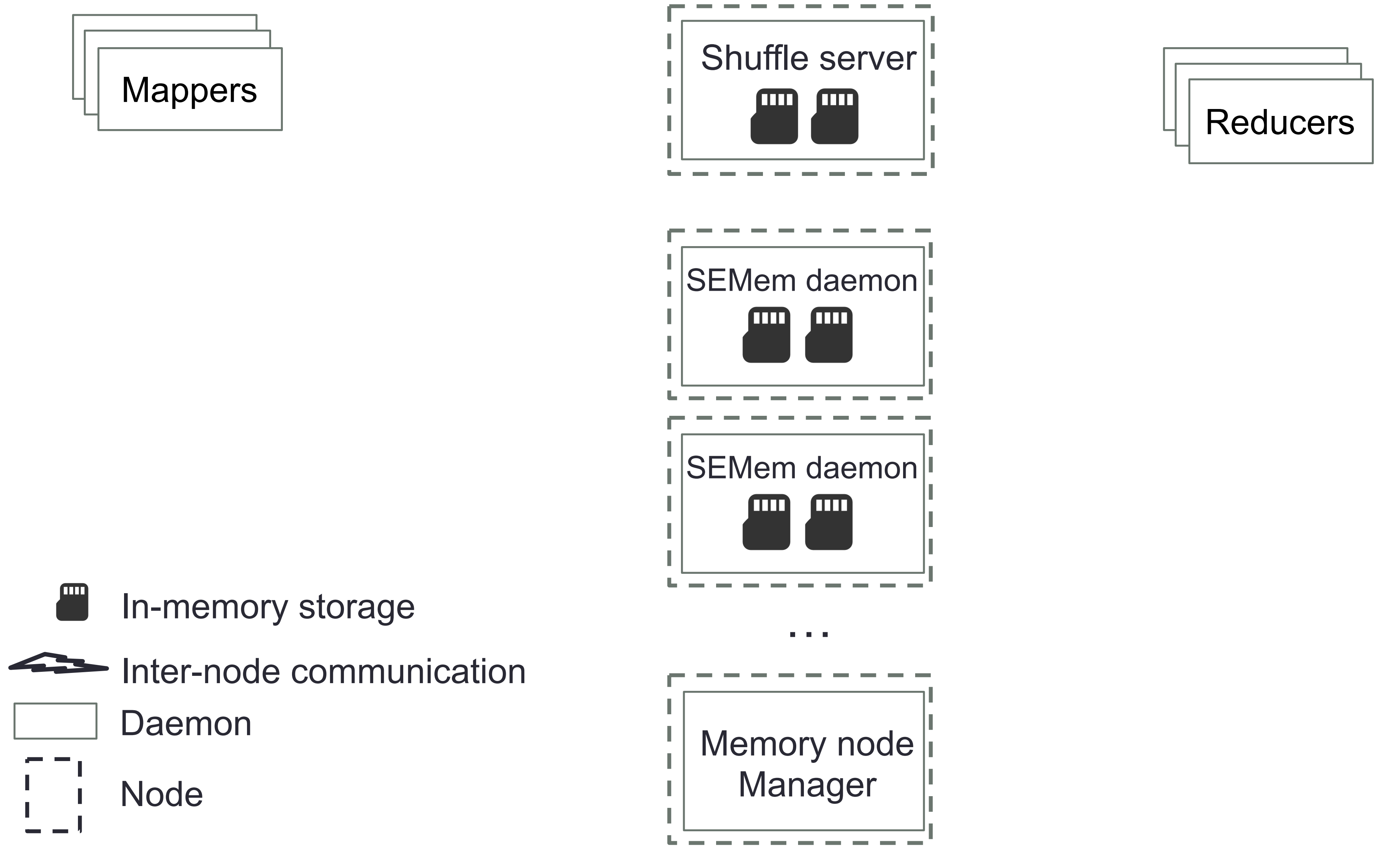


- Dedicated nodes
    - There is no computation task
- It might slow since only 2 of 4 nodes compute the task

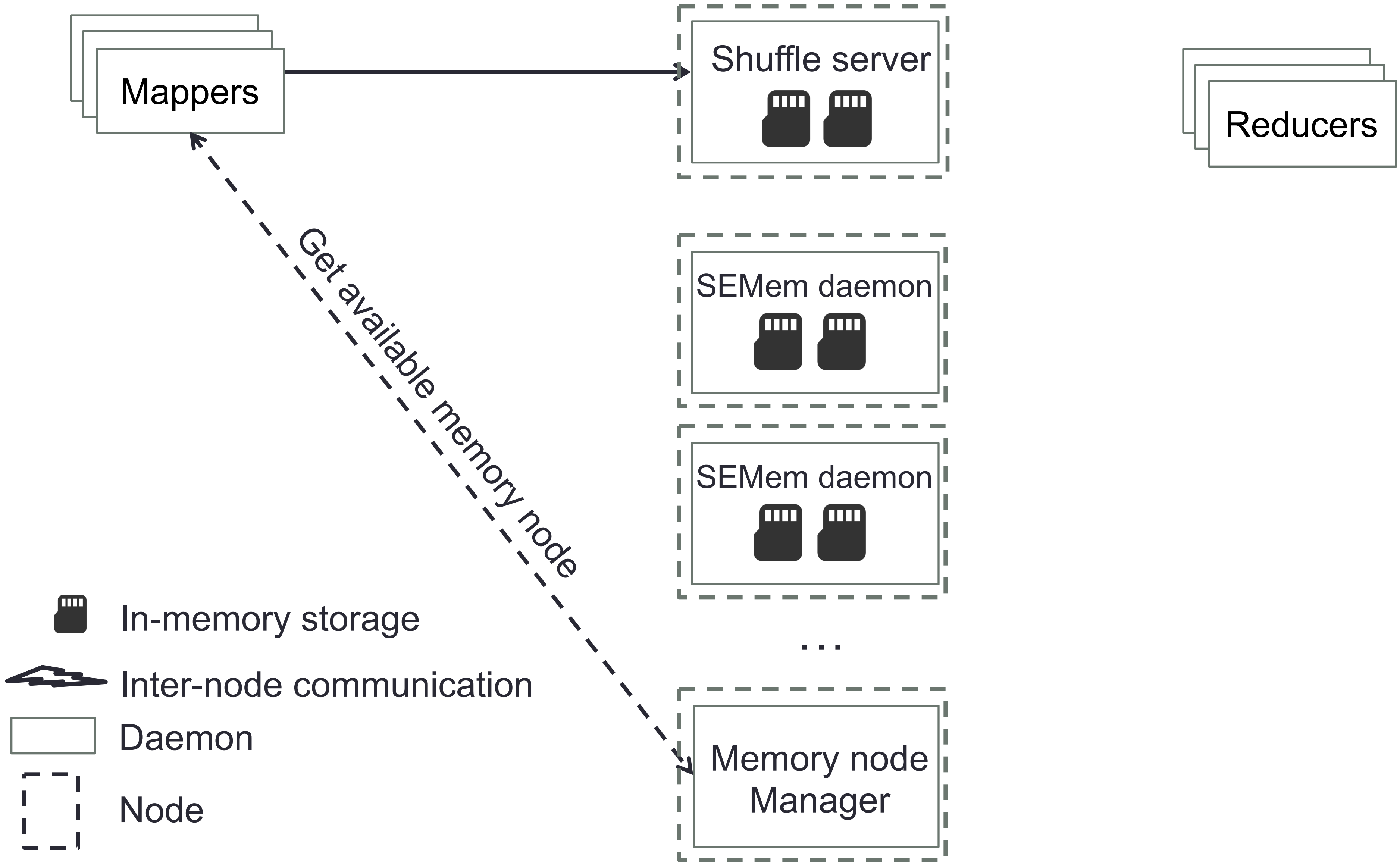
Dedicated-node deployment on Hadoop workflow



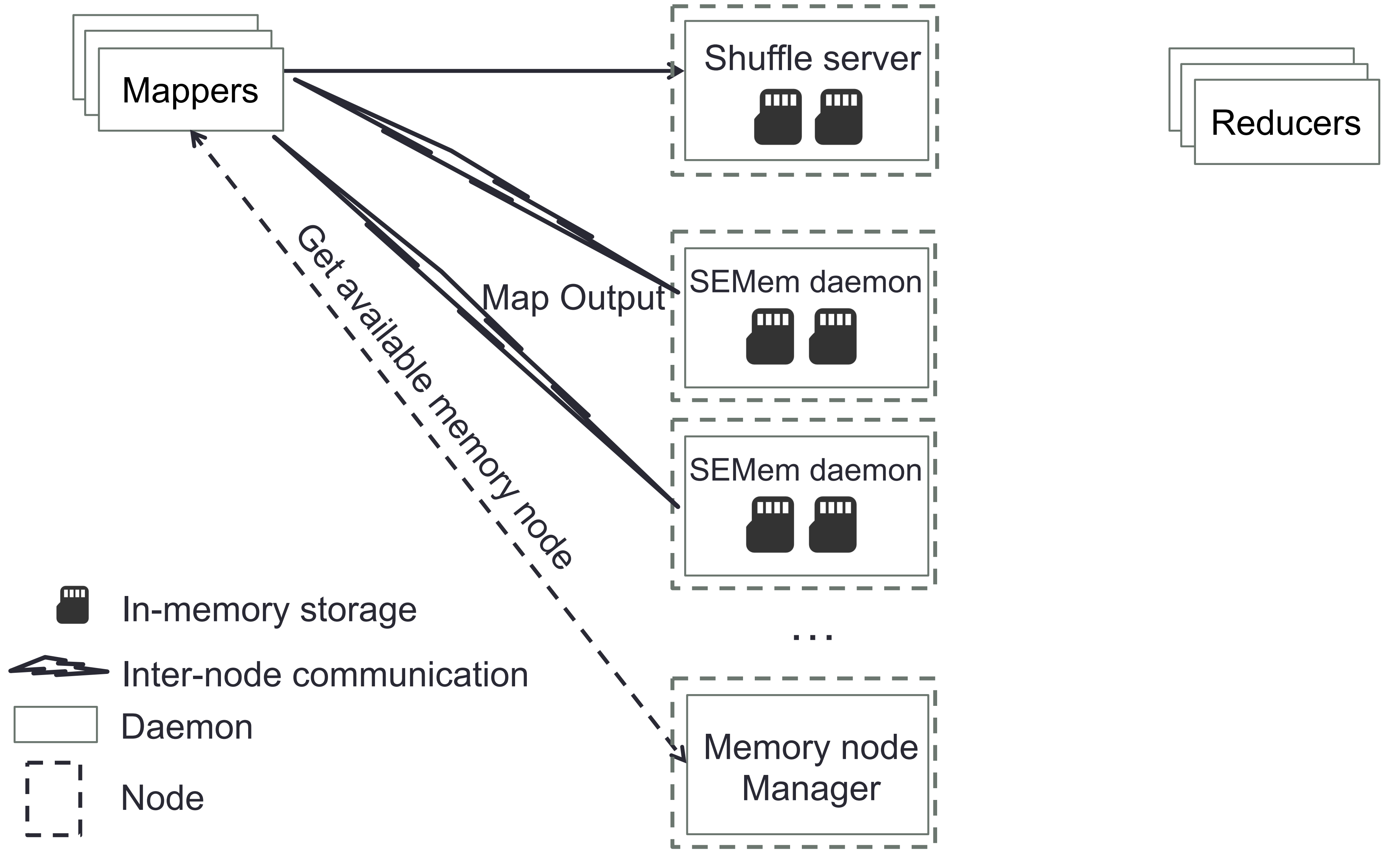
Dedicated-node deployment on Hadoop workflow



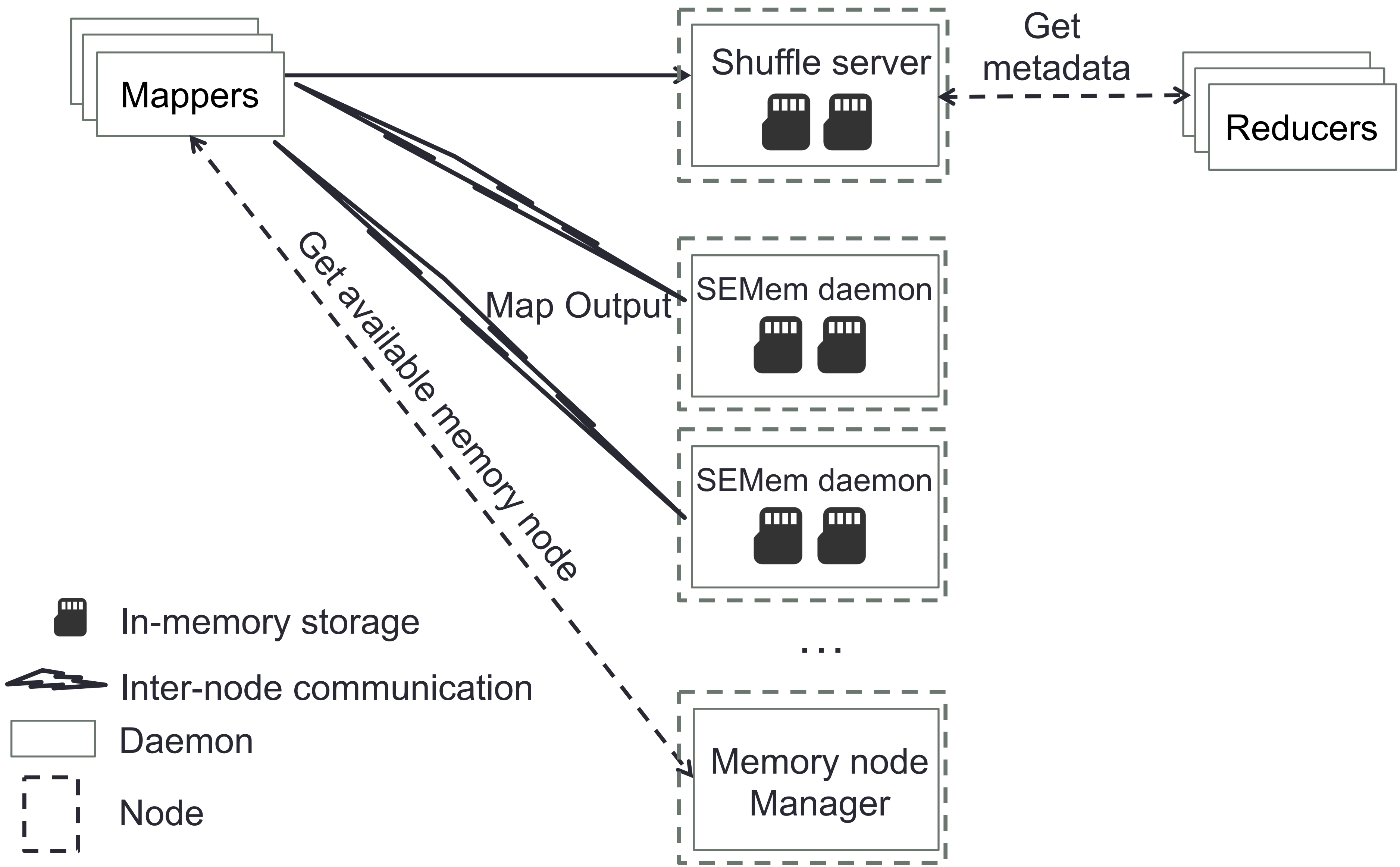
Dedicated-node deployment on Hadoop workflow



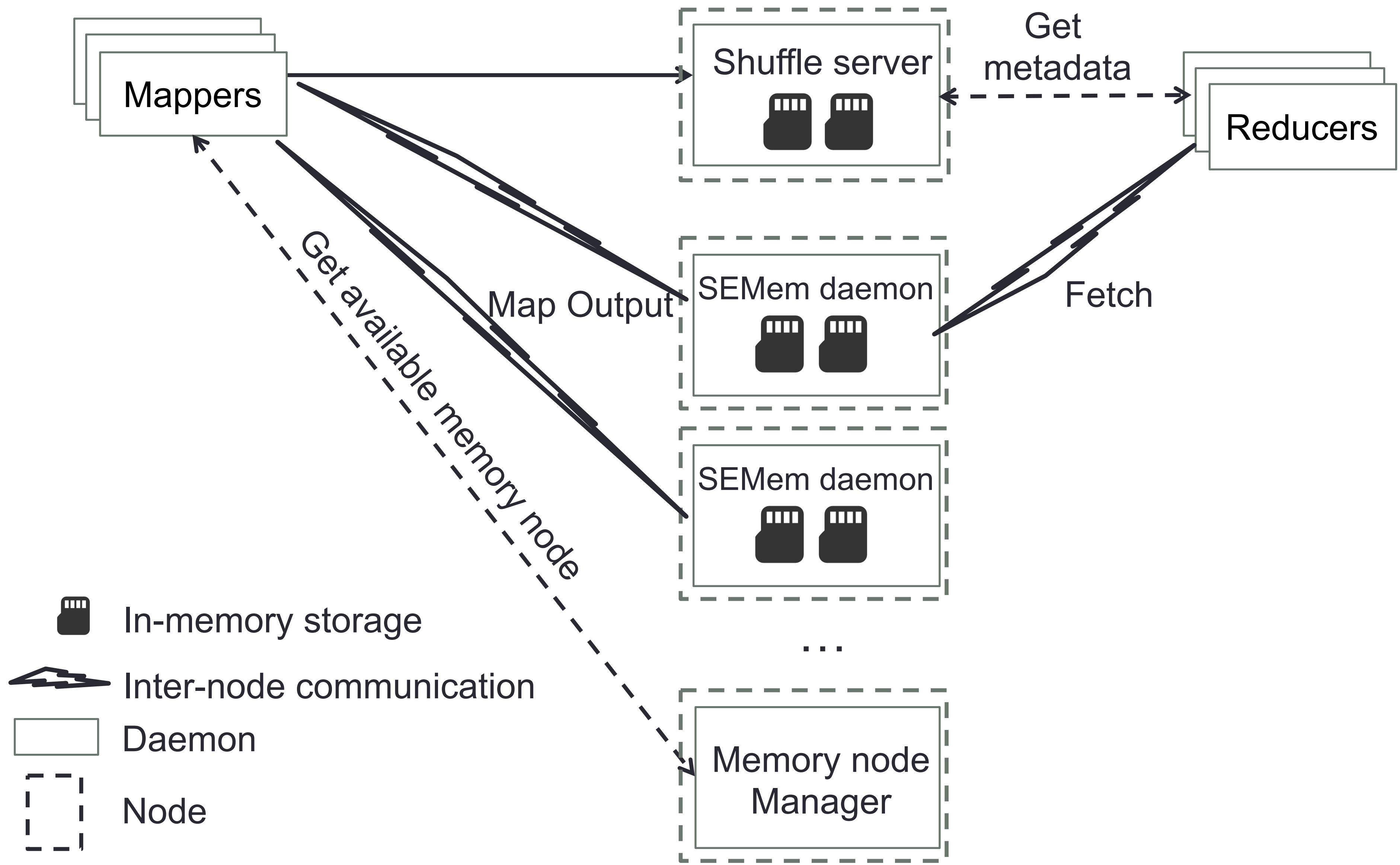
Dedicated-node deployment on Hadoop workflow



Dedicated-node deployment on Hadoop workflow



Dedicated-node deployment on Hadoop workflow



Technical issue 1: communication protocol on SEMem

- MPI communication on SEMem
  - Fast communication protocol is required
    - Since every-node and dedicated-node are network-intensive
    - MPI is the de facto communication on modern supercomputers
  - HPC-Reuse is used
    - Enable MPI over Hadoop processes
      - MPI-friendly dynamic process creation is required
  - Multiplexing non-blocking MPI on memory nodes
    - Since we want to avoid MPI_THREAD_MULTIPLE
      - Handling multiple requests from clients
  - Direct memory is used
    - Since memory copying between JVM's heap and native MPI is slow
      - Current MPI implementation is written in C

MPI over Hadoop processes [Dao, CCGrid 2016]

- Using our HPC-Reuse
  - MPI-friendly dynamic process creation
- Hadoop requires dynamic process creation
  - Minimizing the cost of changes in architecture
- Gang scheduling (of processes) more favorable in MPI
  - All-or-nothing scheduling strategy
    - Statically creating all processes at the beginning
    - Minimizing communication delay
    - Since resizing running jobs might affect performance and fairness
  - MPI-Spawn is slow due to collective operation

Avoiding MPI_THREAD_MULTIPLE

- Multiplexing non-blocking MPI

```
while true do
    if req == null then
        req = MPI.iRecv
    end
    if there is a new request then
        Add req to sendingPool's waitList;
        Reset req = null
    end
    for slot in sendingPool's slots do
        if data reading finishes then
            MPI.iSend to the client
        end
        if iSend finishes then
            free the slot
        end
    end
    Assign req in waitList to free slots;
end
```

At SEMem daemon

```
while any MapOutput do
    Wait for (host, MapOutputs) ;
    for each MapOutput do
        MPI.Send to the host ;
        if MPI.Recv from the host then
            Data in heap ;
        end
    end
end
```

At clients

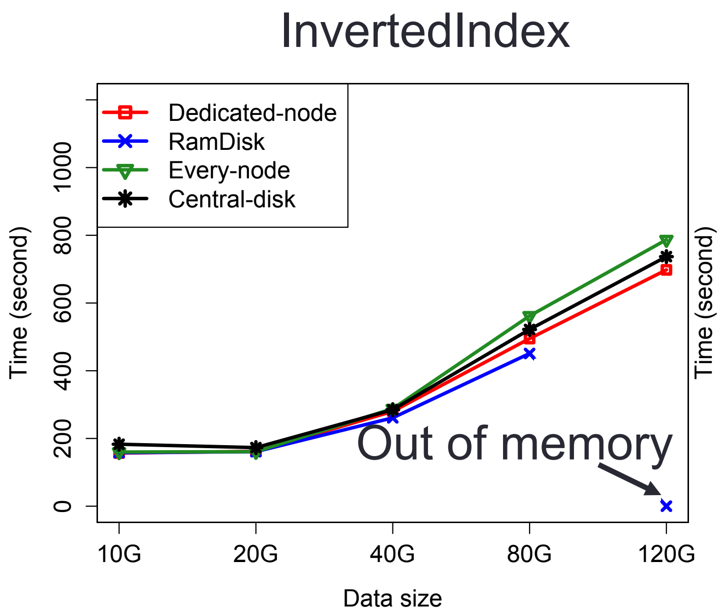
Technical issue 2: storage size in dedicated-node SEMem

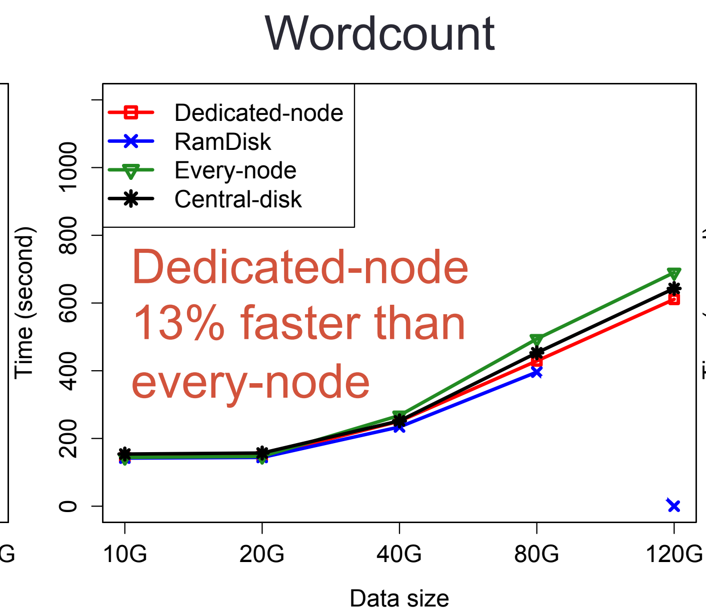
- How to estimate number of memory nodes
  - Need to minimize number of memory nodes
    - Since we trade computation resource for data storage in dedicated-node
  - Our approach: number of memory nodes is estimated roughly based on the size of input data

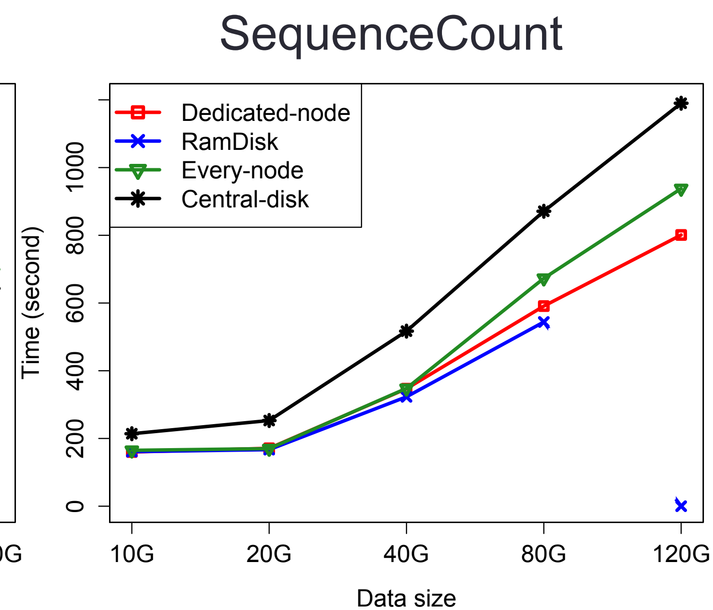
Experiment configuration

- Benchmarks
  - Puma: WordCount, InvertedIndex, and SequenceCount
  - Tera-sort: up to 1 TB of input data
- Supercomputers
  - K computer-like FX10 at the University of Tokyo
  - TSUBAME at Tokyo Institute of Technology
- Hadoop v2.2.0

Dedicated-node is faster than every-node in some benchmarks



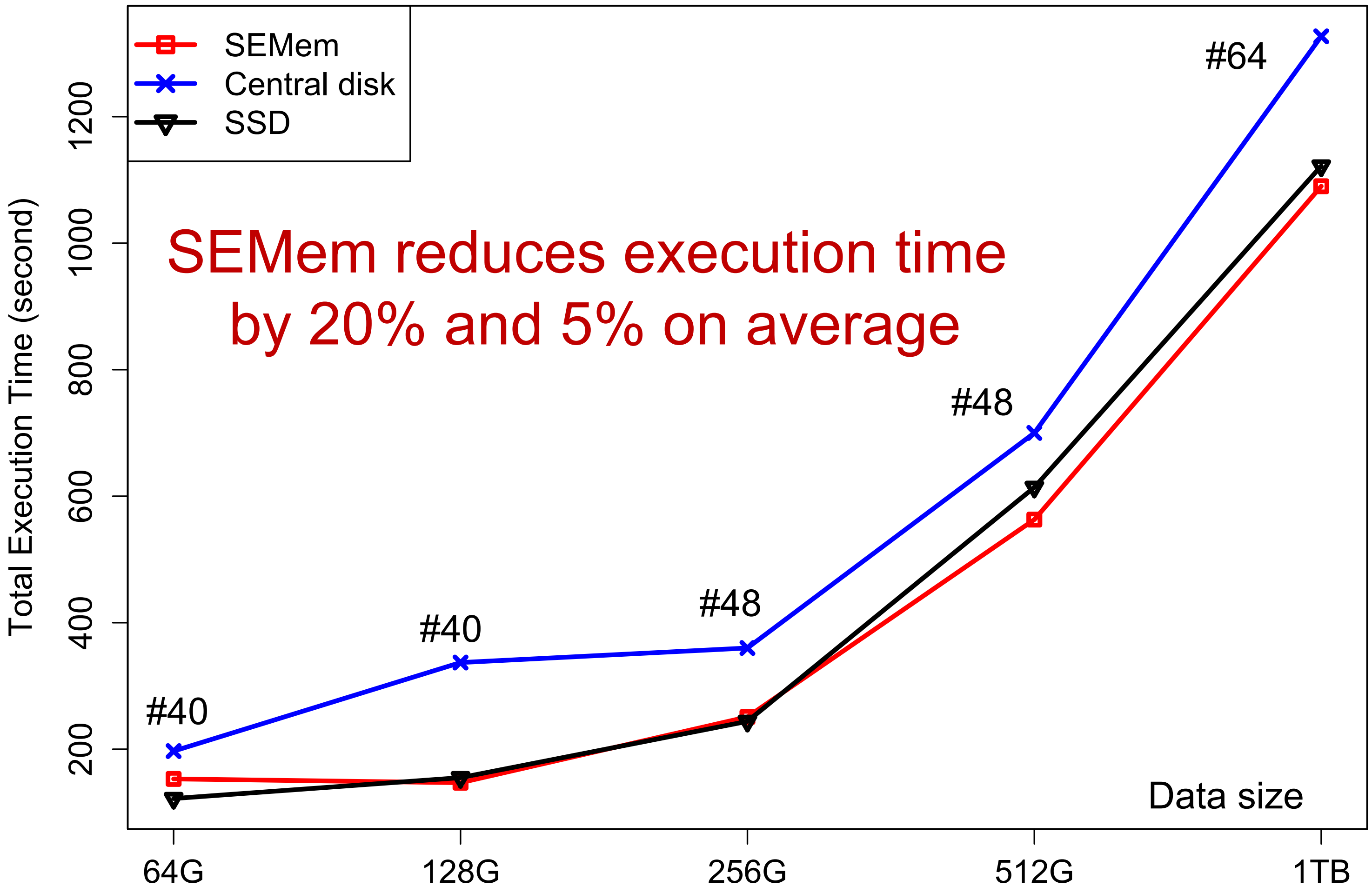




Configuration	#computation nodes
RamDisk	36
Every-node	36
Central-disk	36
Dedicated-node	32

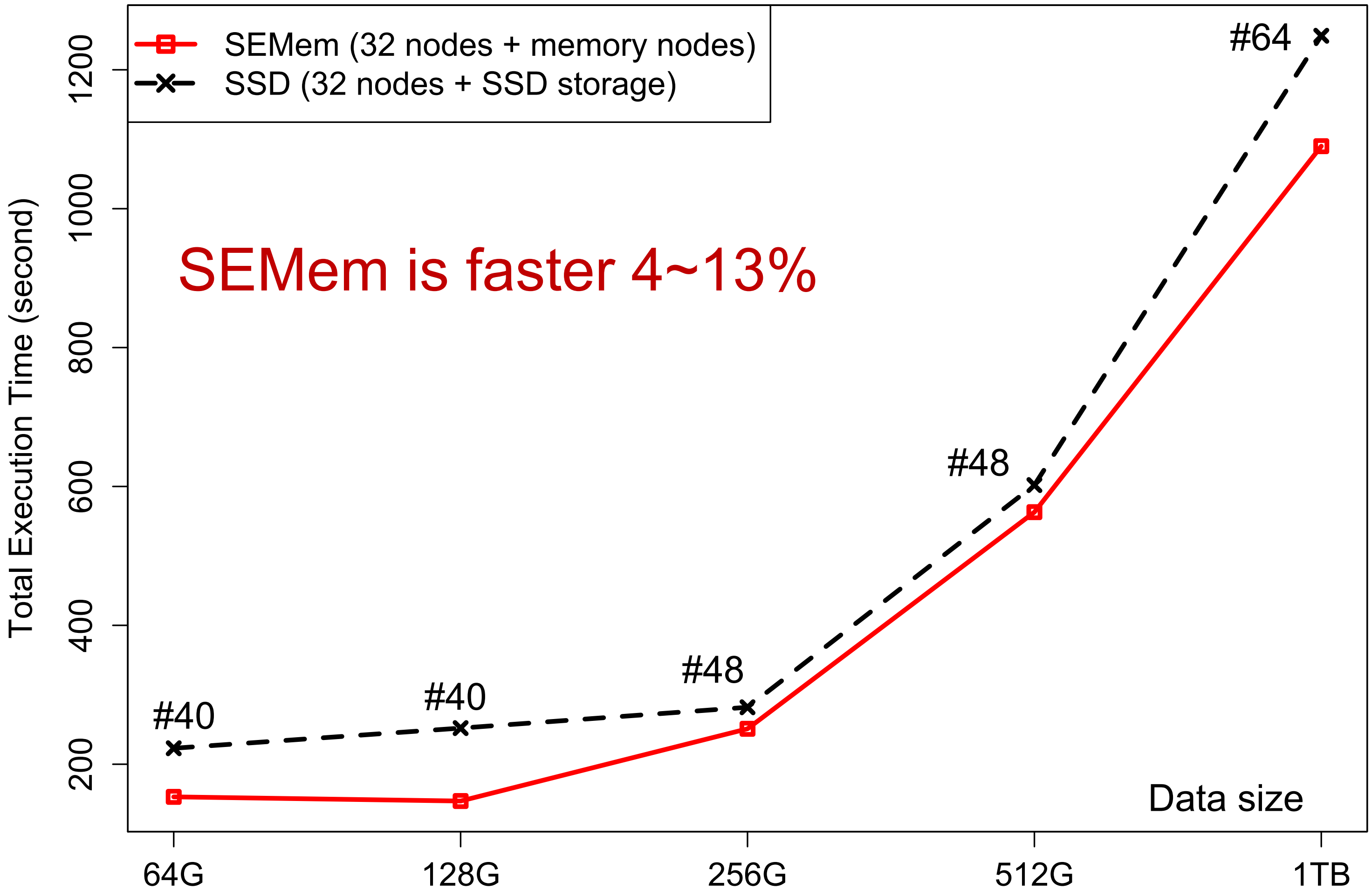
- In every-node deployment, the more complex SEMem daemon disturbs computation tasks at several places
- 4 memory nodes in dedicated-node

Dedicated-node SEMem is faster than central disk and SSD



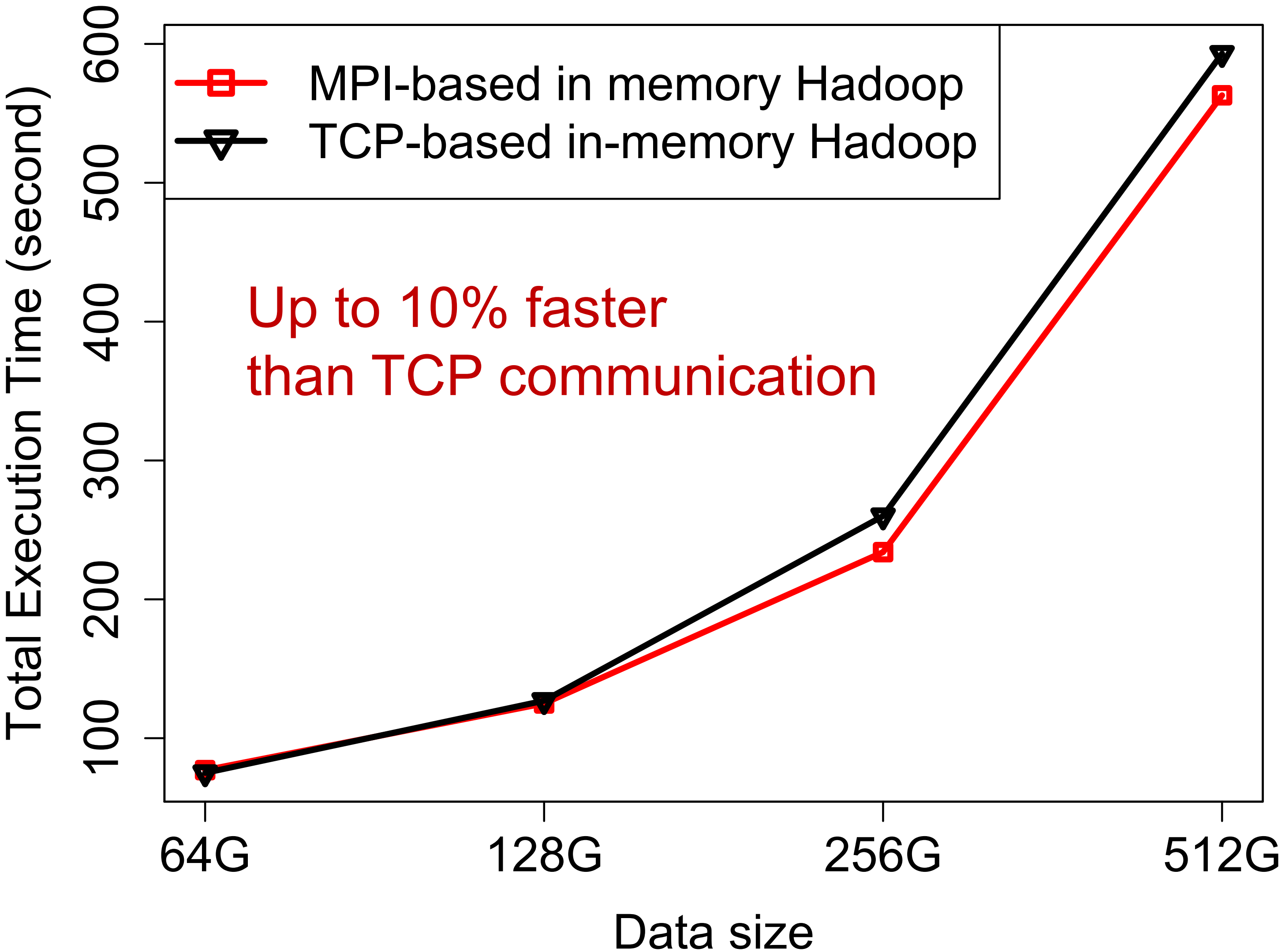
- Total number of nodes is the same
- SEMem has less computation nodes
- Tera-sort application on TSUBAME

Dedicated-node SEMem and SSD-backed storage can be an alternative for each other



- Both have the same number of computation nodes
- SEMem has memory nodes
- Tera-sort application on TSUBAME
- SSD size of each node is 120 GB

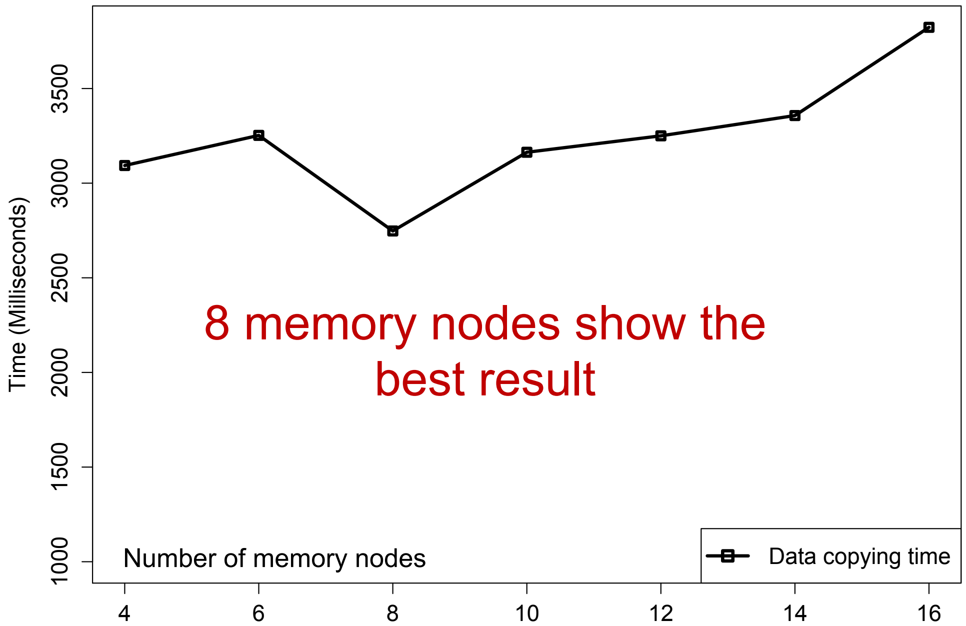
MPI-based Hadoop is faster than TCP

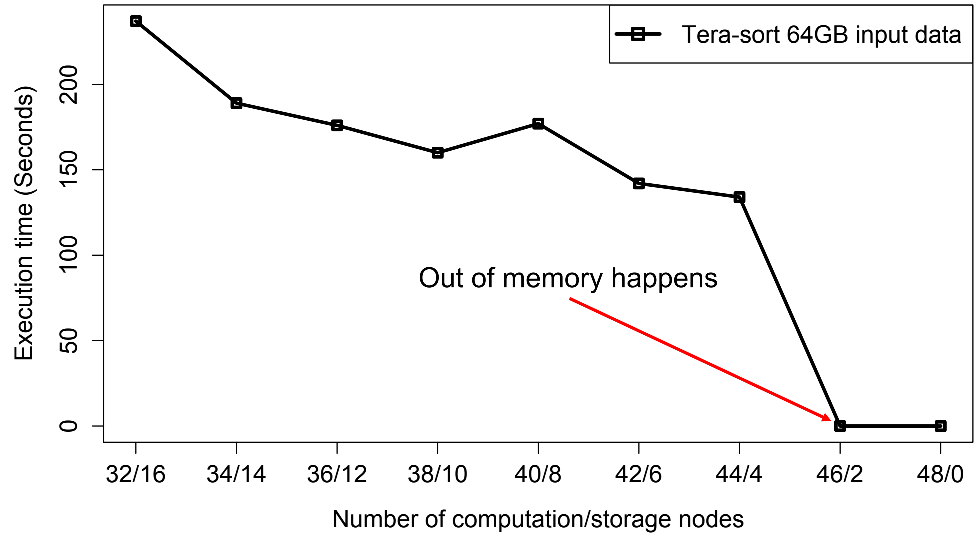


- Tera-sort on TSUBAME (64 nodes)
- Both MPI and TCP test cases using in-memory storage

Number of memory nodes should not be large

- Experiment purpose:
  - Measure performance impact of storage size (left figure)
  - When out of memory happens (right figure)
- Number of memory nodes is estimated based on size of input data
  - 64GB of input data
  - 8GB each memory node





Tera-sort on TSUBAME

Related work

- M3R (VLDB 2012)
  - In-memory storage by providing a shared heap-state
  - Data is stored through *places* and *activities* operators
  - Did not mention storage deployment explicitly and also no evaluation
- HaLoop (VLDB 2010)
  - Caching preferences by providing efficient hash algorithms for reading and writing
  - Deployment strategies are not relevant in this context
- Spark (NSDI 2012)
  - Use in-memory storage
  - Choose a location for Resilient Distributed Datasets (RDD) through *preferredLocations()* operator
  - Does not provide deployment strategies in general
  - There was no evaluation of RDD deployment

Limitations

- No fault-tolerance in MPI
- Multiple levels of storage
- Preferred locations

Future work

- Estimating number of memory nodes
- Topology of memory nodes

Summary

- Goal: Using in-memory storage to provide efficient virtual local disks

- Challenge: choose the best deployment strategy of in-memory storage (or virtual local disks)

- Our approach: SEMem
  - Dedicated-node strategy showed a good performance in some benchmarks
  - Easily configure the system to investigate an appropriate strategy for applications
  - MPI communication