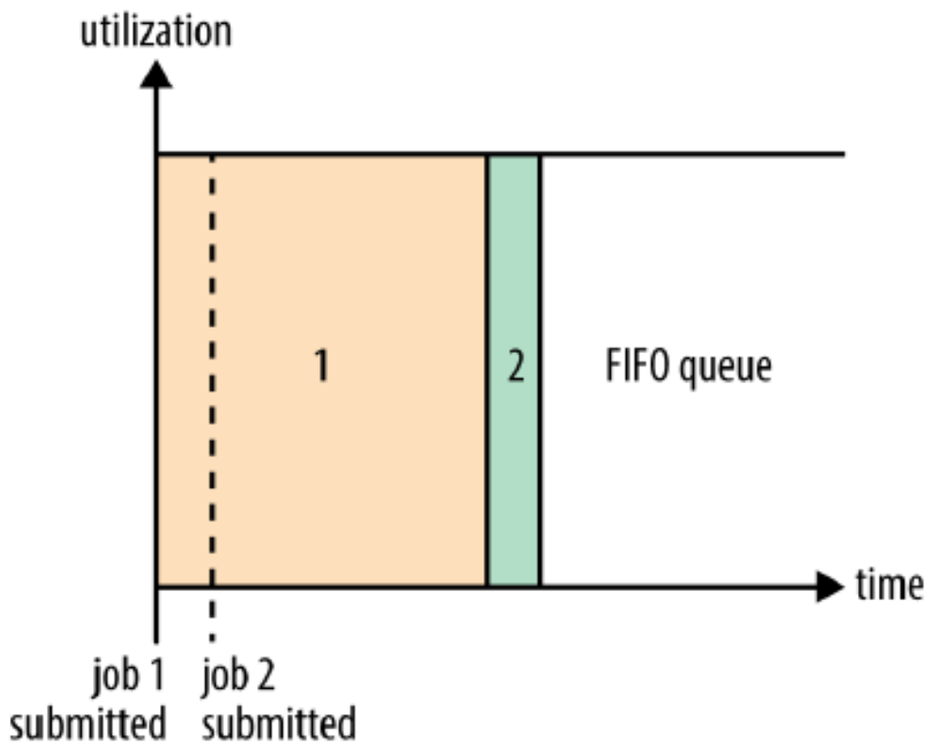
YARN Scheduling (I)

- In the real world, cluster resources are limited
  - Scheduling (i.e., allocating resources to applications) is a complex problem and there is no one “best” policy
- YARN provides different schedulers that allocate resources to applications based on different policies

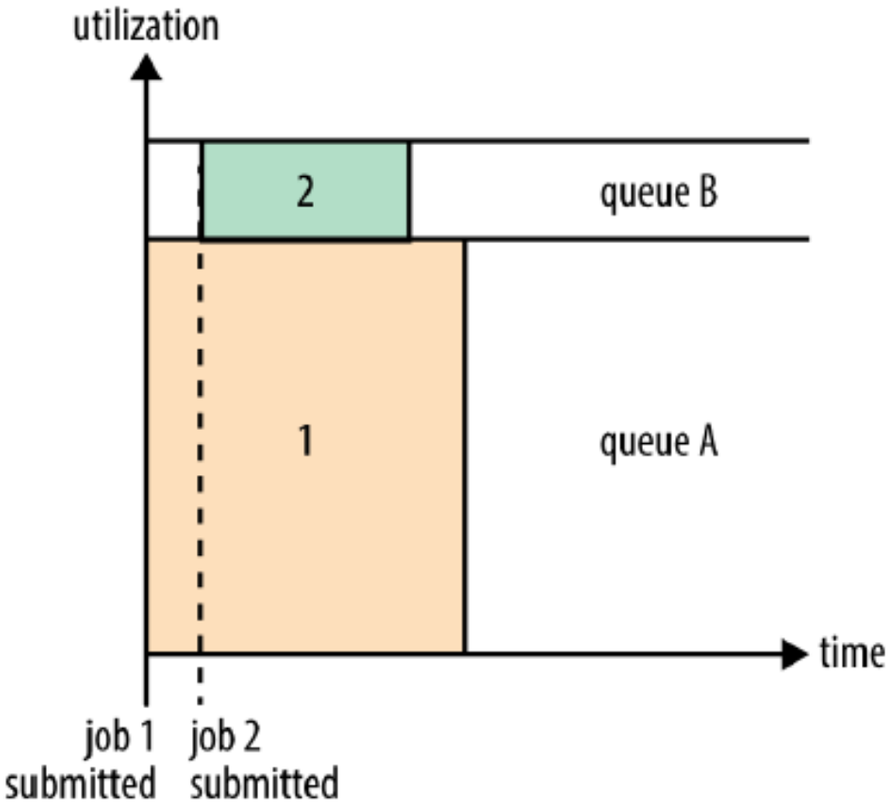
YARN Scheduling (II): FIFO Scheduler

- The FIFO Scheduler places applications in a single queue and runs them in the order of submission (first in, first out)
  - Requests for the first application in the queue are allocated first; once its requests have been satisfied, the next application in the queue is served
  - Simple to understand and no configuration needed
  - Not suitable for shared clusters. Large applications will use all the resources in a cluster, so other applications have to wait their turn



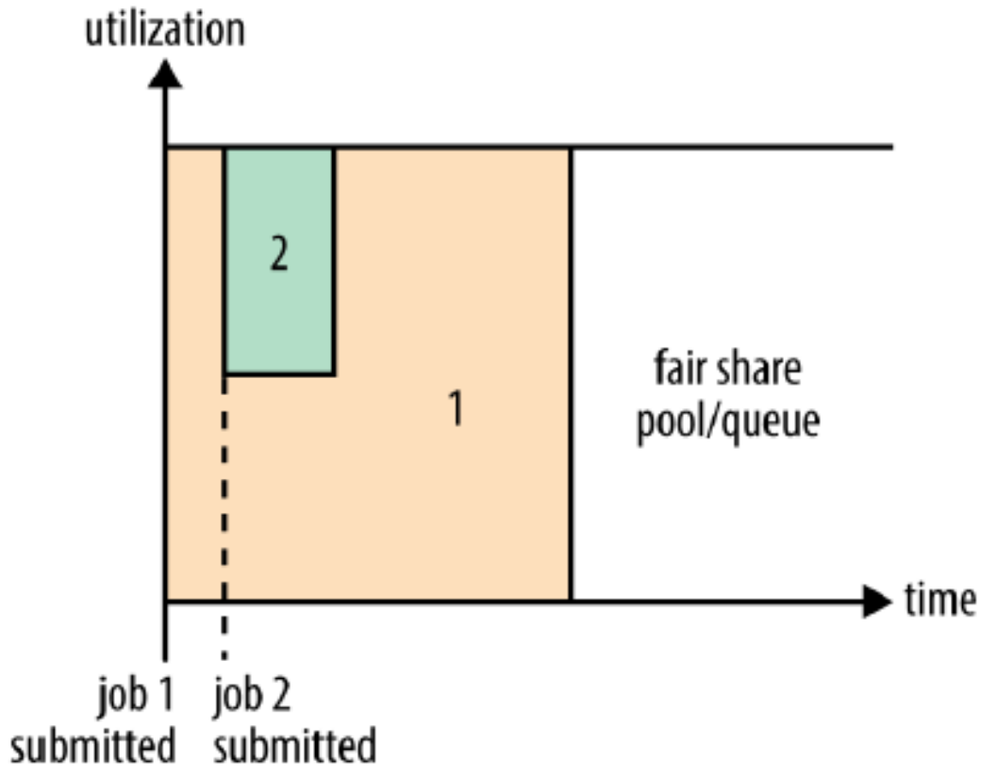
YARN Scheduling (III): Capacity Scheduler

- The Capacity Scheduler has separate queues
  - Each queue dedicated to an organization accessing the cluster
  - Each queue configured to use a given fraction of the cluster capacity
  - Within a queue, applications are scheduled using FIFO scheduling
  - A large job finishes later than when using FIFO Scheduler



YARN Scheduling (IV): Fair Scheduler

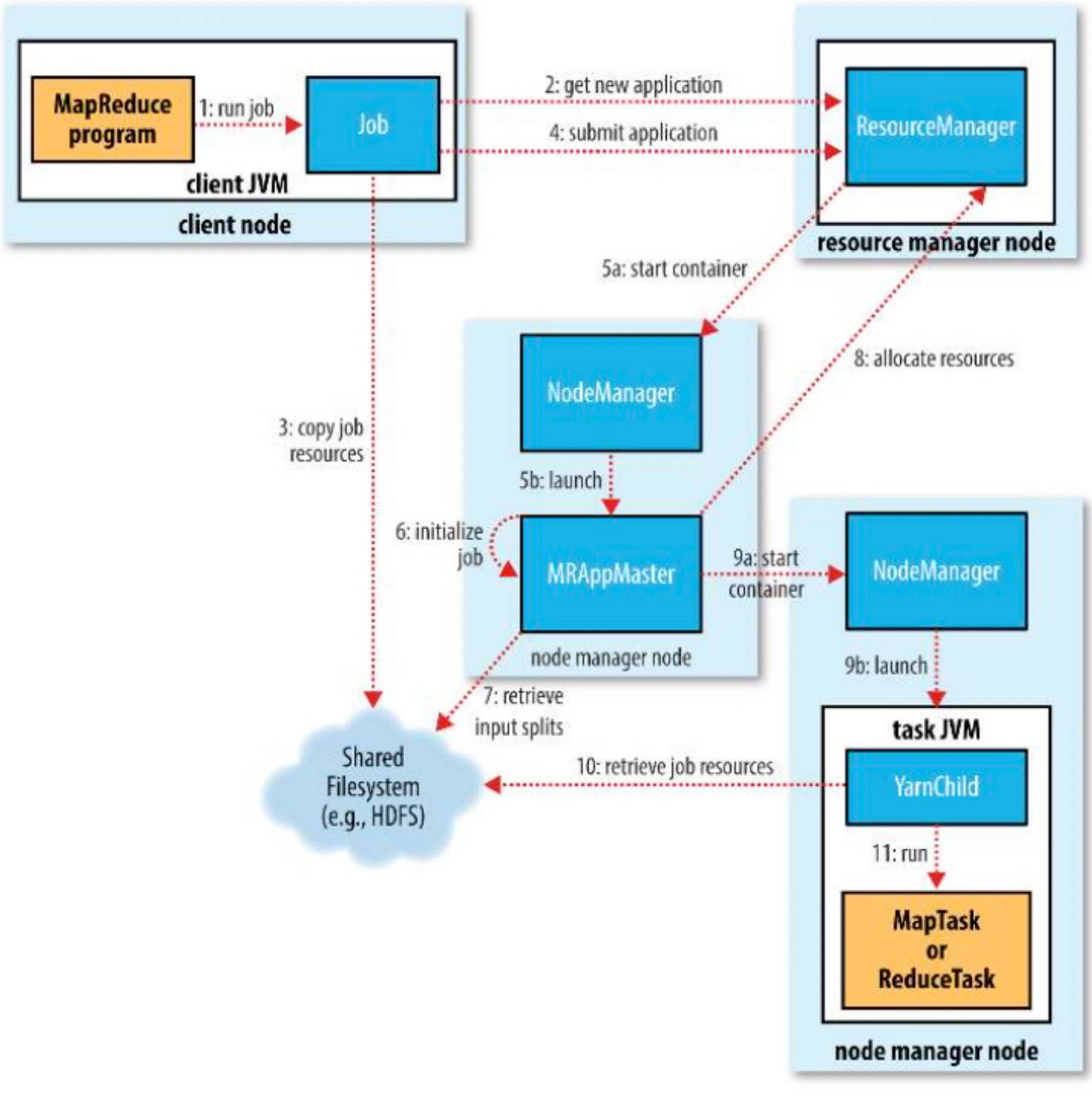
- The Fair Scheduler
  - There is no need to reserve a set amount of capacity
  - It will dynamically balance resources among all running jobs
  - E.g., when the first job starts, it gets all cluster resources (it is the only one running). When the second job starts, it gets half of the cluster resources, and so on
  - Fair scheduling is per-user: if user A submits two jobs and user B only one, then each A's job gets 25% while B's job gets 50% of resources



YARN Scheduling (V): Delay Scheduling

- All YARN schedulers try to honor locality requests.
- On a busy cluster, however, there is a good chance that a required node is saturated by other containers at the time of a request
- As seen, we can loosen the constraint by allocating resources on another node on the same rack
- In practice, waiting a short time (no more than a few seconds) can dramatically increase the chances of being allocated a container on the requested node, and therefore increase the efficiency of the cluster
- This feature is called Delay scheduling and is supported by both the Capacity Scheduler and the Fair Scheduler
- Every node manager in a YARN cluster periodically sends a heartbeat request to the resource manager
  - Heartbeats carry information about the node manager's running containers and the resources available for new containers
  - Each heartbeat is a potential scheduling opportunity for an application to run a container
  - When using delay scheduling, the scheduler doesn't simply use the first scheduling opportunity it receives, but waits for up to a given maximum number of scheduling opportunities to occur before loosening the locality constraint

Anatomy of a MapReduce application run



Source: Tom White, "*Hadoop - The Definitive Guide (4 ed)*", O'Reilly, 2015

Fault Tolerance (I)

- Task Failure
  - The user code in the map or reduce task throws a runtime exception
    - the task JVM reports the error back to its parent application master before it exits.
    - The application master marks the task attempt as failed
    - The application master frees up the container
  - Sudden exit of the task JVM
    - the node manager informs the application master
    - The application master marks the task attempt as failed
    - The application master frees up the container
- Hanging tasks
  - The application master notices that it hasn’t received a progress update for a while
  - The task JVM process will be killed automatically after this period (typically 10 minutes)
  - The application master marks the task as failed
- When the application master is notified of a task attempt that has failed, it will reschedule execution of the task
- The application master will try to avoid rescheduling the task on a node manager where it has previously failed

Fault Tolerance (II)

- Application Master Failure
  - Application master periodically sends heartbeats to the resource manager.
  - The resource manager will detect the failure and start a new instance of the master running in a new container (managed by a node manager)
  - The client needs to go back to the resource manager to ask for the new application master's address (this process is transparent to the user)
  - If a MapReduce application master fails twice it will not be tried again and the job will fail
- Node Manager Failure
  - Node managers periodically send heartbeats to the resource manager
  - The resource manager will notice a node manager that has stopped sending heartbeats if it hasn't received one for 10 minutes
  - The resource manager will remove it from its pool of nodes to schedule containers on
  - Any task or application master running on the failed node manager will be considered failed and will be therefore recovered by using the mechanisms seen before
  - Node managers may be blacklisted if the number of failures for the application is high (even if node manager itself has not failed)
  - Blacklisting is done by the application master

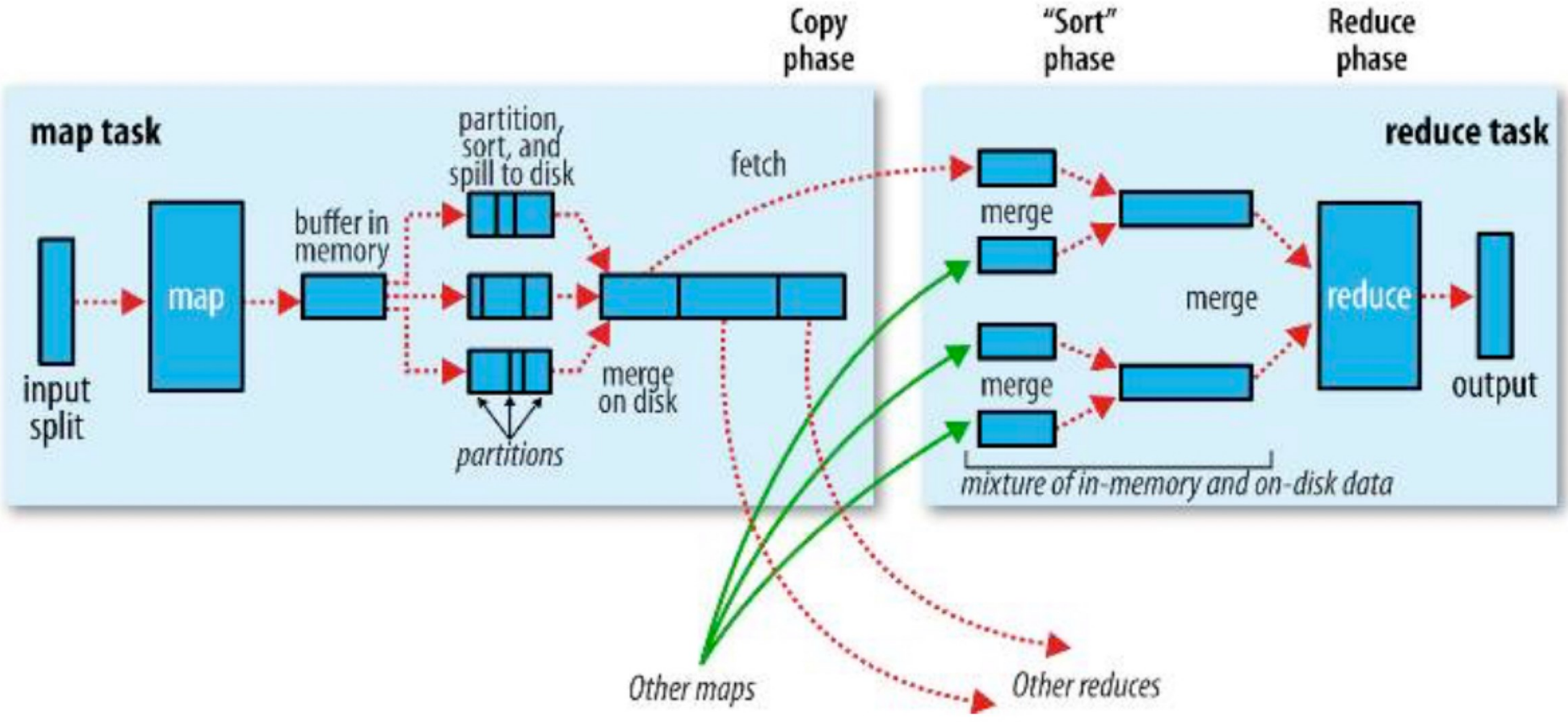
Fault Tolerance (III)

- Resource Manager Failure
  - Failure of the resource manager is serious: without it neither jobs nor task containers can be launched
  - In the default configuration, the resource manager is a single point of failure
  - To achieve high availability, it is necessary to run a pair of resource managers in an active-standby configuration
  - Information about all the running applications is stored in a highly available state store
  - The standby resource manager can recover the core state of the failed active resource manager
  - The transition of a resource manager from standby to active is handled by a failover controller
  - The default failover controller uses leader election mechanism to ensure that there is only a single active resource manager at one time
  - Clients and node managers must be configured to handle resource manager failover. They try connecting to each resource manager in a round-robin fashion until they find the active one.

Speculative Execution

- Problem: Stragglers (i.e., slow tasks) significantly lengthen the completion time of the job
- Tasks can be slow due to, e.g.,
  - Hardware degradation
  - Software misconfiguiration
- Solution: Speculative execution
  - Application master tracks the progress of all tasks of the same type (Map and Reduce) in a job
  - Launches speculative duplicate tasks for the small portion of tasks that are running significantly slower than the average
  - Whichever one (i.e., original and speculative) finishes first, “wins”. The other is killed.
- Additional cost: a few percent more resource usage

Shuffle and Sort



Source: Tom White, "*Hadoop - The Definitive Guide (4 ed)*", O'Reilly, 2015