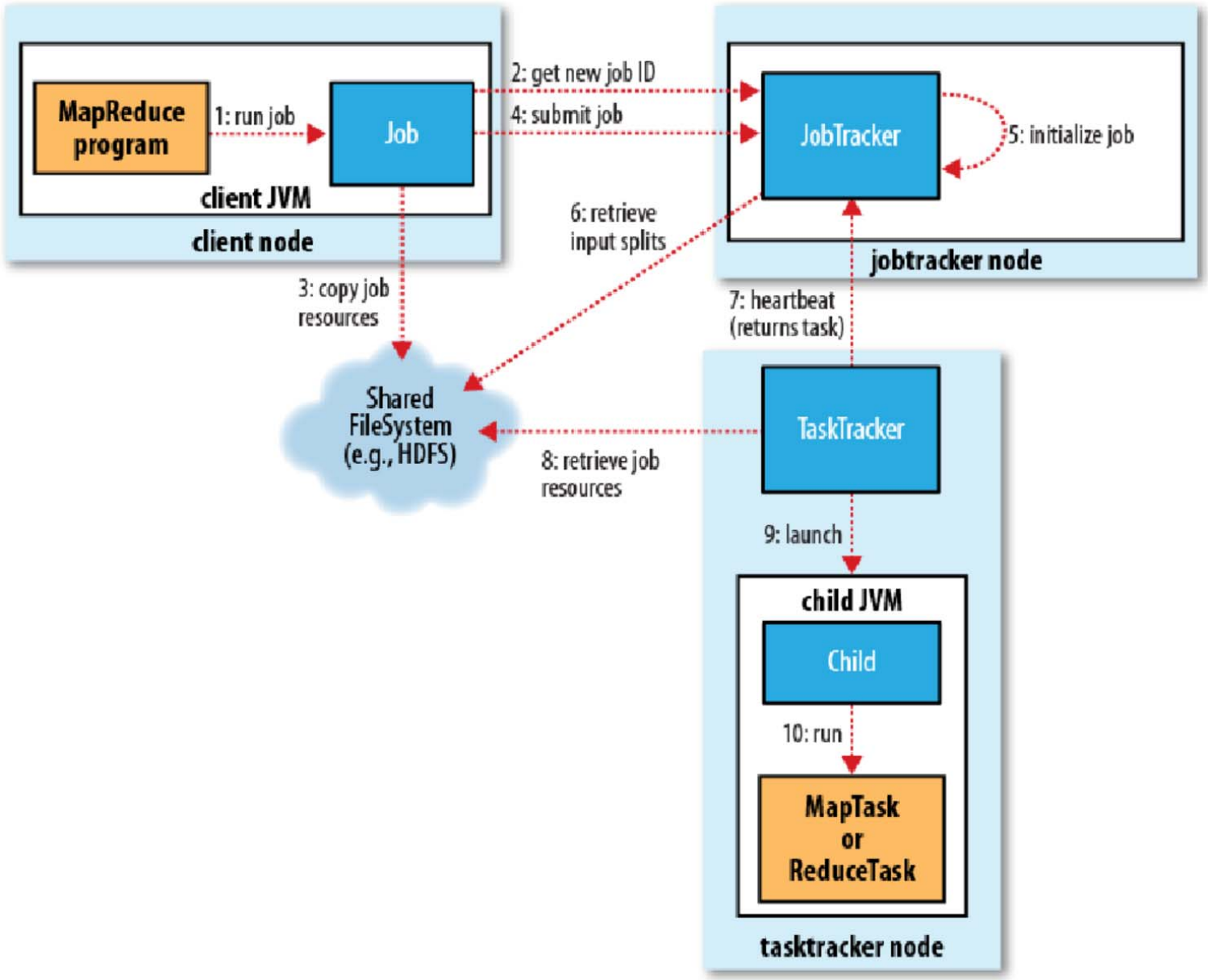
CSE535 Asynchronous Systems

Hadoop MapReduce in Details

YoungMin Kwon

Classic MapReduce (MapReduce 1)

- Four independent entities
  - Client: submits the MapReduce job
  - Jobtracker: coordinates the jobs
  - Tasktrackers: run the tasks that the job has been split into
  - Distributed filesystem (HDFS): sharing job files between other entities


MapReduce program
1: run job
Job
client JVM
client node
2: get new job ID
4: submit job
JobTracker
5: initialize job
jobtracker node
6: retrieve input splits
3: copy job resources
7: heartbeat (returns task)
Shared FileSystem (e.g., HDFS)
8: retrieve job resources
TaskTracker
9: launch
child JVM
Child
10: run
MapTask or ReduceTask
tasktracker node

MapReduce Job Run

- To run a MapReduce job
  - submit() call on Job object
  - waitForCompletion()
    - Submit a Job if not submitted already
    - Wait for the Job to finish
- Mapred.job.tracker property: determines where the jobtracker runs
  - Local
  - A colon separated pair of host and port

Job Submission

- submit method on Job (step 1) creates JobSumitter that does
  - Ask the jobtracker for a new job ID (getNewJobId()), (step 2)
  - Check the output specification for the job
  - Compute the input split for the job
  - Copy the job resources to the distributed filesystem (step 3)
    - Job JAR file, configuration file, input splits
  - Tell the jobtracker that the job is ready for execution (submitJob()), (step 4)

Job Initialization

- When submitJob() of JobTracker is called
  - It is enqueued for the job scheduler to pick up for the initialization
  - Initialization: creating an object representing the job encapsulating tasks, bookkeeping information for the status, and the progress of the tasks (step 5)
- The input splits are retrieved from the shared filesystem (step 6)
- Create tasks
  - A map task for each input split
  - Reduce tasks (mapred.reduce.tasks)
  - Job setup task and a job cleanup task

Task Assignment

- Tasktrackers send periodic heartbeats to the jobtracker
  - Heartbeat indicates whether the tasktracker is ready for a new task
  - Using the return value jobtracker informs the task to run (step 7)
- If a tasktracker has at least one empty map task slot, the jobtracker will select a map task; otherwise, it will select a reduce task
- Reduce task allocation doesn't consider data locality, but map task allocation does (data-local, rack-local)

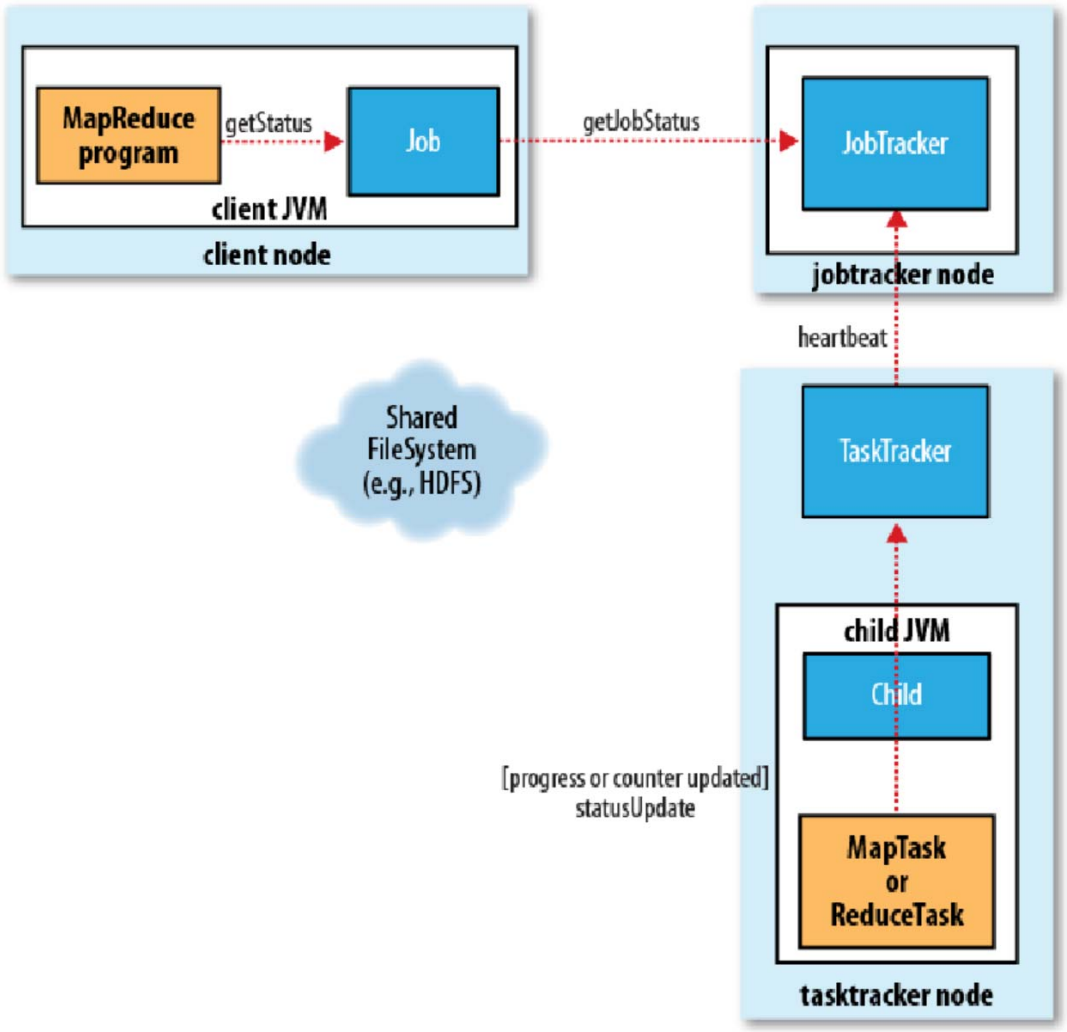
Task Execution

- Tasktracker runs its assigned task by
  - Get the resources (job JAR, configuration, …) from the shared filesystem (step 8)
  - Creates an instance of TaskRunner
- TaskRunner
  - Launches a new JVM (step 9) to run each task in (step 10)
  - It prevents bugs in user defined map/reduce functions affecting the tasktracker

Progress and Status Update

- A job and each of its tasks have
  - Status: running, failed, successfully completed
  - Progress
- Progress
  - Map task: the proportion of the input that has been processed
  - Reduce task: three parts corresponding the three phases of shuffle (copy, sort, reduce)

Progress and Status Update



Job Completion

- When the jobtracker receives a notification that the last task for the job is complete (from the job cleanup task), it changes the job to "successful"
- Returns from waitForCompletion() method

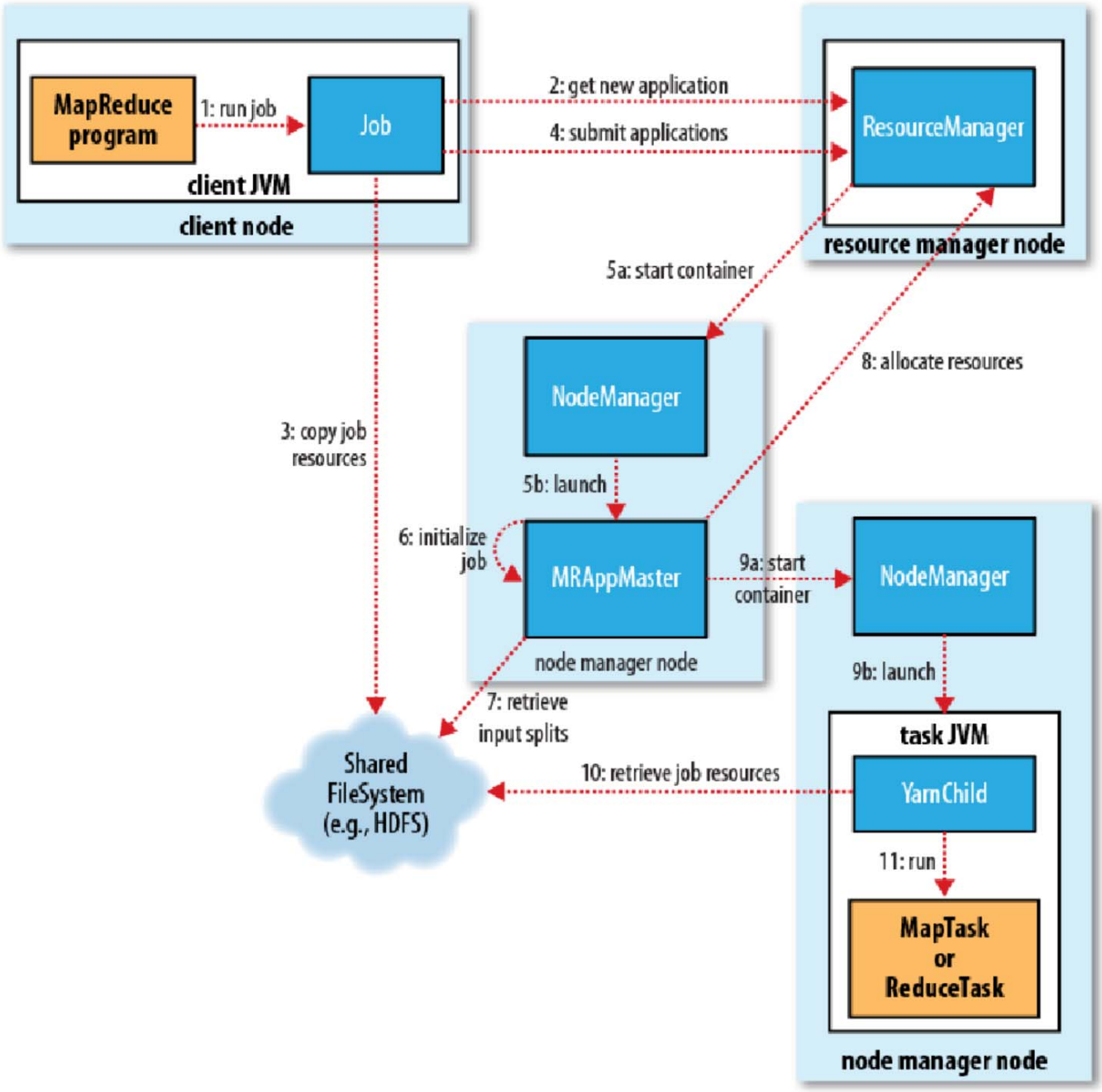
YARN (MapReduce 2)

- Scalability bottleneck
  - For very large clusters with more than 4000 nodes
  - Jobtracker takes care of both job scheduling and task progress monitoring

- Yet Another Resource Negotiator (YARN)
  - Separate the two roles of jobtracker
  - Resource manager: manages the use of resources across the cluster
  - Application master: manages the lifecycle of applications running on the cluster

YARN

- MapReduce entities in YARN
  - Client: submit the MapReduce job
  - Resource Manager (one per cluster): coordinates the allocation of compute resources on the cluster
  - Node manager (one per node): launches and monitors the compute containers on machines in the cluster
  - Application master: coordinates the tasks running the MapReduce job
    - Application master and MapReduce tasks run in containers scheduled by the resource manager and managed by the node manager
  - Distributed filesystem (HDFS): share job files


MapReduce program
1: run job
Job
client JVM
client node
2: get new application
4: submit applications
ResourceManager
resource manager node
5a: start container
NodeManager
5b: launch
8: allocate resources
3: copy job resources
6: initialize job
MRAppMaster
node manager node
9a: start container
NodeManager
9b: launch
task JVM
YarnChild
11: run
MapTask or ReduceTask
node manager node
7: retrieve input splits
Shared FileSystem (e.g., HDFS)
10: retrieve job resources

Job Submission

- Jobs are submitted using the same user API (submit) as MapReduce 1 (step 1)
- New job ID (application ID) is retrieved from the resource manager (step 2)
- The job client
  - Checks the output specification
  - Computes the input splits
  - Copies job resources (job JAR, configuration, split information) to HDFS (step 3)
- Job is submitted to the resource manager (step 4)

Job Initialization

- Scheduler launches the application master
  - submitApplication() call to the resource manager is handed off to the scheduler
  - The scheduler allocates a container (step 5a)
  - The node manager launches the application master process (step 5b)
- The application master
  - Initializes the job by creating a number of bookkeeping objects (step 6)
  - Retrieves the input splits from the shared filesystem (step 7)

Task Assignment

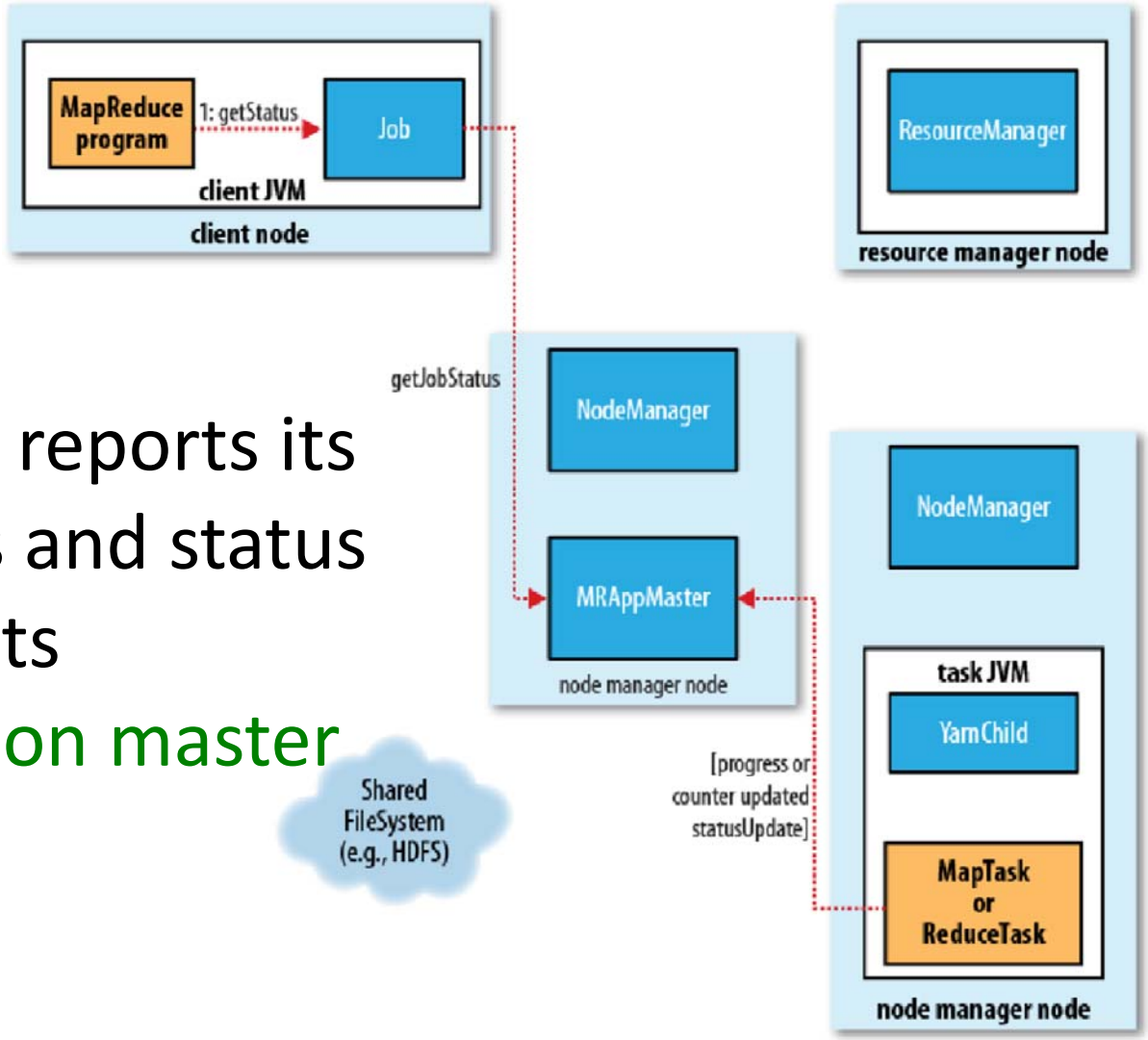
- The application master
  - If the job is small, the application master may choose to run the tasks in the same JVM as itself
  - Requests containers for all tasks from the resource manager (step 8)
  - The request has information about each map task's data locality
    - Hosts and corresponding racks that the input split resides on

Task Execution

- The application manager starts a container by contacting the node manager (step 9a)
- The node manager launches a JVM and starts YarnChild (step 9b)
- YarnChild retrieves resources (step 10)
  - Including job JAR file, configuration
- It runs the map or the reduce task (step 11)

Progress and Status Updates

- The task reports its progress and status back to its application master

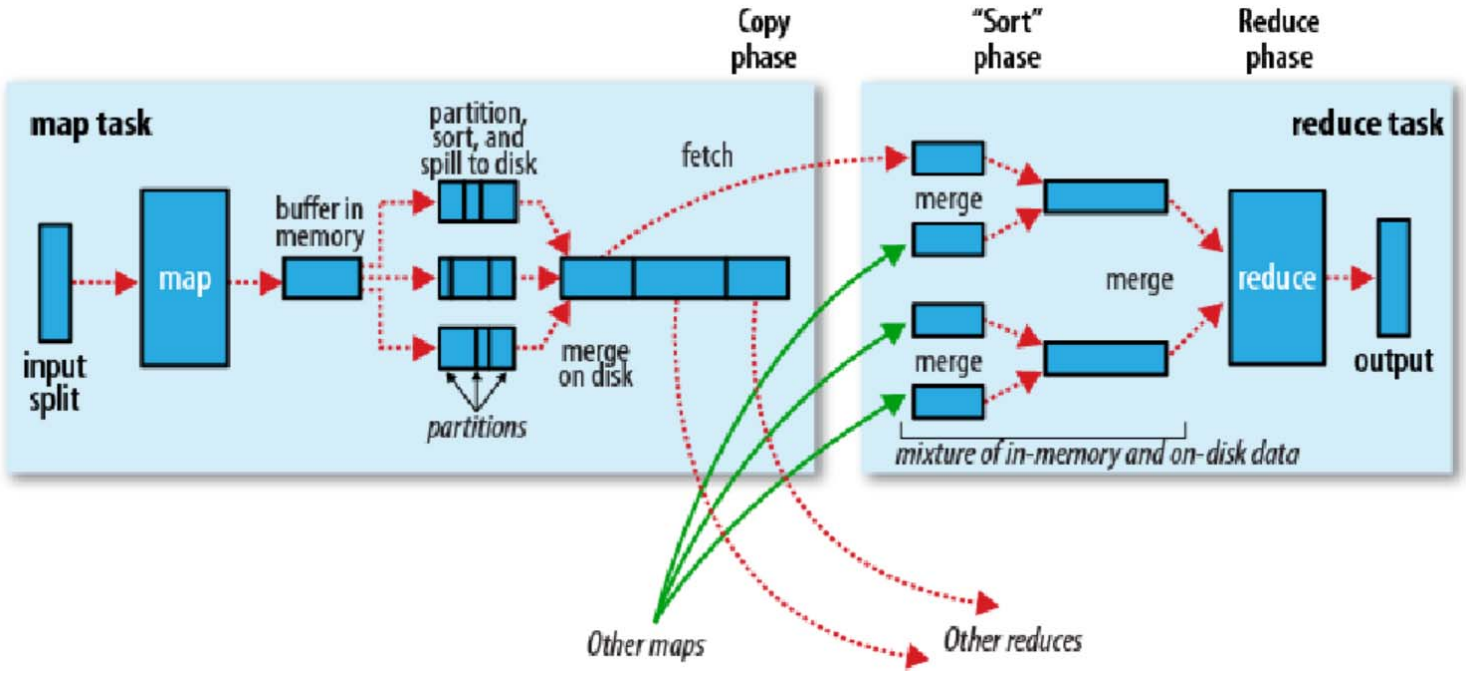


Job Completion

- Every 5 sec, the client checks whether the job has completed and the progress by polling the application master
- On job completion, the application master and the task containers clean up their work state and call OutputCommitter's job cleanup method

Shuffle and Sort

- MapReduce guarantees that the input to reducer is sorted by key
- The process that performs the sort is known as the shuffle



The Map Side

- Each map task has a circular memory buffer that it writes its output to
- When the buffer reaches a threshold, it spills the contents to disk
  - Before writing to disk, the data is partitioned for the reducers
  - Each partition is sorted in memory
  - Each time the buffer is spilled, a new spill file is created
- The spill files are merged into a partitioned-and-sorted output file.

The Reduce Side

- Reduce task starts copying its partition in the map output files as soon as they are generated (copy phase)
- Once all partitions are copied, sort them using the merge sort (sort phase)
- The final round of merge is not saved on disk; the reduce function runs directly on the merged results in memory (reduce phase)

Output Committers

- Hadoop MapReduce uses a commit protocol to ensure that jobs and tasks either succeed or fail cleanly
  - If the job succeeds, the commitJob() is called
  - If the job did not succeed, the abortJob() is called
  - If a task succeeds, commitTask() is called
  - Otherwise, abortTask() is called
  - needsTaskCommit() can disable the commit phase for tasks