L4,7,8: MR Runtime Hadoop and HDFS

Yogesh Simmhan

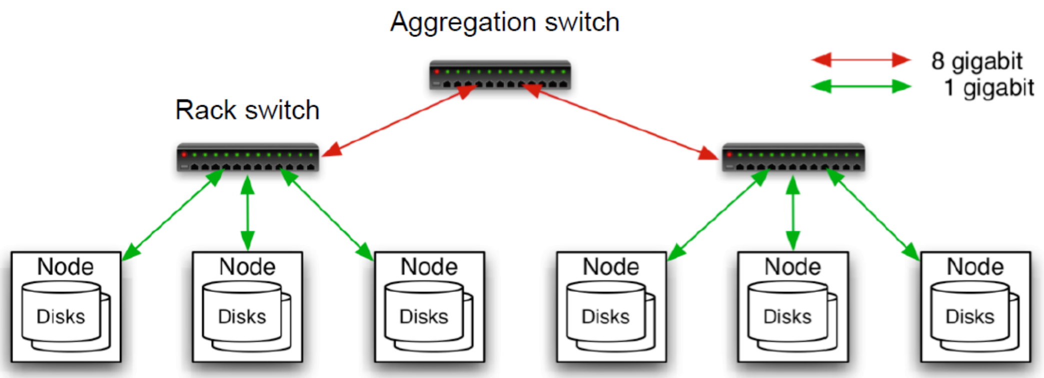
2 Feb, 2017

Learning Objectives

1. *How* does **HDFS** work? *Why* is it effective?
2. *How* does Hadoop **MapReduce** work? *Why* is it effective?
3. Optimizations for performance and reliability in Hadoop MR

Data Centre Architecture *Recap*

- Commodity hardware
  - 1000's machines of medium performance and reliability
  - Failure is a given. Design to withstand failure.
- Network bottlenecks
  - Hierarchical network design
  - Push compute to data



Data Centre Architecture *Recap*

- I/O bottlenecks & failure
  - Multiple disks for cumulative bandwidth
  - Data redundancy: Hot/Hot
- Example: How long to read 1TB of data?
  - HDD at 100 MB/s … 2.7hrs
  - SSD at 400 MB/s … 41 mins
- Can you do faster?
  - 2 SSD per machine, 500GB each … 20mins
  - 20 SSD per machine, 50 GB each … 2 mins?
    - *20 mins!* SATA Speed is ~800 MB/s
  - Say cluster with 10 nodes, 1 Gbps Ethernet, 8 HDD each, reading over network … 2 mins!
- *Time to read across network is not very different from time to read from stressed disk*

GrayWulf, Scalable Clustered Architecture for Data Intensive Computing, Szalay, HICSS, 2008

E.g. Open Cloud Server

- High density: 24 blades / chassis, 96 blades / rack
- Compute blades
  - Dual socket, 4 HDD, 4 SSD
  - 16-32 CPU cores
  - 4-16TB HDD/SSD
- JBOD Blade
  - 10 to 80 HDDs, 6G or 12G SAS
  - 40-160TB HDD

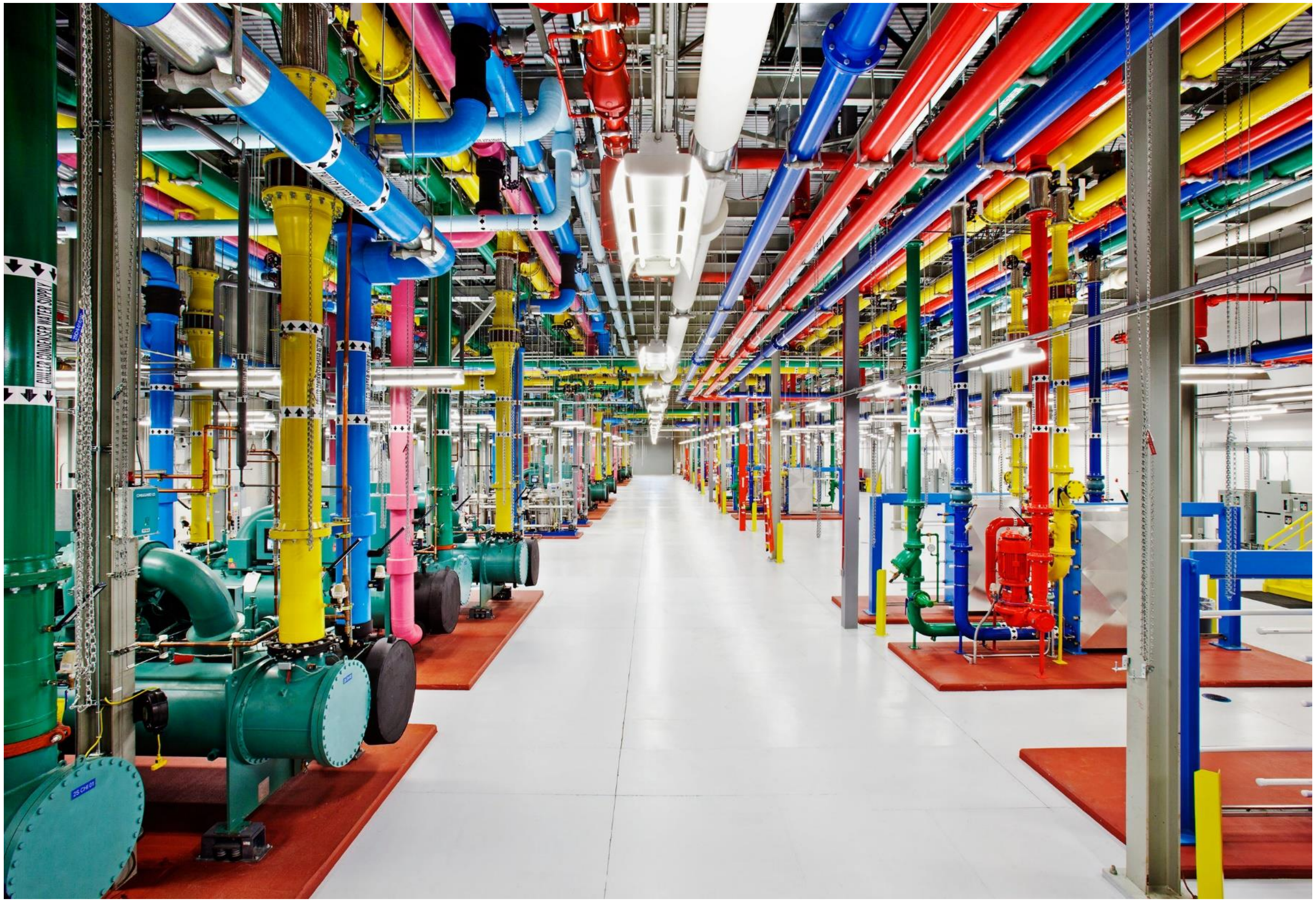
http://www.opencompute.org/wiki/Motherboard/SpecsAndDesigns

Class Cluster

- Nodes
  - 8 core AMD Opteron 3380, 2.6GHz
  - 32GB DDR3
  - 2TB HDD
  - 1Gbps LAN
- 12 nodes, 3U
- 1 Gigabit within switch, 10Gbps across switches

http://www.supermicro.com/Aplus/system/3U/3012/AS-3012MA-H12TRF.cfm

Cisco's Data Center in Texas



Google's Data Center in Georgia

Microsoft's Data Center in Ireland

NSA’s Data Center in Utah

Who is this?

Doug Cutting and *Hadoop* the elephant

Hadoop was created by Doug Cutting (Yahoo) and Mike Cafarella (UW) in 2006.
Cutting's son, then 2, was just beginning to talk and called his beloved stuffed yellow elephant "Hadoop" (with the stress on the first syllable).

http://www.cnbc.com/id/100769719
https://en.wikipedia.org/wiki/Apache_Hadoop#History

Hadoop: Big Picture Interactions

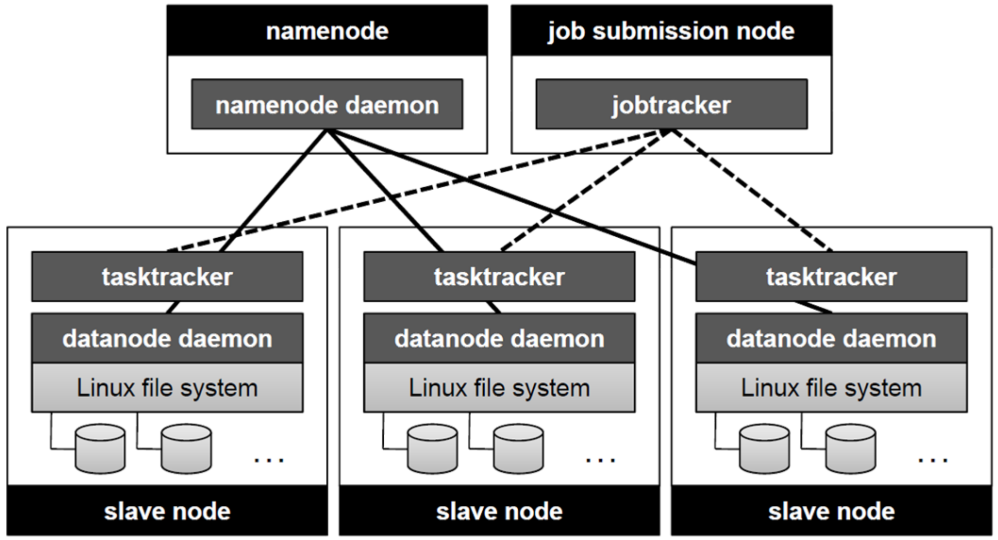


Figure 2.6: Architecture of a complete Hadoop cluster, which consists of three separate components: the HDFS master (called the namenode), the job submission node (called the jobtracker), and many slave nodes (three shown here). Each of the slave nodes runs a tasktracker for executing map and reduce tasks and a datanode daemon for serving HDFS data.

Hadoop: Big Picture Interactions

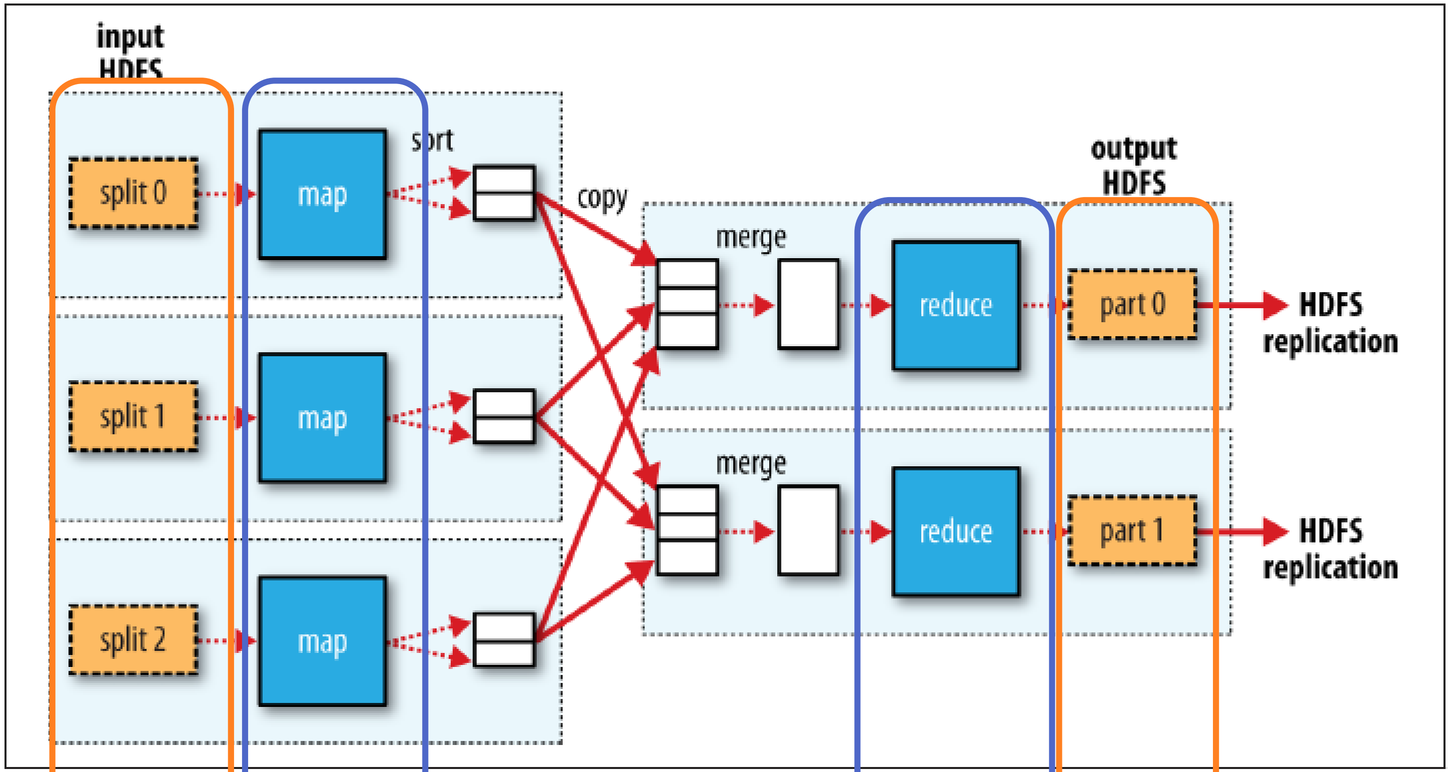


Figure 2-4. MapReduce data flow with multiple reduce tasks

Hadoop Distributed File System

Chapter 3, Tom White, 4th Ed.

Hadoop Distributed File System (HDFS)

- Based on Google File System (GFS)
- Optimized for huge files
- Write once, read many
  - Create new data. Never update-in-place, only *append*.
  - No write locks (only 1 writer!). Initial write-cost is amortized.
- Optimized for sequential reads
  - Typically, start at a point and read to completion
- Throughput favoured over low latency
  - Low total time for reading all data, than time per small files
- Survive high disk/node failures
  - Both persistence, availability

The Google File System, Sanjay Ghemawat, et al, SOSP, 2003

HDFS File Distribution

- Files are split into *blocks* of equal size
  - Unit of data that can be read or written
- Block sizes are large
  - e.g. 128MB…*configurable per file*
- Blocks themselves are persisted on local disks
  - e.g. using POSIX file system
  - Only use as much disk space as block content, i.e. 1MB content in 128MB block
- Blocks are replicated
  - Default 3x…*configurable per file…e.g. high for "hot" files*
  - Blocks on "lost" disk can be re-created

HDFS File Distribution

- Larger reads/writes
  - *Time to read 1*1GB vs. 1000*1MB files?*
- Files can be larger than single disk
  - Distributed across nodes
- Eases distributed management
  - Same size, opaque content, complexity pushed up.
  - Unit of recovery, replication
  - Separate data (blocks) from metadata

HDFS Design

- Master-slave architecture
  - Master manages namespace, directory/file names/tree structure, metadata, block ids, permissions
  - Slave manages blocks containing data

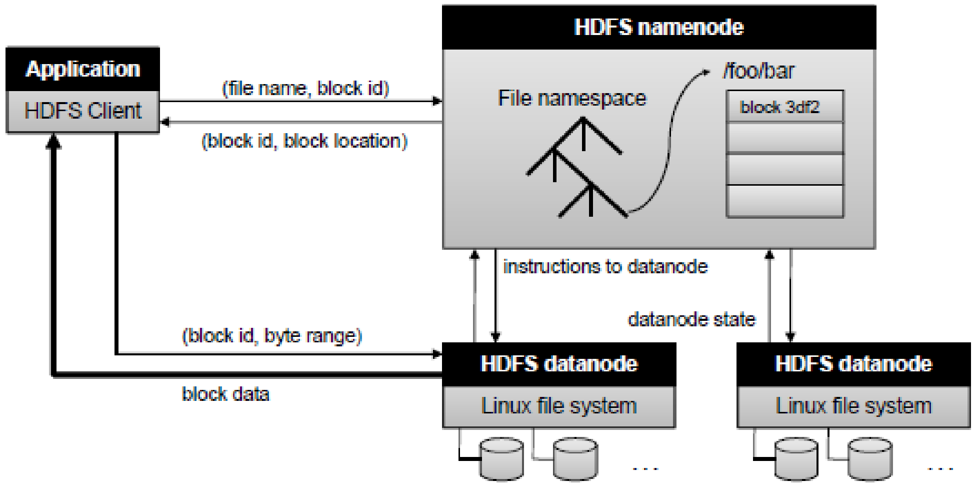


Figure 2.5: The architecture of HDFS. The namenode (master) is responsible for maintaining the file namespace and directing clients to datanodes (slaves) that actually hold data blocks containing user data.

Data-Intensive Text Processing with MapReduce, Jimmy Lin and Chris Dyer, 2010

Master: Name Node

- Persists names, trees, metadata, permissions
  - Namespace image (fsimage), cached in-memory
  - Edit log of deltas (rename, permission, create)
    - Transaction persisted on disk, then applied to in-memory fsimage
  - fsimage and edit log merged on disk when HDFS restarted
  - Mapping from files to list of blocks
- Block location not persistent, kept in-memory
  - Mapping from blocks to locations is *dynamic*
    - *Why?*
  - *Reconstructs* location of blocks from data nodes
  - ~150 bytes of in-memory metdata per block/file/dir

Master: Name Node

- Detects health of FS
  - Is data node alive?
  - Is data block under-replicated?
  - Rebalancing block allocation across data nodes, improved disk utilization
- Coordinates file operations
  - Directs application clients to datanodes for reads
  - Allocates blocks on datanodes for writes
- Security is not a priority
  - Basic file and dir permissions (rwx)
  - Default enforcement relies on client machine ‘username’

Master: Name Node

- File system does no work if NameNode not accessible!
- Single Point of failure! *(Hadoop 1.x)*
  - Cold start → 10mins load FS image, 1hr for block list for every file
  - Upgrades → Downtime
  - Host recovery → Copy FS image, config data node
  - Disk Failure → Data loss *(file names, file:block ID mapping)*
- Sync atomic writes to multiple disk file systems
  - Local disk+NFS
- Secondary NameNode
  - Merge NS image with edit log periodically…avoids downtime when merging
  - Serves as stale copy of NS image…but data loss possible

http://blog.cloudera.com/blog/2012/03/high-availability-for-the-hadoop-distributed-file-system-hdfs/

Secondary Name Node

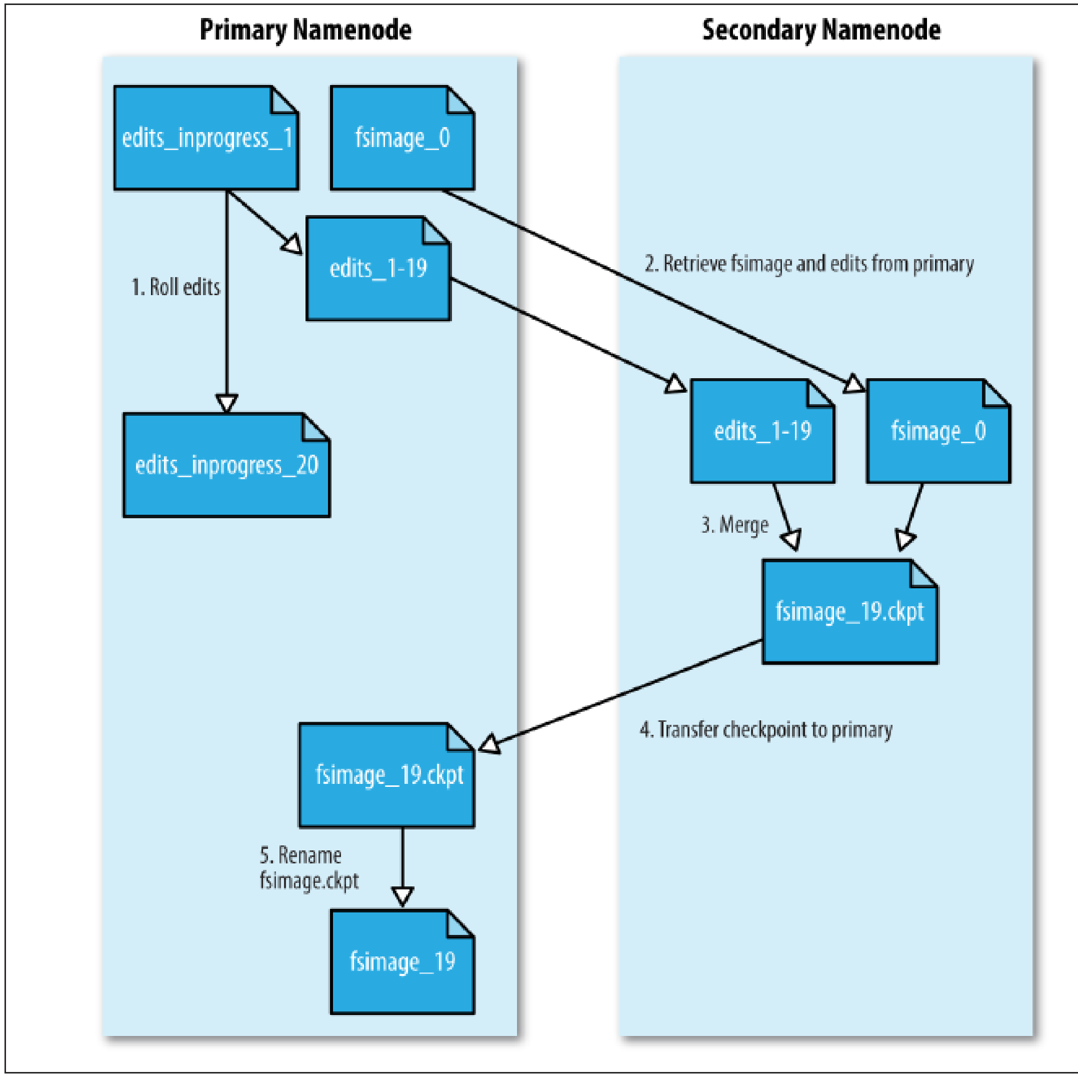
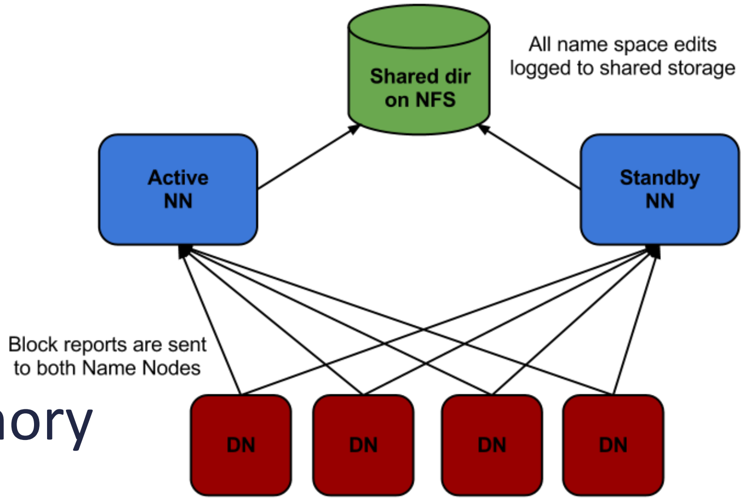


Figure 11-1. The checkpointing process

Hadoop: The Definitive Guide, Tom White, 4th **Edition**, 2015

Name Node


Shared dir on NFS
All name space edits logged to shared storage
Active NN
Standby NN
Block reports are sent to both Name Nodes
DN
DN
DN
DN

- NameNode High Availability *(2.x)*
  - Reliable shared NFS for edit log
  - Hot standby loads NS image in-memory
  - Constantly reads edit logs from disk
  - DataNodes send heartbeat, block list to both
    - But ops received only from active
  - On NameNode failover, standby can takeover immediately
- NameNode Federation (2.x)
  - Distributes NS volumes (dir paths) on different NameNodes
  - Reduces memory footprint for NS image, block pool
  - Independent of each other

http://blog.cloudera.com/blog/2012/03/high-availability-for-the-hadoop-distributed-file-system-hdfs/

Slave/Worker: Data Node

- Store & retrieve blocks
- Respond to client and master requests for block operations
- Sends heartbeat every 3 secs for liveliness
- Periodically sends list of block IDs and location on that node
  - Piggyback on heartbeat message
  - e.g., send block list every hour
- *Caches blocks in-memory* using cache-directives per file, on single data node
  - E.g. index, lookup table, etc.
  - Can be used by schedulers

File Reads

- Client-Data Node direct transfer…*Not* through the Name Node
- Client gets data node list for each block from NameNode
  - First few blocks returned initially, Sorted by distance
- Blocks read in order
  - Connection opened and closed to nearest DataNode for each block
  - Tries alternate data nodes on network failure, checksum failure
  - Remembers & reports failures/corrupt blocks to Name Node
- Allows scaling to many concurrent clients

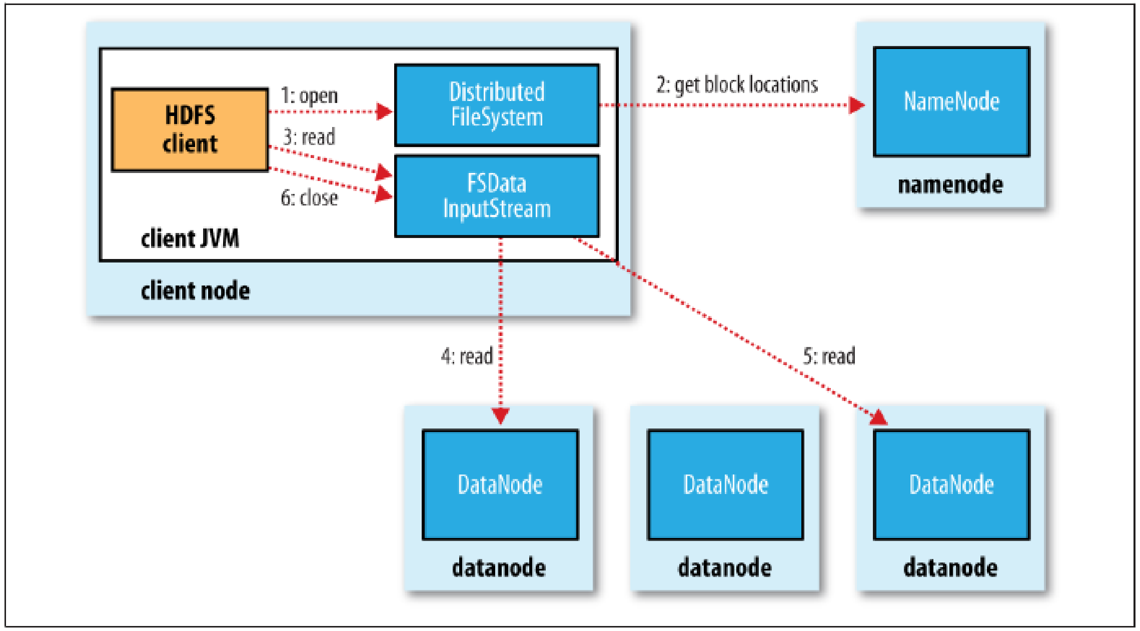


Figure 3-2. A client reading data from HDFS

Network Topology

- Same Node, Same Rack, Same Data Center, Different Data Centers
- Distance function between two logical nodes provided in config
  - /dc/rack/node … default is “flat”, i.e. same distance

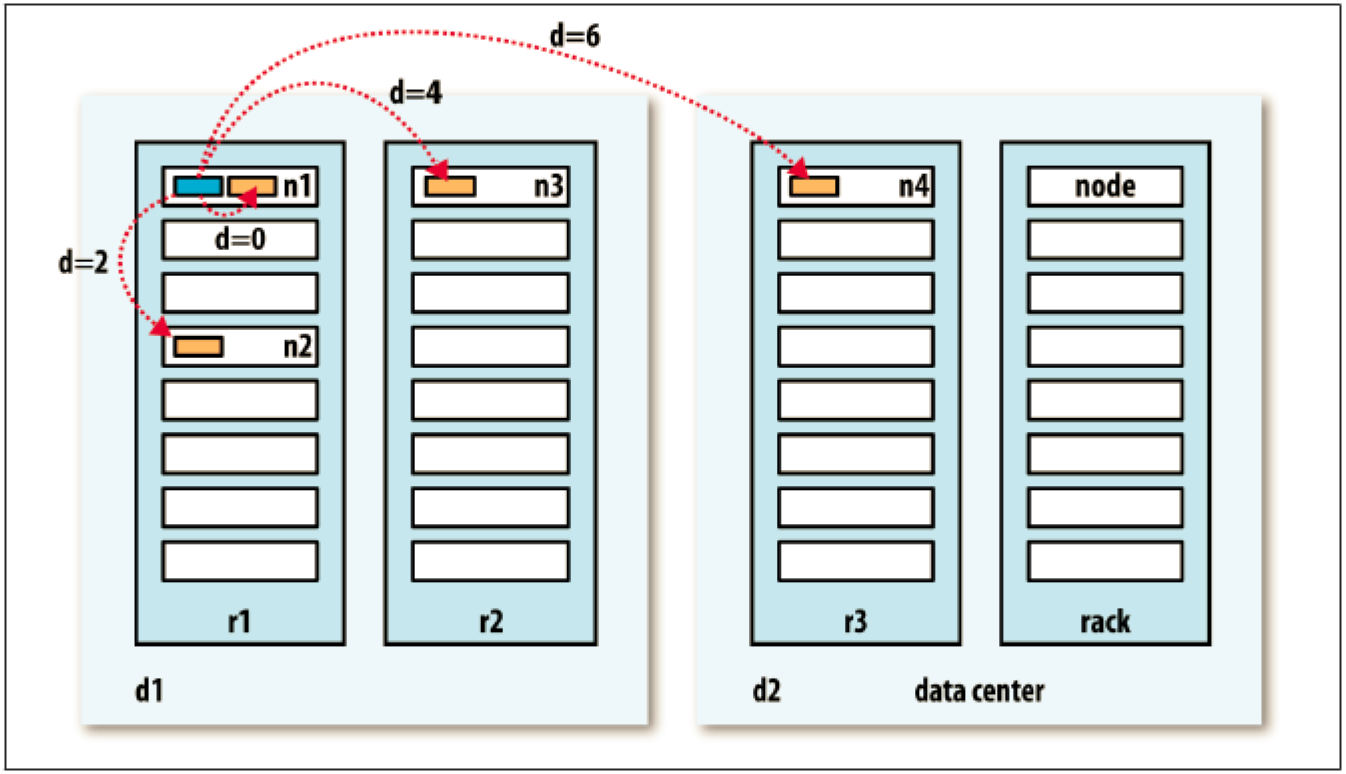
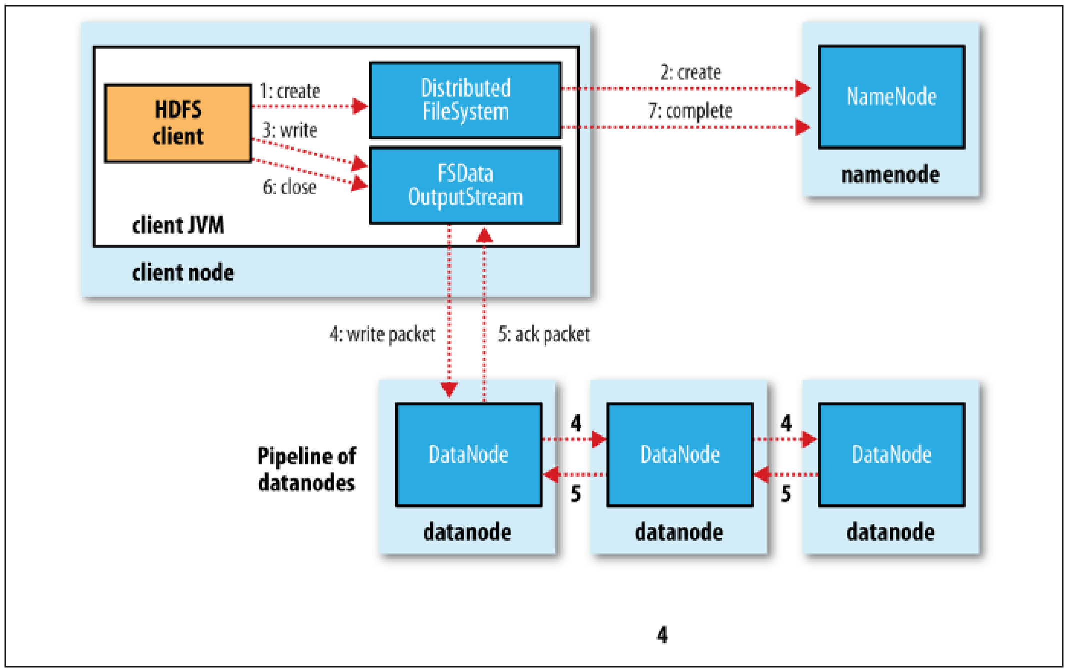


Figure 3-3. Network distance in Hadoop

File Writes

- Write one only…Append, Truncate…Strict one writer at a time, per file
- Clients get list of data nodes to store a block's replica
  - First copy on same data node as client, or random.
  - Second is off-rack. Third on same rack as second.
- Blocks written in order. Forwarded in a pipeline. Acks from all replicas expected before next block written.



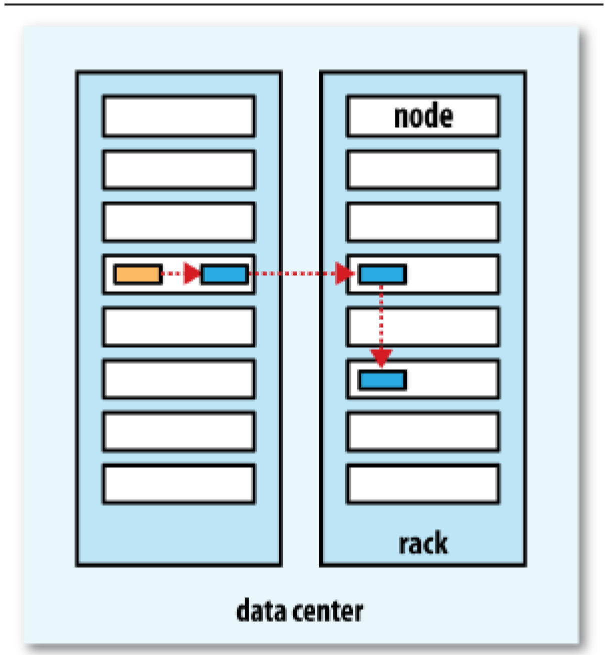


Figure 3-4. A client writing data to HDFS

Hadoop YARN

Yet Another Resource Negotiator

MapReduce v1 → MapReduce v2 (YARN)

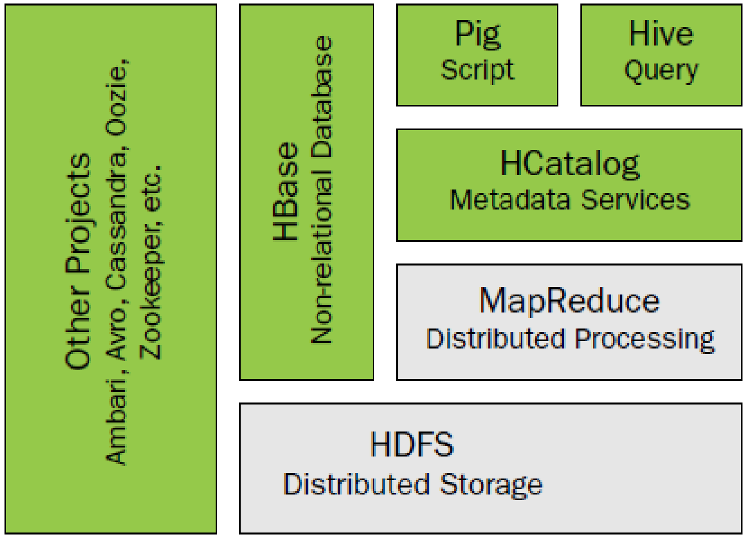


Figure 3.1 The Hadoop 1.0 ecosystem. MapReduce and HDFS are the core components, while other components are built around the core.

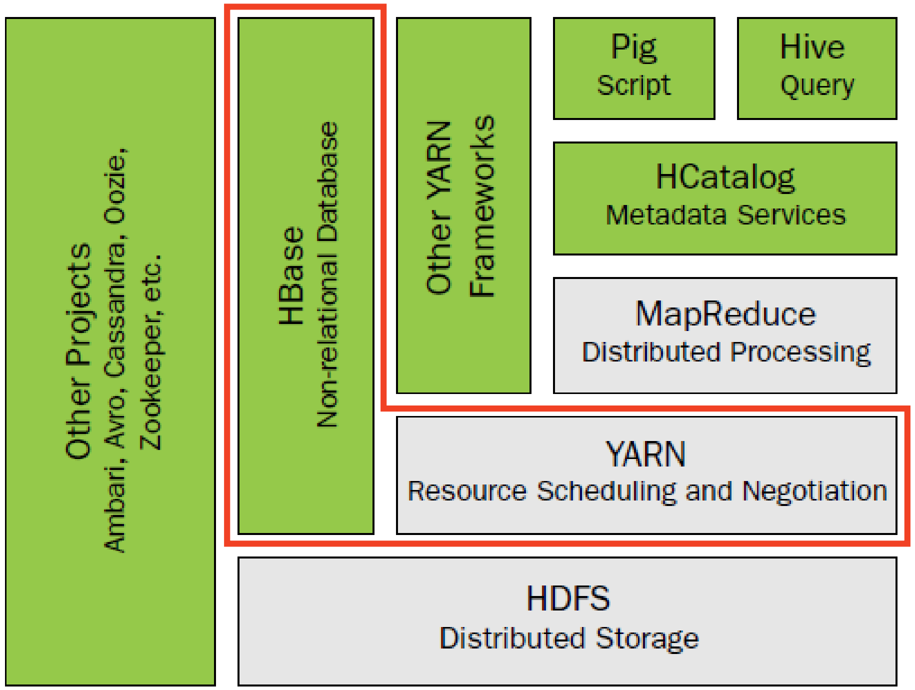


Figure 3.2 YARN adds a more general interface to run non-MapReduce jobs within the Hadoop framework

YARN

- Designed for scalability
  - 10k nodes, 400k tasks
- Designed for availability
  - Separate application management from resource management
- Improve utilization
  - Flexible slot allocation. Slots not bound to Map or Reduce types.
- Go beyond MapReduce

YARN

- ResourceManager for cluster
  - Keeps track of nodes, capacities, allocations
  - Failure and recovery (heartbeats)
- Coordinates scheduling of jobs on the cluster
  - Decides which node to allocate to a job
  - Ensures load balancing
- Used by programming frameworks to schedule distributed applications
  - MapReduce, Spark, etc.
- NodeManager
  - Offers slots with given capacity on a host to schedule tasks
  - Container maps to one or more slots…Container can be a Unix process or cgroup

Application Manager

- Coordinates
  - resource acquisition,
  - scheduling,
  - monitoring progress ,
  - and termination
  - for a specific application type
- E.g. MapReduce, MPI, Spark, etc.
- AppManager runs in its own container
  - May launch additional containers for its compute tasks
  - Or may run job locally in JVM for “small” applications

YARN Application Lifecycle

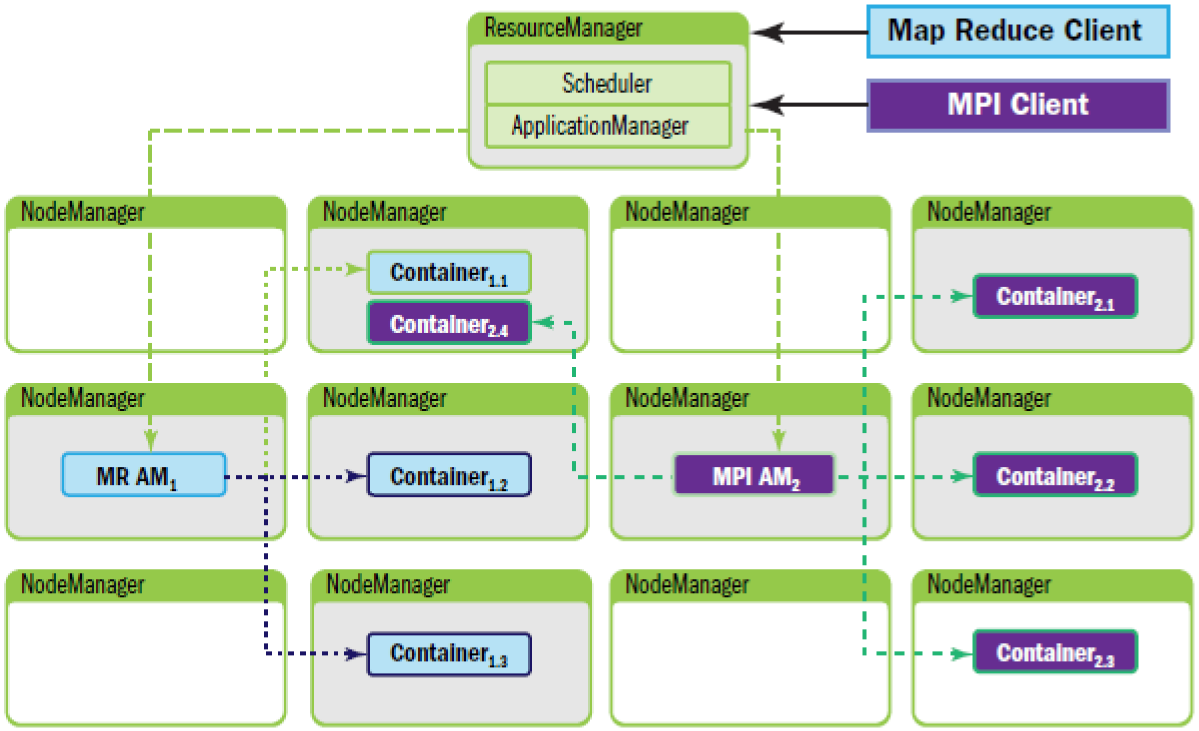


Figure 4.1 YARN architecture with two clients (MapReduce and MPI). The client MPI AM$_2$ is running an MPI application and the client MR AM$_1$ is running a MapReduce application.

v1 vs. v2 Application Lifecycle

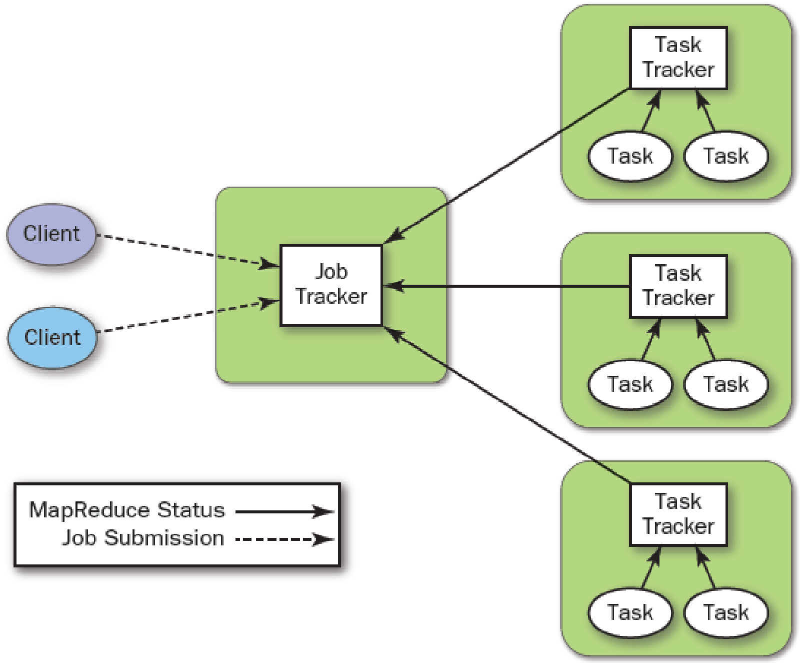


Figure 3.3 Current Hadoop MapReduce control elements

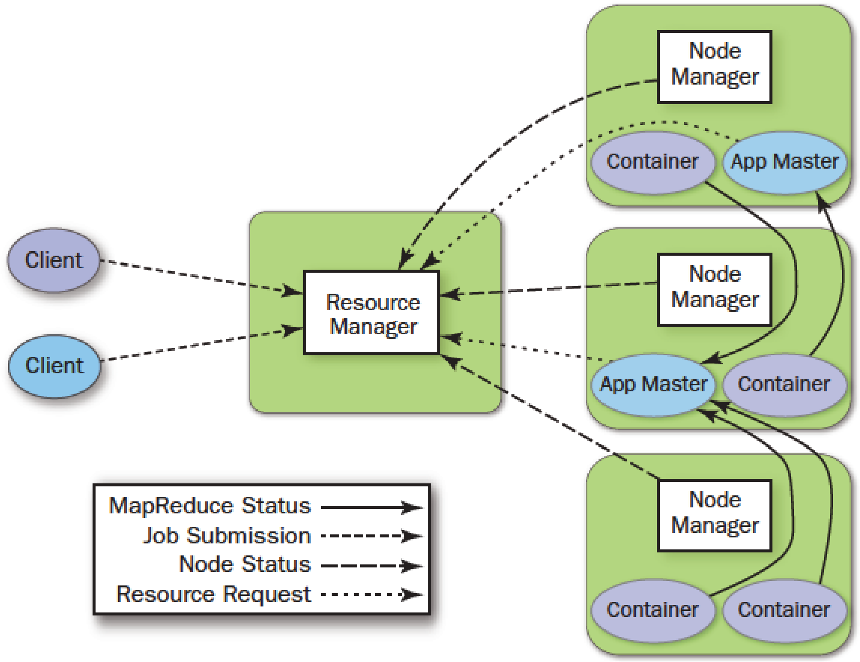


Figure 3.4 New YARN control elements

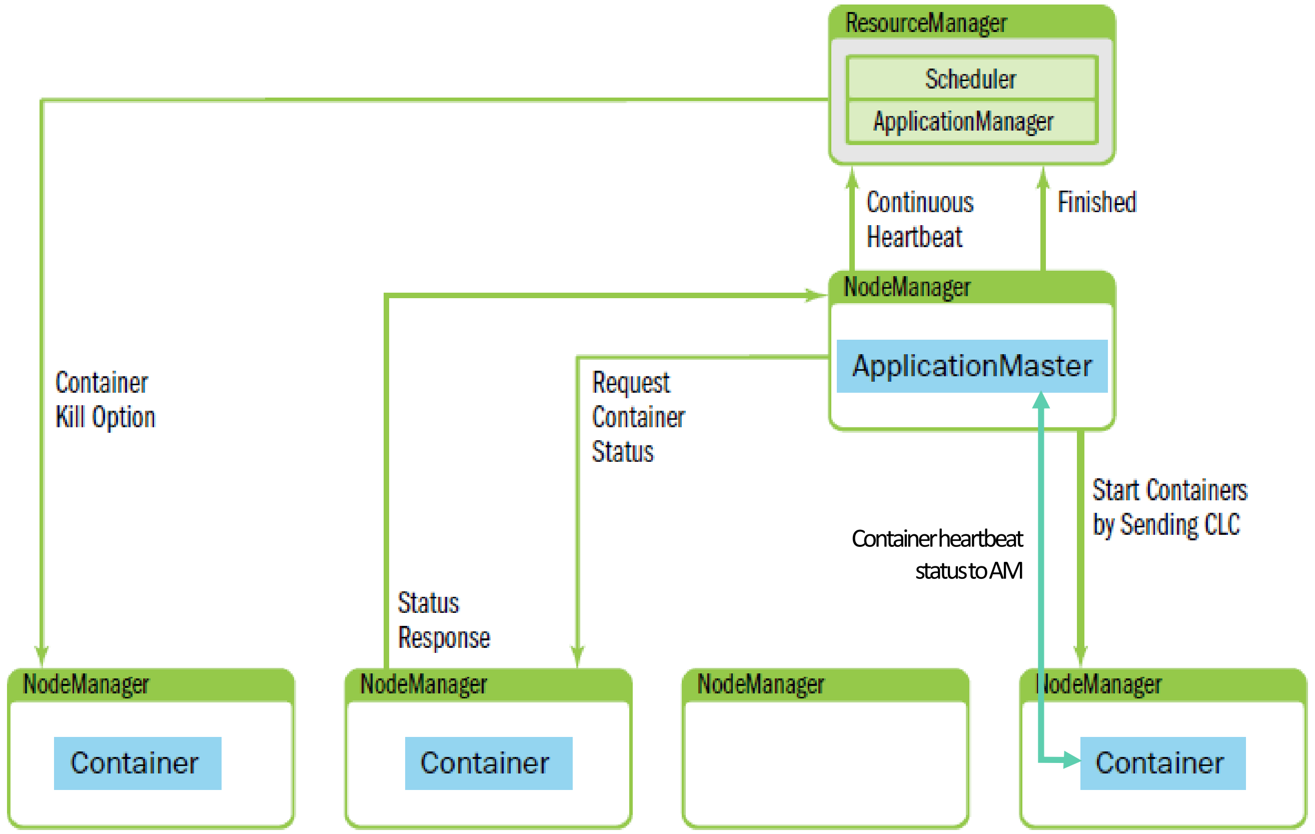


Figure 4.3 ApplicationMaster NodeManager interaction.

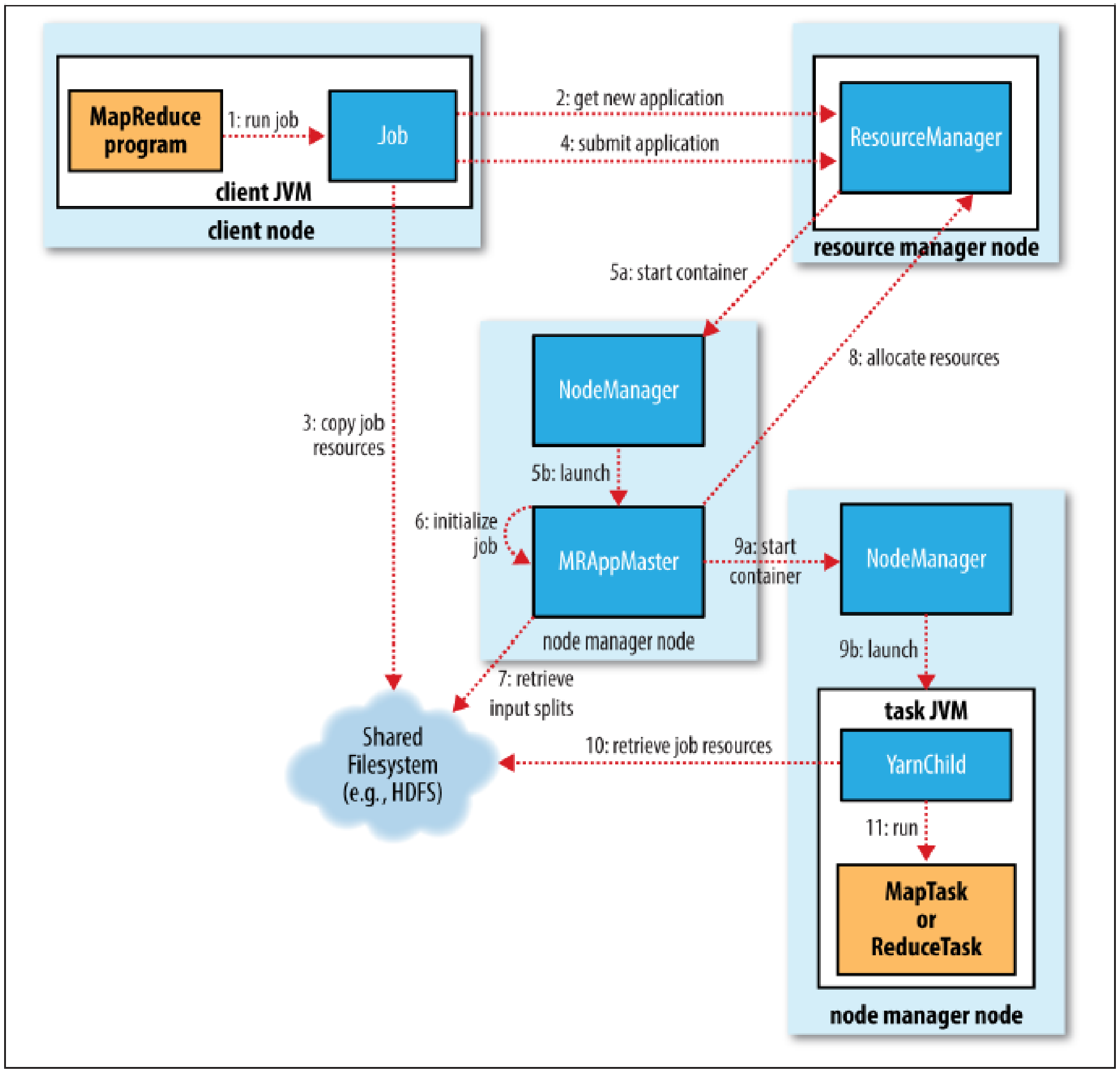


Figure 7-1. How Hadoop runs a MapReduce job

MapReduce AppManager

- First requests Map containers
  - As many as number of splits
- Reduce containers requested after 5% Map tasks complete
  - User specified. *1 by default!*
- Map containers try for data locality as “split”
  - Same node, Same rack
- Containers have CPU and Memory resource requirements
  - Config per job, or default for cluster
- AppManager asks Node Manager to start container
  - Container task fetches jar, config locally, executes, commits

Scheduling in YARN

- Scheduler has narrow mandate
- FIFO, as soon as resource available
- Capacity
  - using different queues, min capacity per queue
  - Allocate excess resource to more loaded
- Fair
  - Give all available
  - Redistribute as jobs arrive

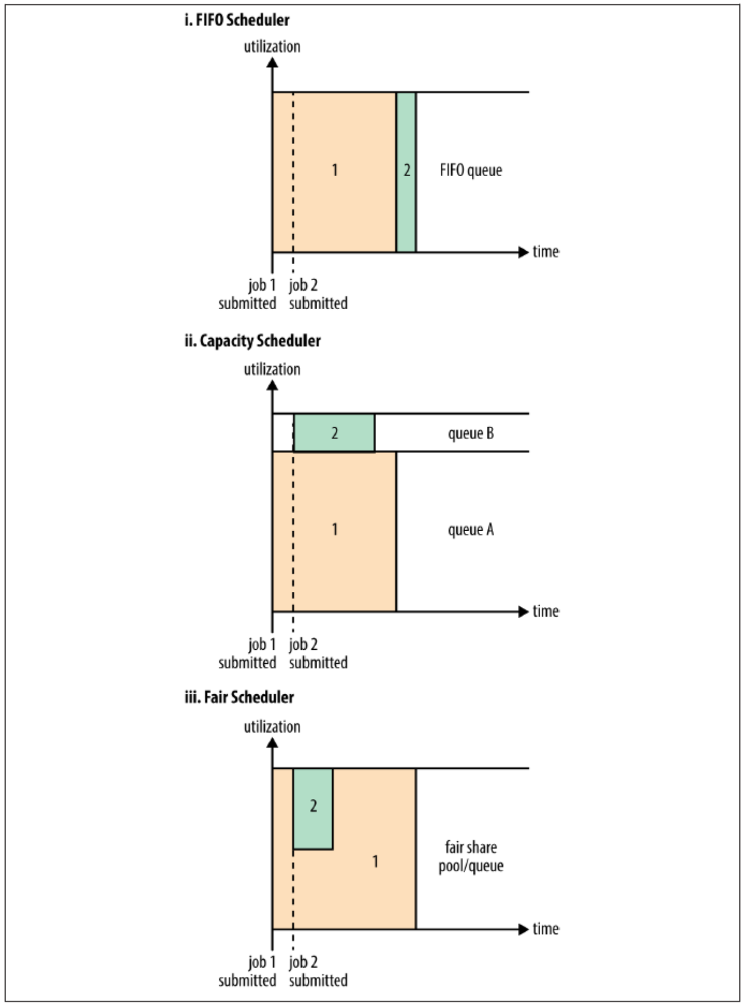


Figure 4-3. Cluster utilization over time when running a large job and a small job under the FIFO Scheduler (i), Capacity Scheduler (ii), and Fair Scheduler (iii)

Hadoop: The Definitive Guide, 4th **Edition**, 2015

Hadoop MapReduce

Mapping tasks to blocks

- FileInputFormat converts blocks to "*splits*"
  - Typically, 1 split per block … reduce task creation overhead vs. overwhelm single task
  - Can specify splits smaller/larger than a block size
  - 'sync' record to ensure logical boundaries
  - Affects locality if spanning blocks
  - Affects performance with many small files (combine!)
- Each split handled by a single Mapper task
  - *Records* read from each split, forms Key-Value pair input to Map function

Resource Mapping

- Resource acquisition either at beginning (Map tasks) or during (Reduce tasks) application lifetime
  - Higher priority for Map container requests
- AppManager can specify locality constraints to YARN
  - Compute tasks are moved to data *block* location
  - Location of one of three replicas of block
  - Prefer same node, followed by rack, then cluster

Mapping tasks to blocks

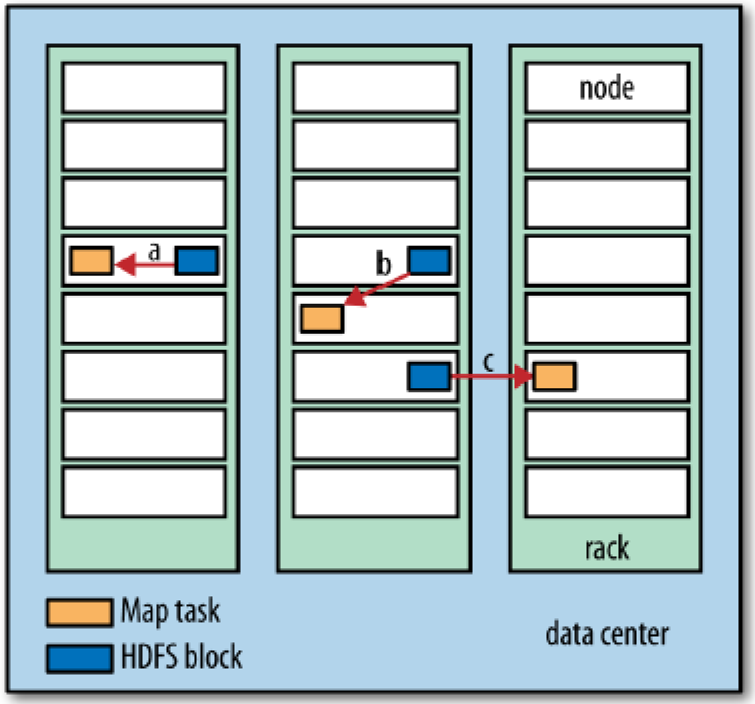


Figure 2-2. Data-local (a), rack-local (b), and off-rack (c) map tasks

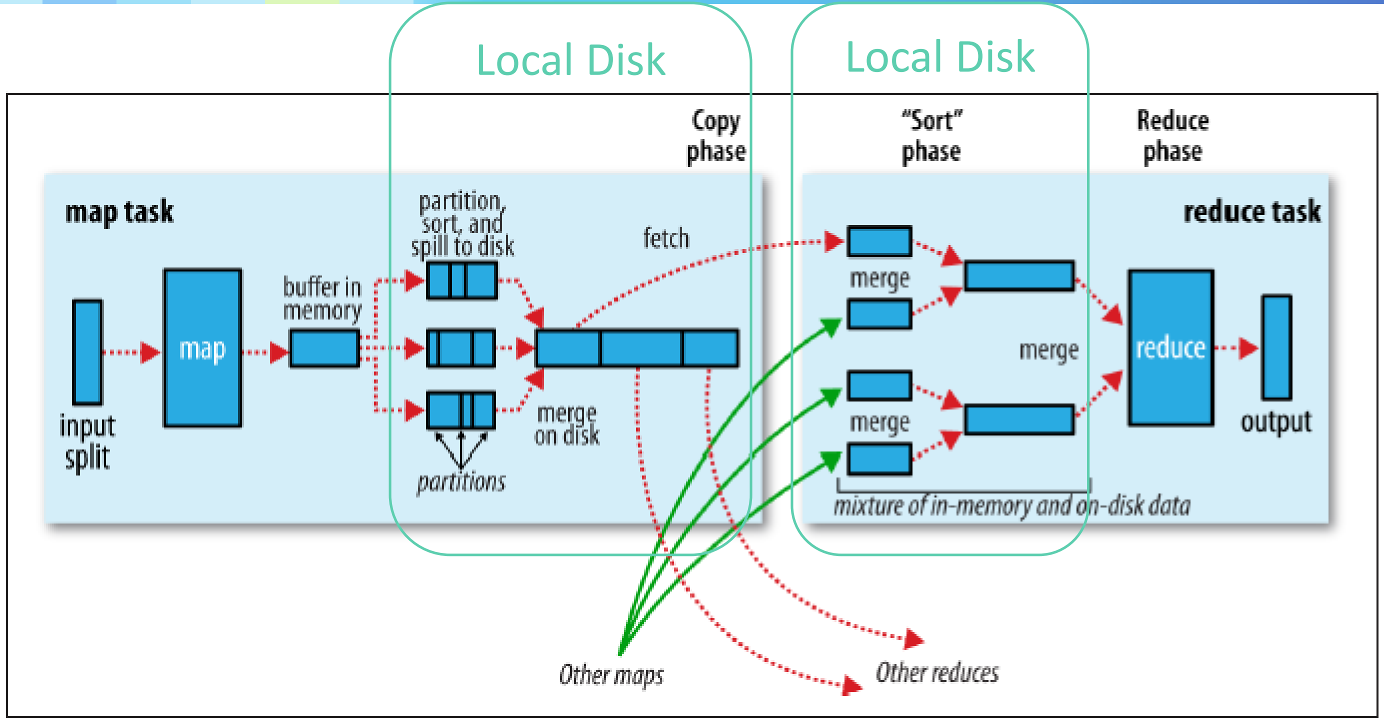


Figure 7-4. Shuffle and sort in MapReduce

- Background thread "spills" to disk when circular memory buffer (100MB) threshold reached (80%)
  - Asynchronous, avoid blocking unless thread write slower than Map task
- Divides the data into in-memory partitions, one for each reducer
  - Performs sort by key
  - Runs combiner sorted outputs
  - Writes to local directory, accessible by reducers over HTTPS (Not HDFS!)

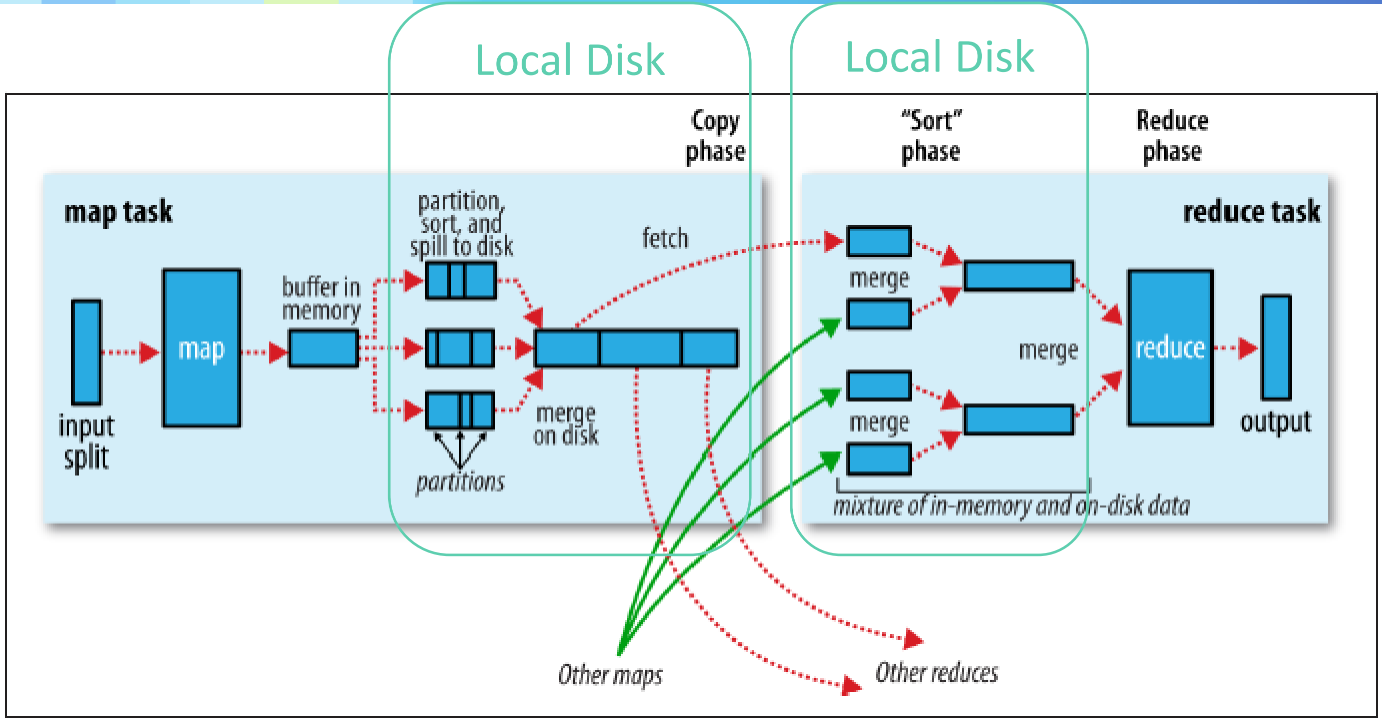


Figure 7-4. Shuffle and sort in MapReduce

- Output files are merged, partitioned and sorted into single file on disk
  - If multiple spill files (3) once Map task done, runs combiner again.
  - Optionally compressed
- Map task output always written to disk…recovery!

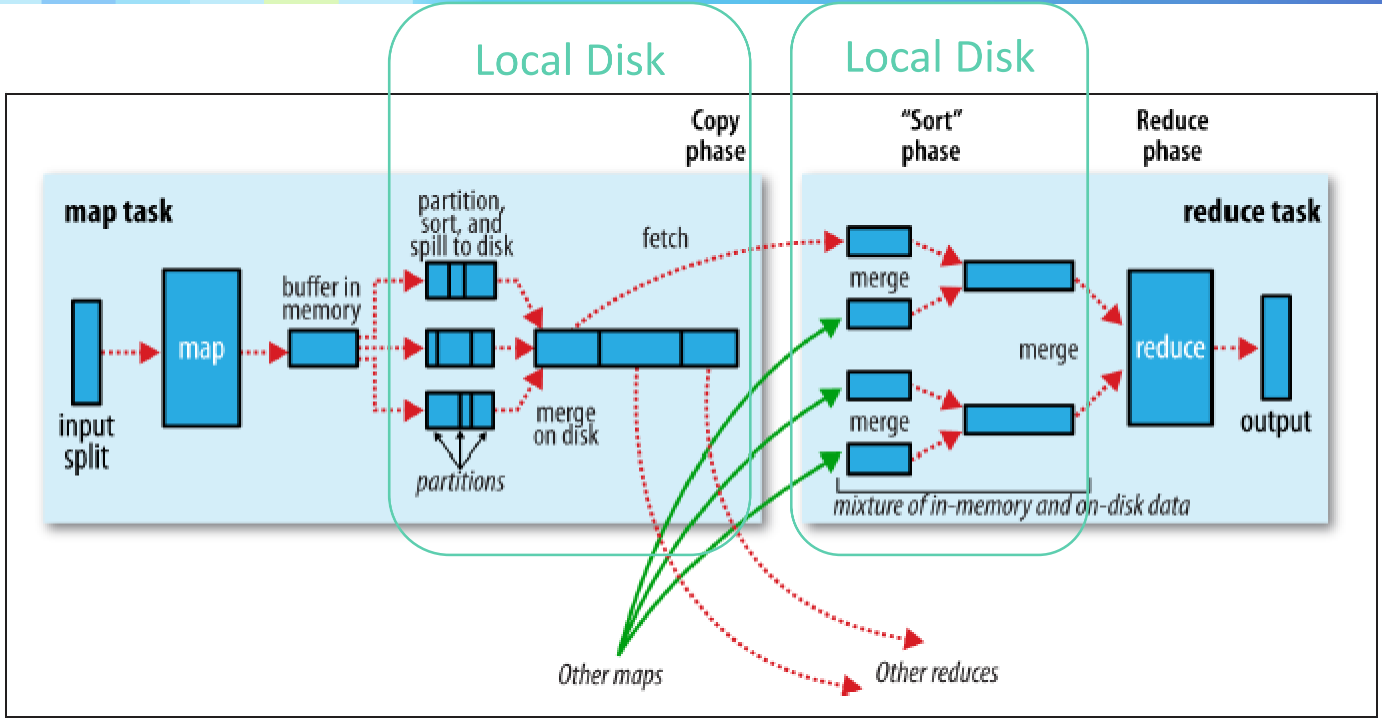


Figure 7-4. Shuffle and sort in MapReduce

- Reducer copies files as soon as available from any Map task
  - Copied to reducer memory if small,
  - On threshold: Merged , Combiner then spilled to disk
- Incremental merge sort takes place in background thread

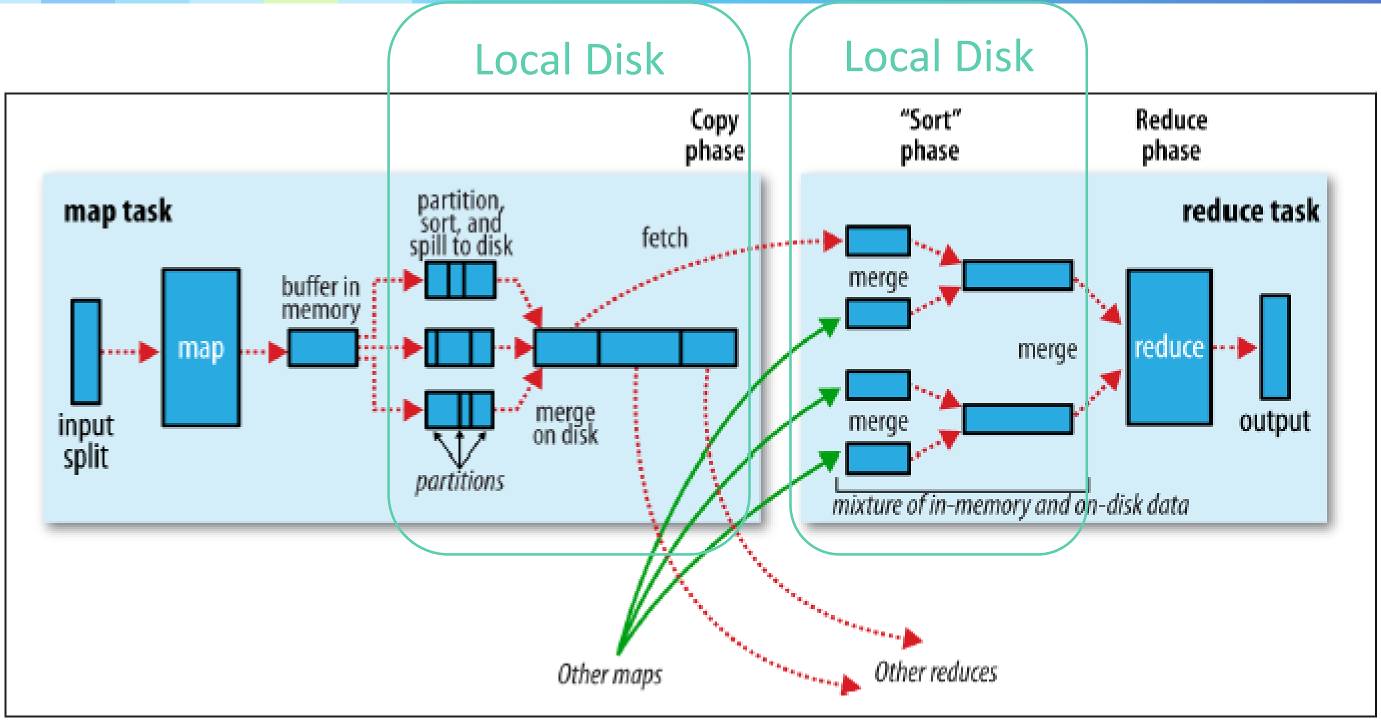


Figure 7-4. Shuffle and sort in MapReduce

- When output from all Map tasks available, final Merge-sort over all spilled files, before reduce method called
  - Multiple rounds, 10 files merged per round
  - Input to reducer from sorted file and trailing in-memory sorted KVP

Liveliness

- A Hadoop job or task is alive as long as it is making progress
  - Reading/writing input record
  - Setting status or incrementing counter
- Progress reported to App-Manager by Tasks ~3secs
- Client polls AppManager
  - ~1 sec

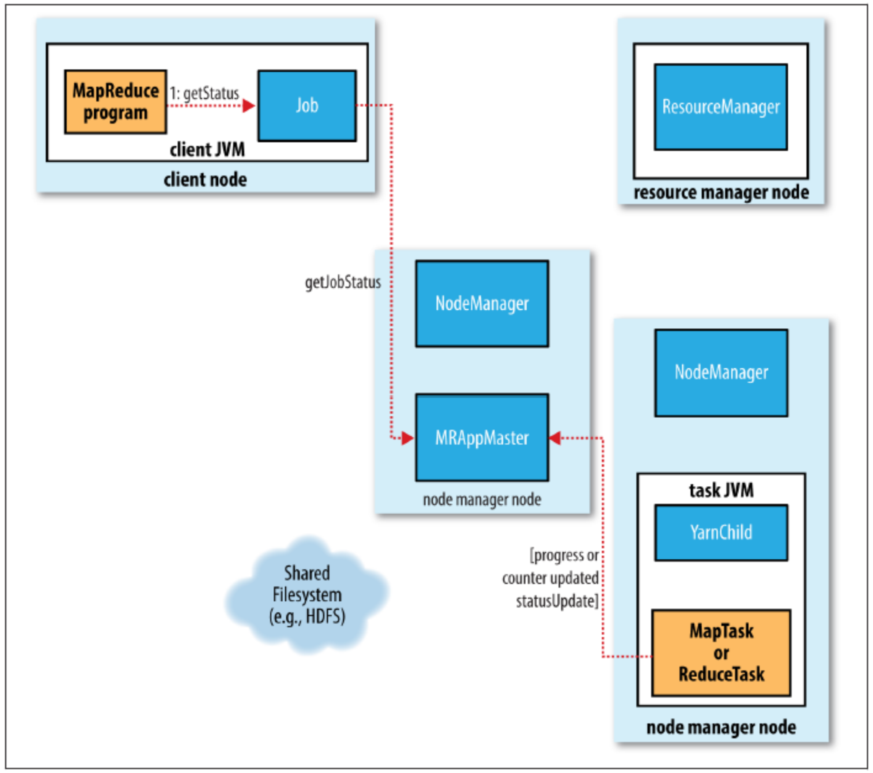


Figure 7-3. How status updates are propagated through the MapReduce system

Fault Tolerance

- Idempotent, “side-effect free”
- Save data to local disk before reduce
- Task crash & recover
- Node crash & recover
- Skipping bad records

Fault Tolerance

- Task logic fails: AppManager notified
- JVM fails: NodeManager notified AppManager
- Hanging: Timeout for progress update
  - AppManager marks task as failed, releases container
  - Retries, on different Node, typically 4 times
- Sometimes, Job can be considered success even if some tasks fail

Speculative execution

- Weakest link can slot things down…Stragglers
- Tasks that are slower than average completed tasks
- Duplicate task killed once one succeeds
- Improves utilization on exclusive cluster
- Wastes resources on a shared cluster
- Only works for idempotent tasks
- Does not help if task fails

Reading

- Hadoop: The Definitive Guide, **4th Edition**, 2015
  - **Chapters 3, 4, 7**

Additional Resources

- Apache Hadoop YARN: Moving Beyond MapReduce and Batch Processing with Apache Hadoop, 2015
  - **Chapters 1, 3, 4, 7**

Guest Lecture

- **Feb 25 (Saturday), 3PM**
- **Dr. Manish Gupta, Microsoft**
  - Azure Big Data Platform and ML Service