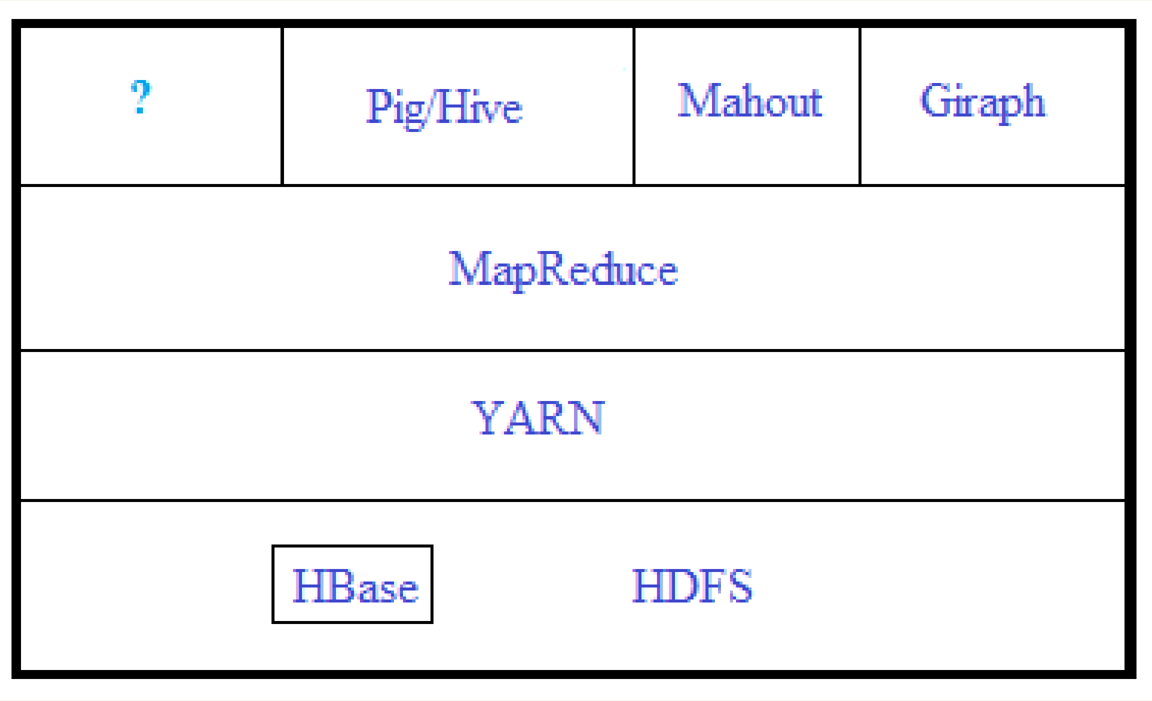
Hadoop

Rathinaraja Jeyaraj

Hadoop Big Data analytics stack



?

Pig/Hive

Mahout

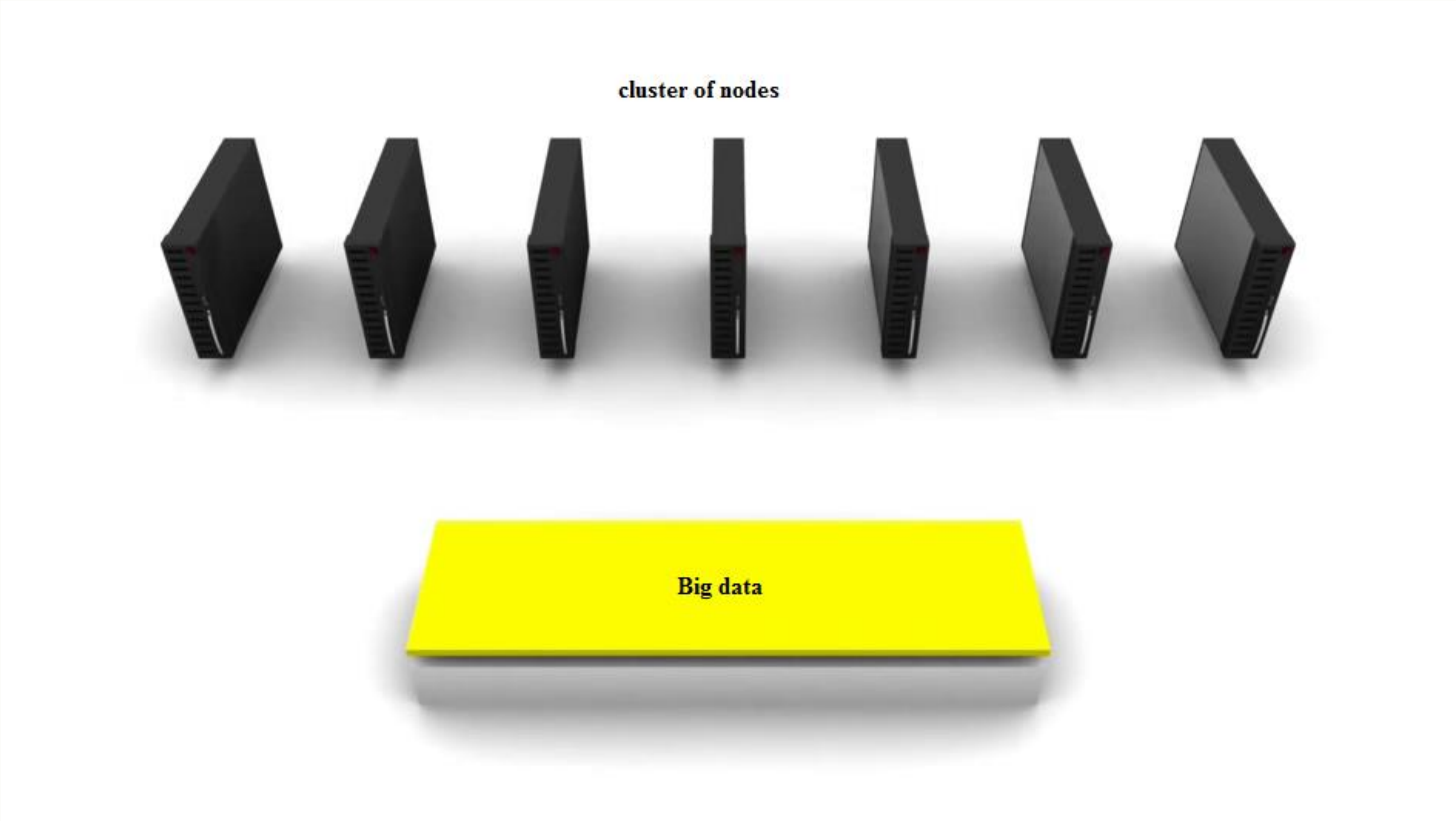
Giraph

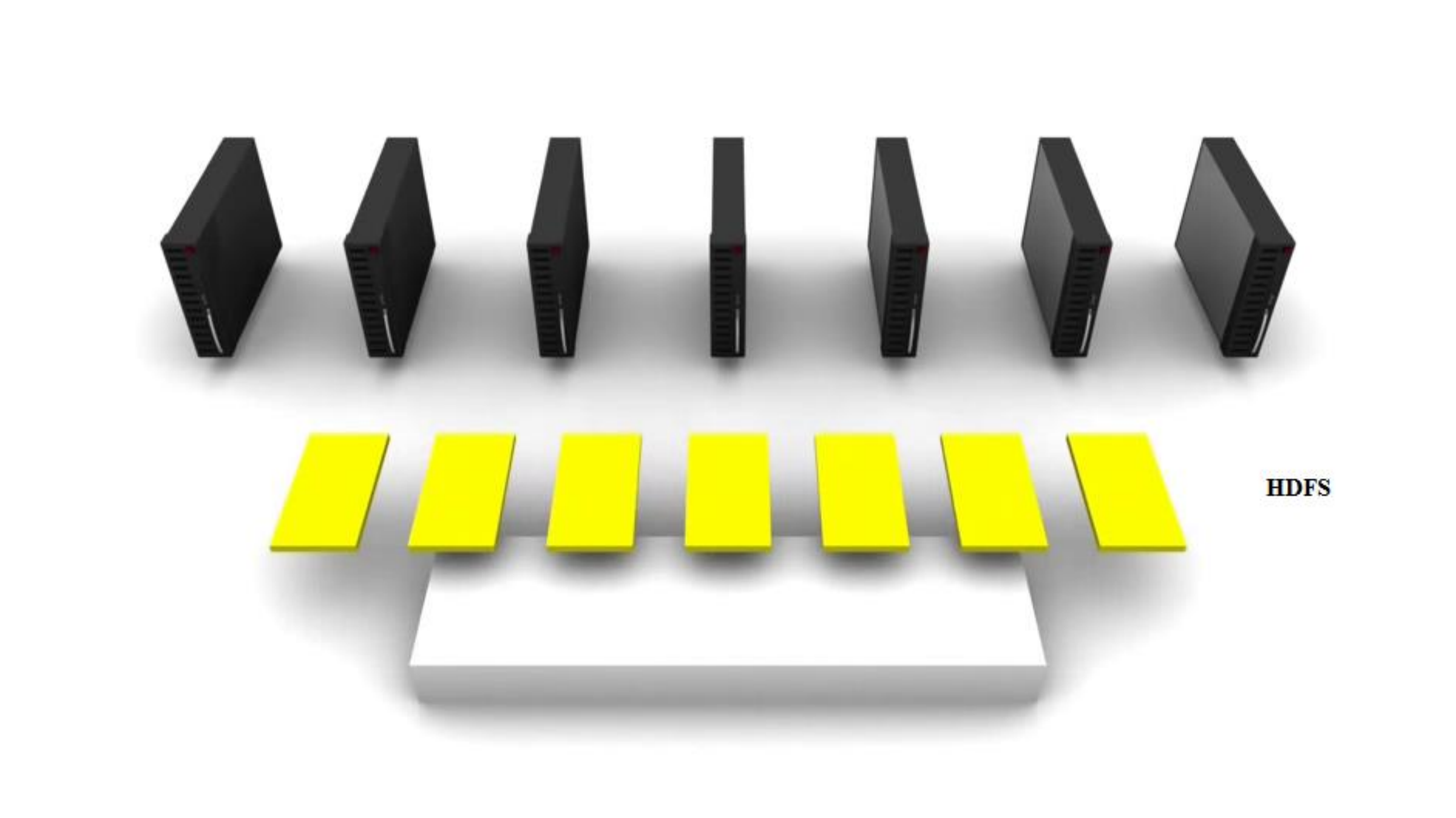
MapReduce

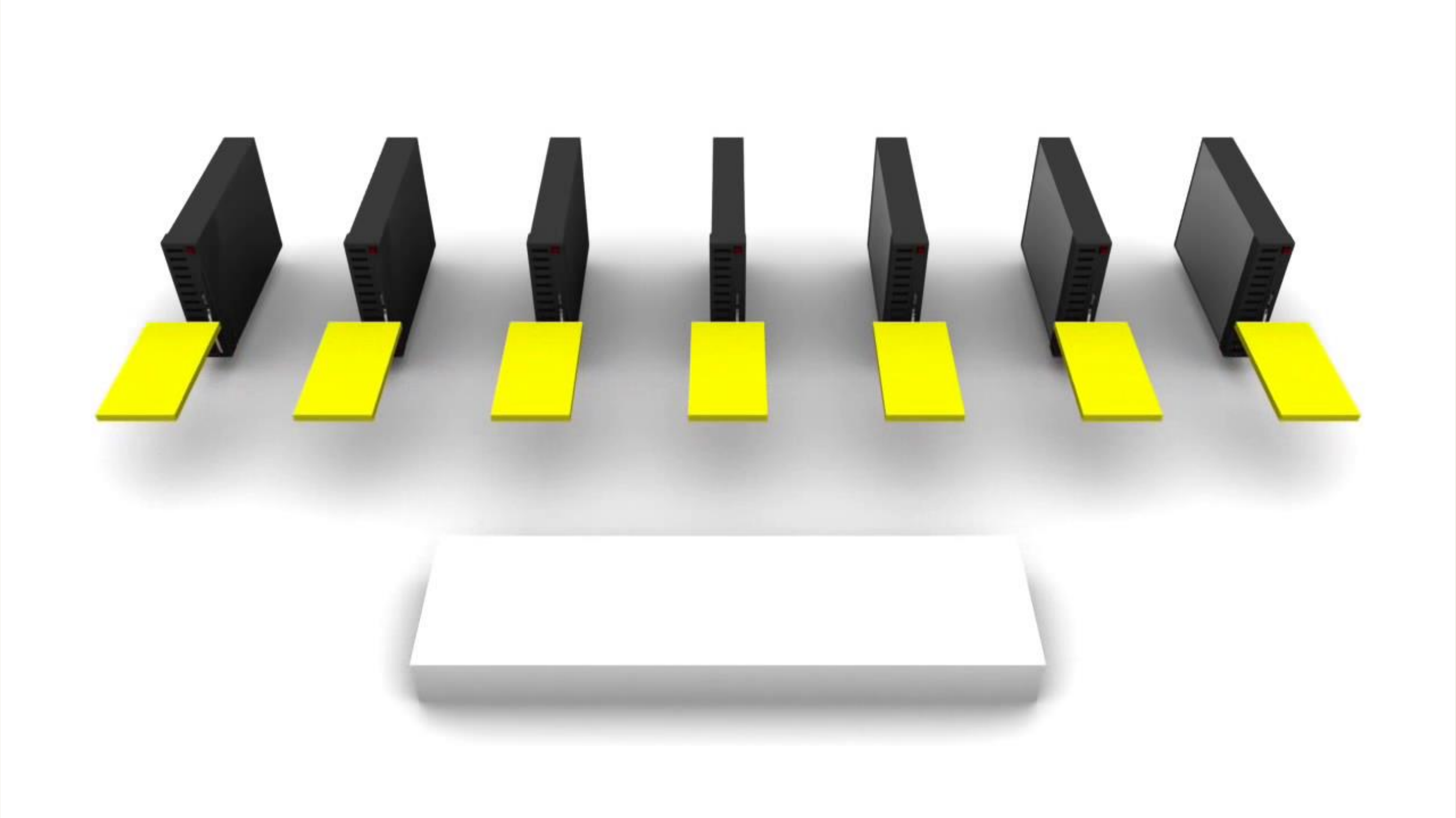
YARN

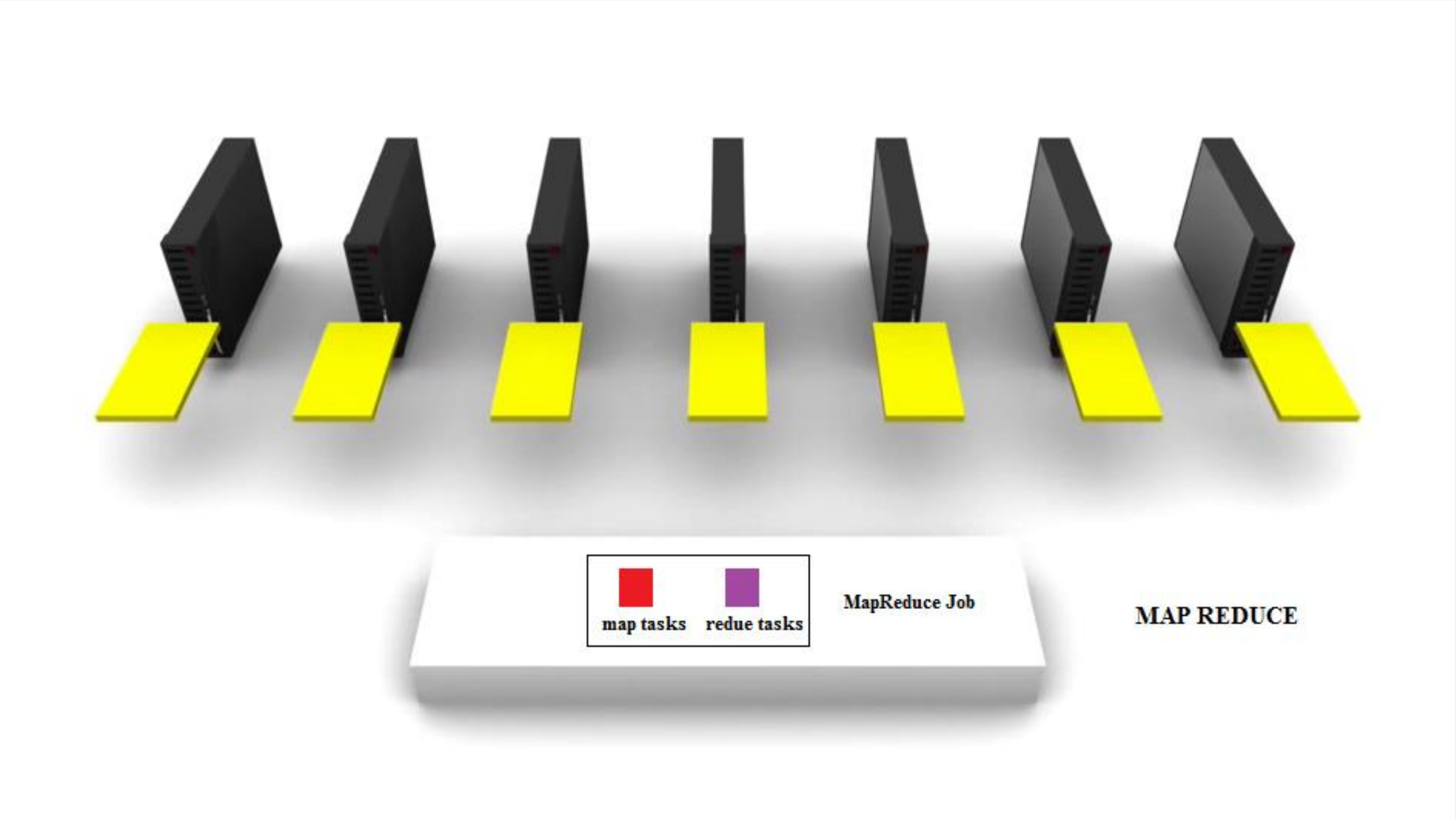
HBase

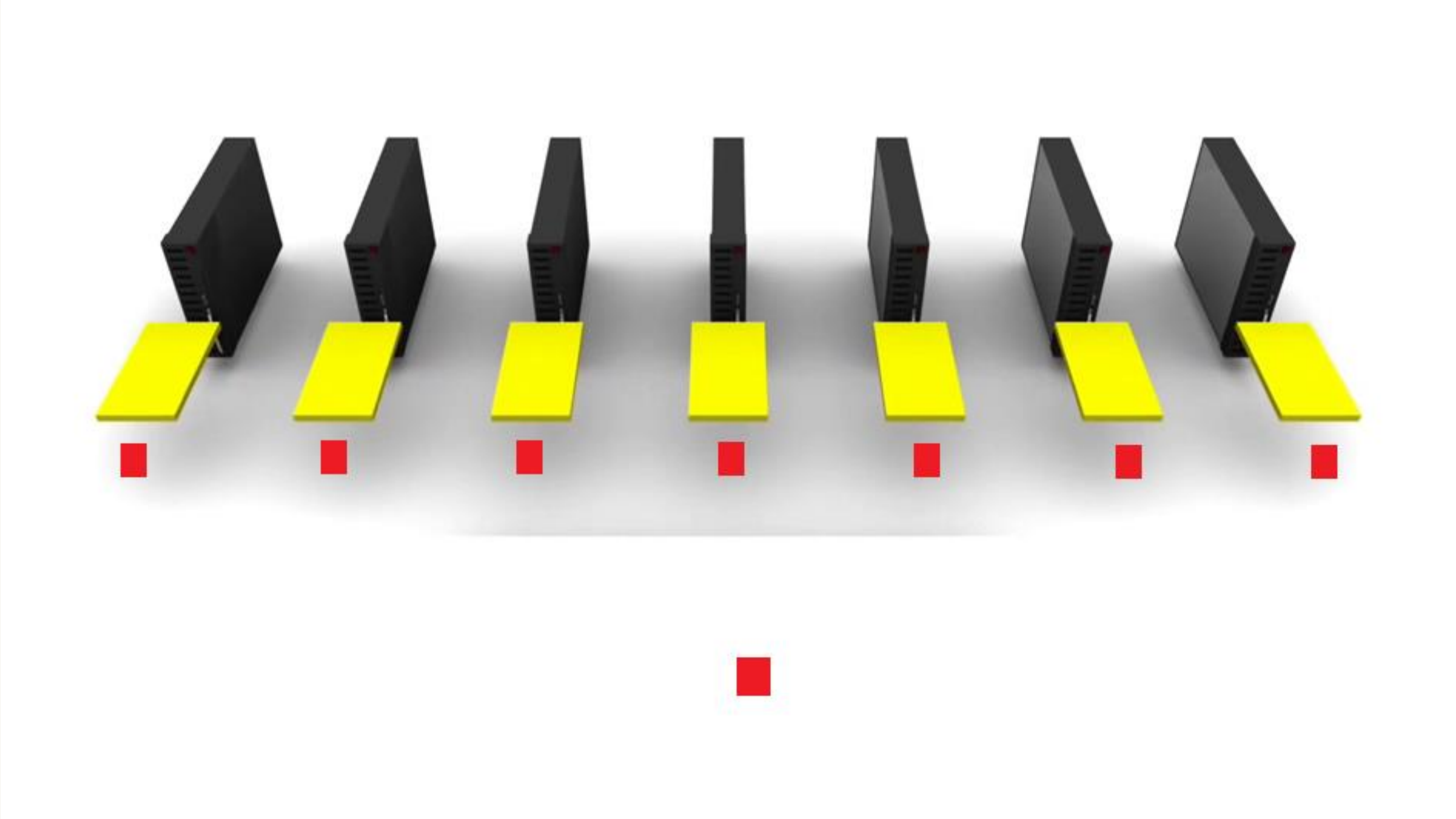
HDFS

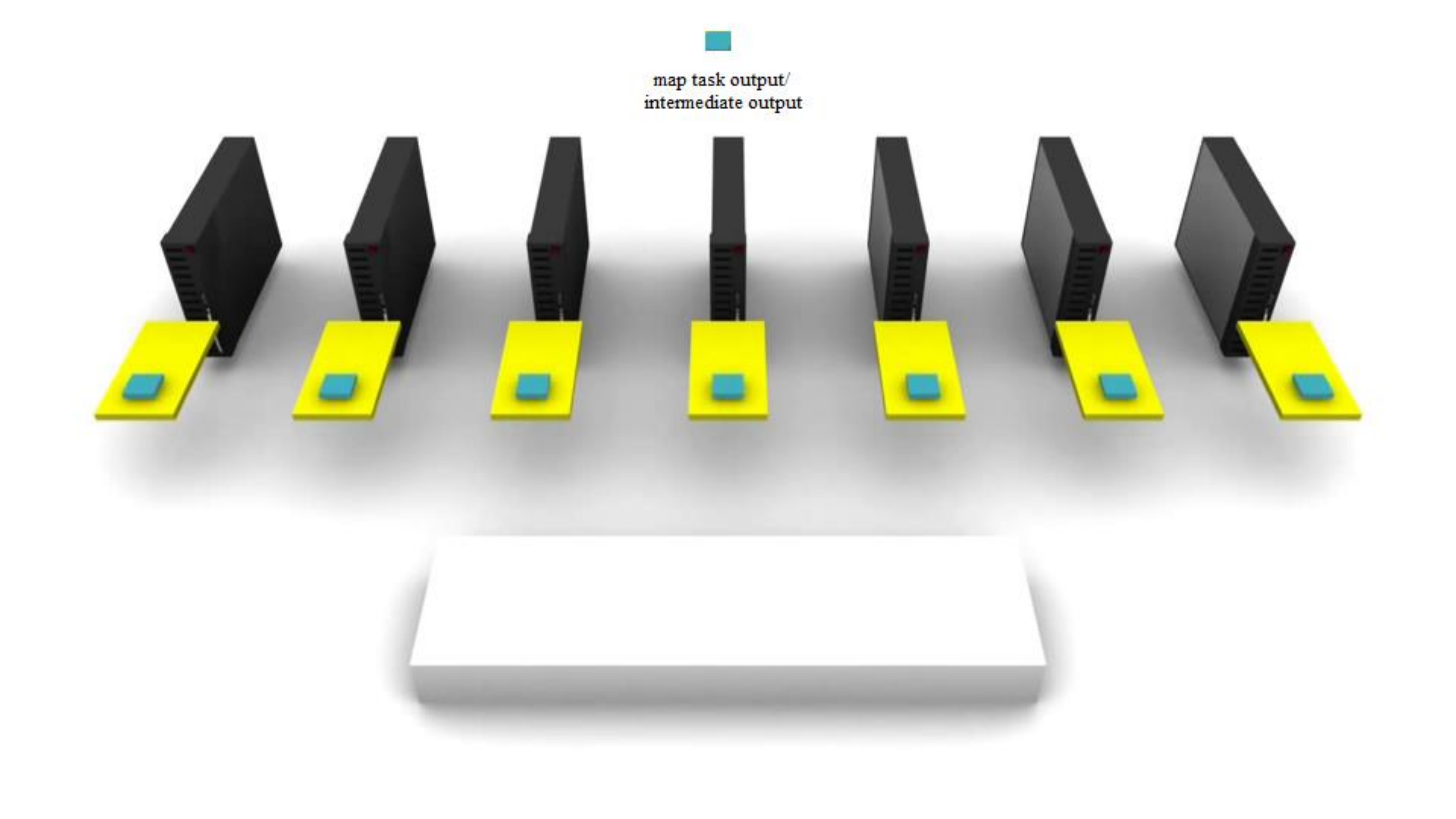

cluster of nodes
Big data

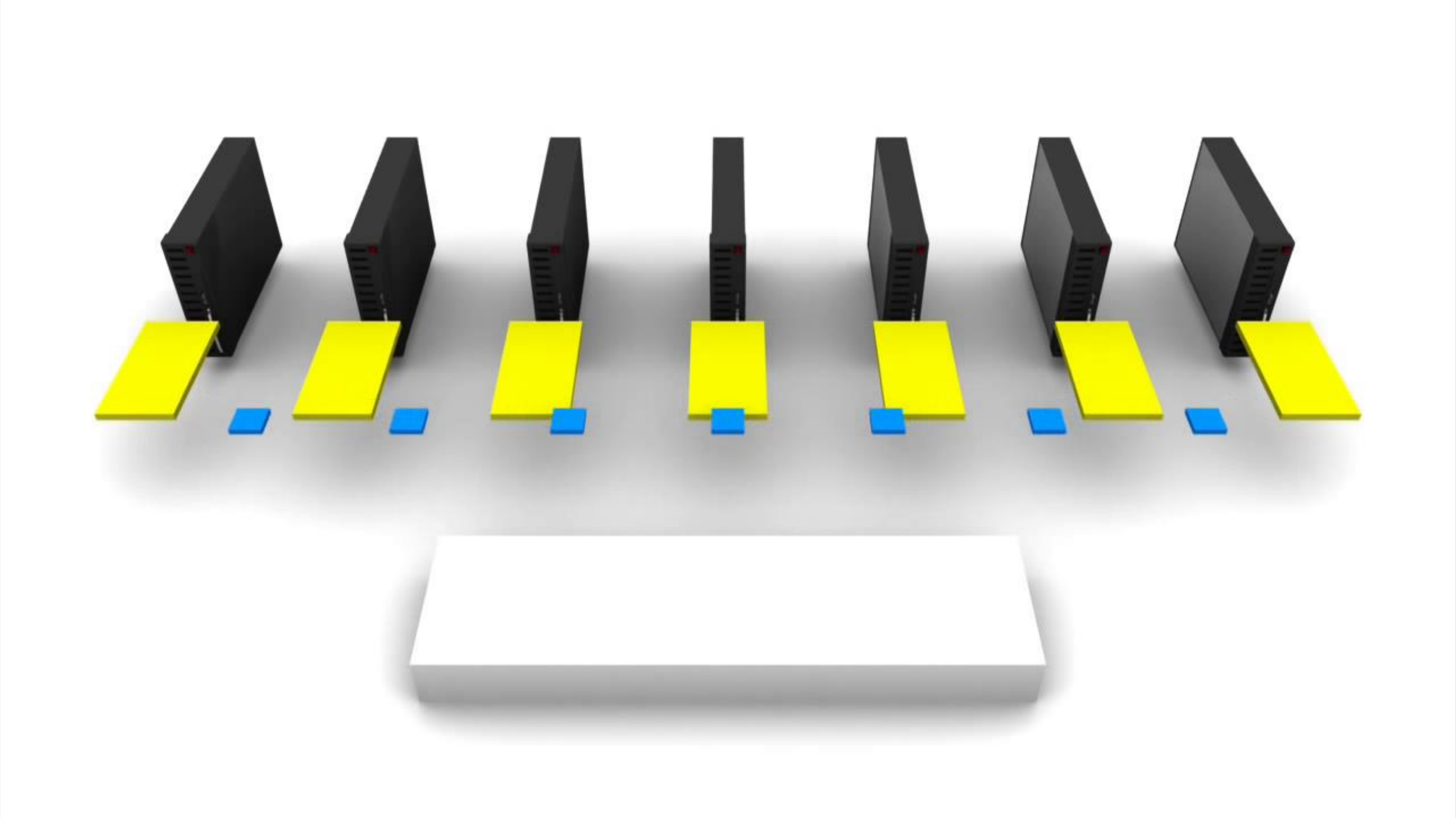

HDFS

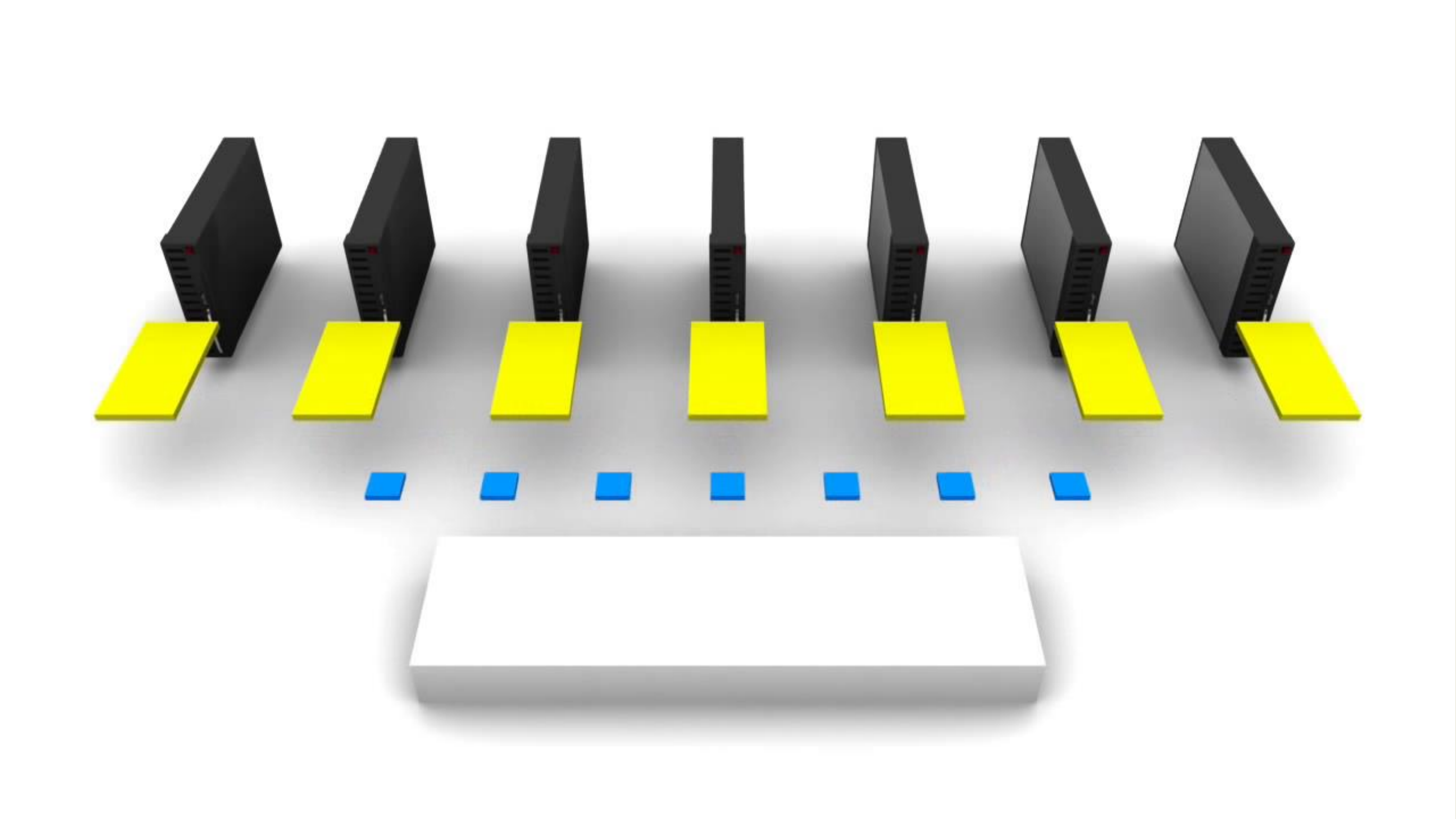


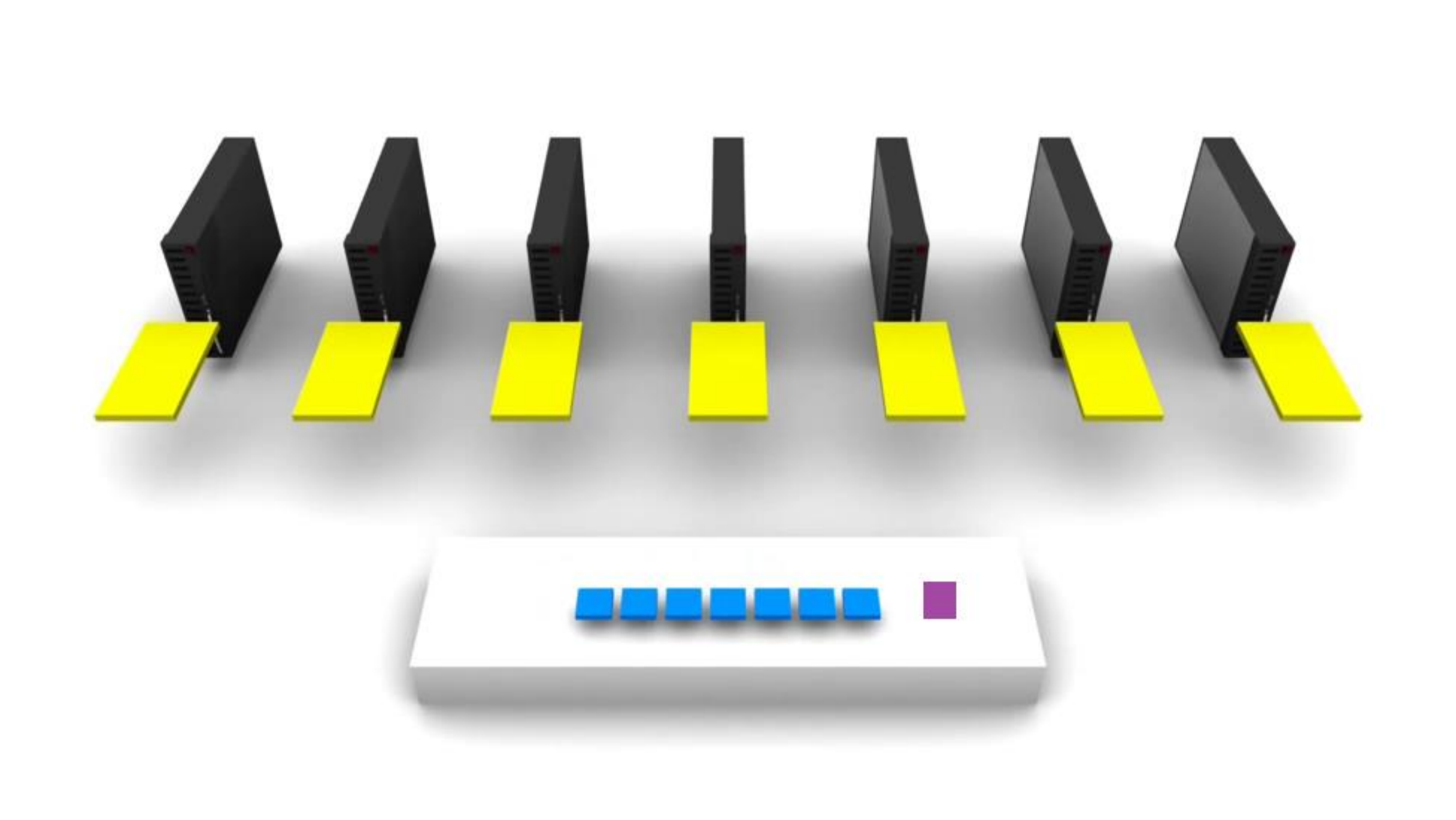

🟥 map tasks
🟪 redue tasks
MapReduce Job
MAP REDUCE

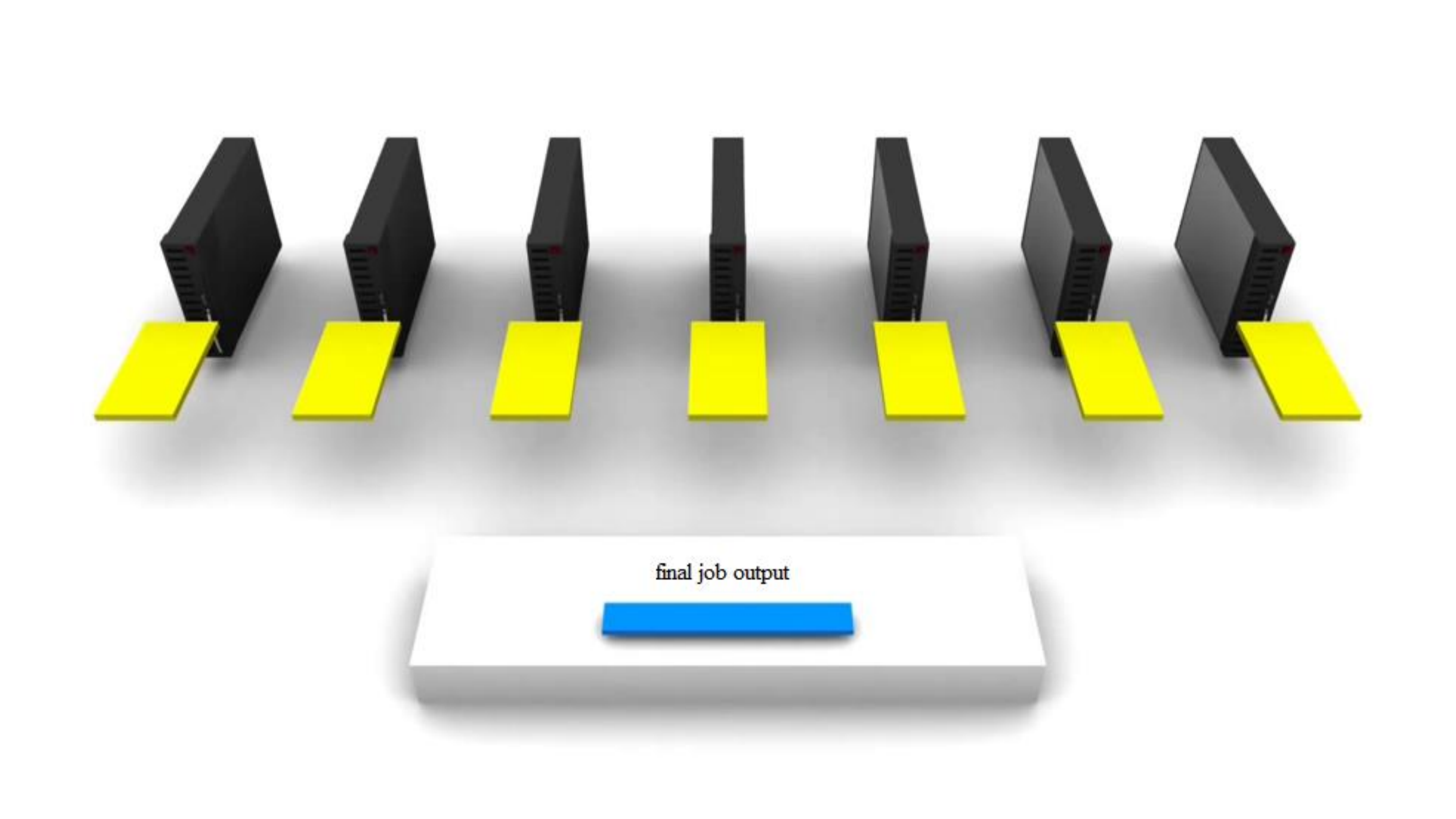



map task output/
intermediate output








final job output

Terminology

- **File System:** is a software that controls how data is stored and retrieved from disks and other storage devices. It manages storage disks with different data access pattern such as file based (NTFS, ext4), object based (swift, S3), block based (cinder, databases)…

- **Distributed System:** group of networked heterogeneous/homogeneous computers that work together as a single unit to accomplish a task. That is, group of computers provide single computer view. Ex: Four systems each with dual core, 4 GB memory, 1 TB storage in a cluster can be said as a distributed system having 8 cores, 16 GB memory, and 4 TB storage.

- Term "distributed" means that more than one computers involved, where data/program can be moved from one computer to another computer.

- **Distributed Computing:** Managing set of processes across cluster of machines/processors that communicate by exchanging messages using MPI and co-ordinate each other to finish a task.

- **Distributed Storage:** Set of storage devices from different computers in a cluster providing single disk view.

- **Distributed File System (DFS):** Every file system has namespace (path) to files in tree structure. Local file system gives namespace only for the disks attached to it. DFS provides unique global (unified) namespace on distributed storage. Ex: if a user executes a command at node5 to display a value of “x” which is in node10, result is displayed in node 5 itself. User doesn’t know where “x” is actually stored.

In short,

- Managing resources across cluster of machines and providing single system view is called distributed system.
- Managing set of processes of an application across cluster of machines is called distributed computing.
- Managing set of storage devices across cluster of machines and providing single disk view is called distributed storage.
- providing global namespace on distributed storage is called distributed file system.

Hadoop Components

Two major components:

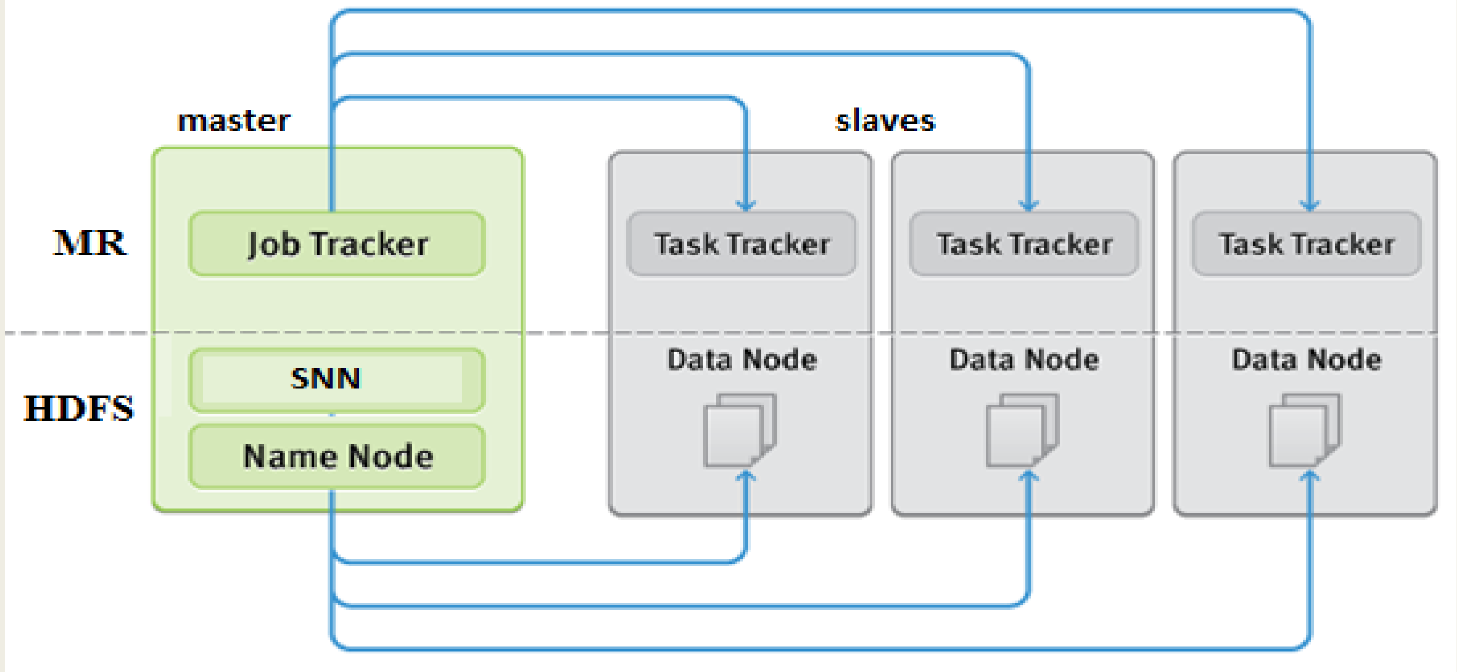
storage part (HDFS): Name Node (NN), Secondary Name Node (SNN), Data Node (DN).

processing part (MapReduce):Tracker (JT), Task Tracker (TT).

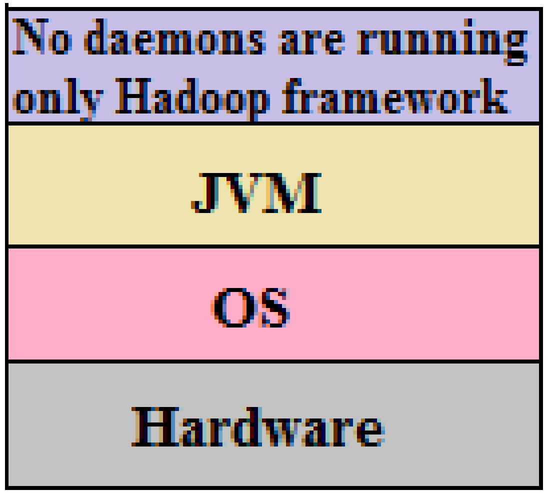
all these components run as a background process (daemons) in servers.

Master daemons: NN, SNN, JT

Slave daemons: TT, DN



Hadoop single node implementation



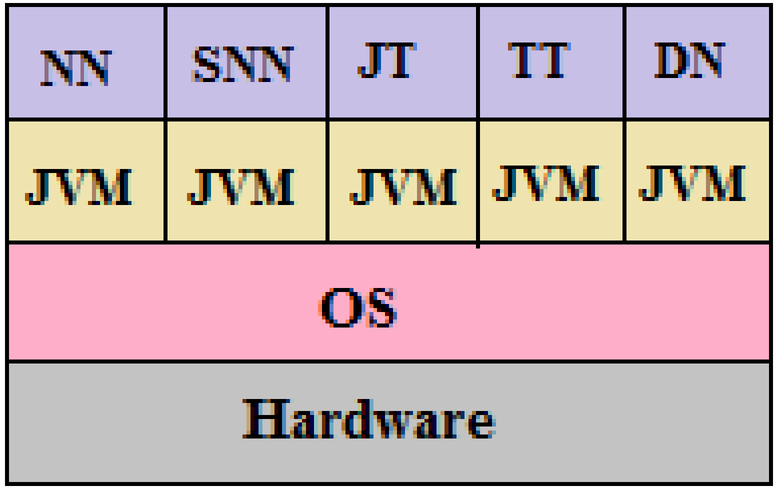
No daemons are running only Hadoop framework

JVM

OS

Hardware

Local or Standalone Mode



NN | SNN | JT | TT | DN

JVM | JVM | JVM | JVM | JVM

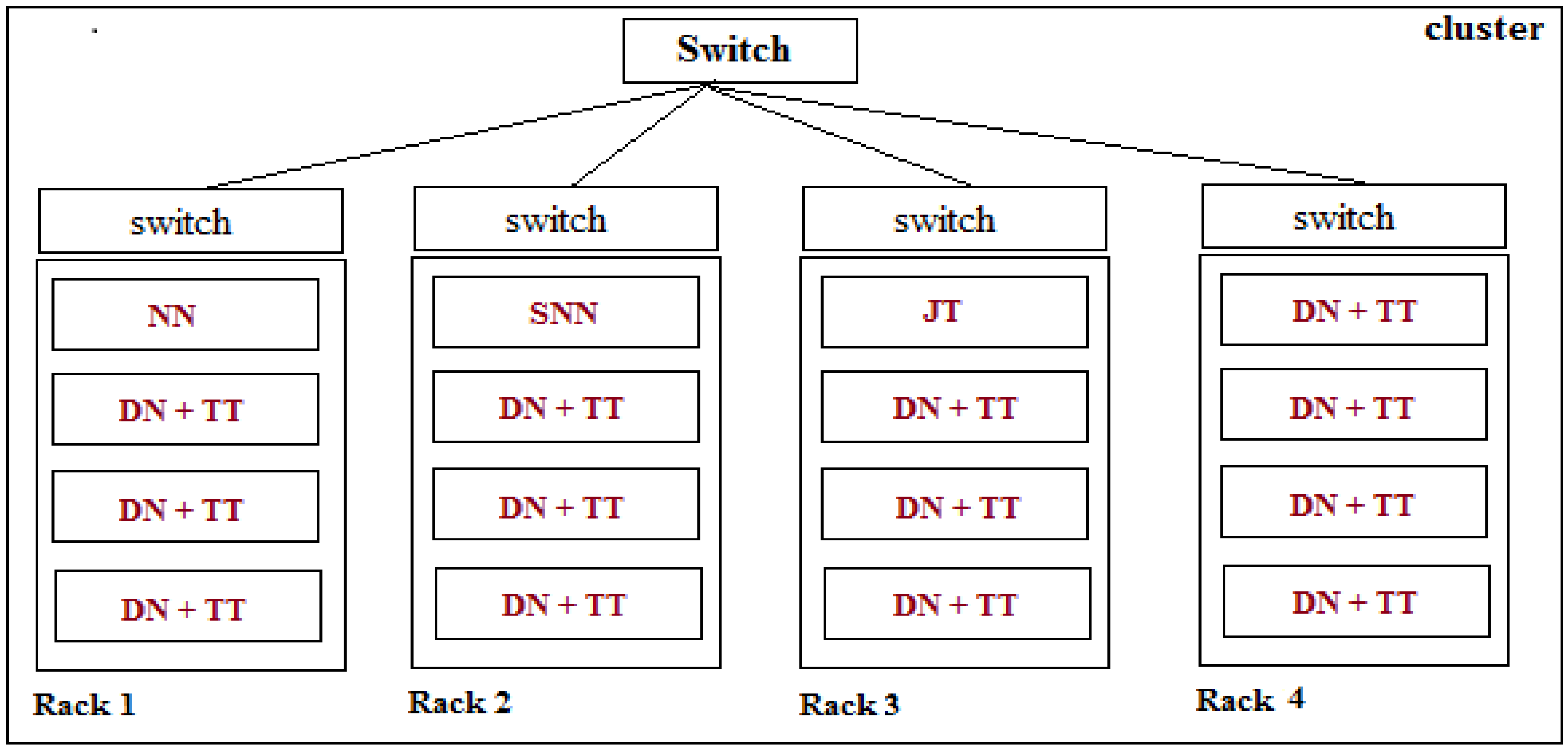
OS

Hardware

Pseudo Distributed Mode

Hadoop Cluster (multi-node)

- data-center contains set of racks stacked with set of servers (nodes) forming a cluster with high speed local network.
- Hadoop works on shared nothing architecture, every node has their own storage lodged (no SAN/NAS/DAS).
- Only one NN, SNN, JT, and multiple slaves can exist in Hadoop cluster. Every slave node runs both DN and TT.
- Master daemons are usually run in dedicated nodes to balance load. Because, NN and JT are heavily hit by slaves.
- Every slave node resources (CPU and memory) are managed by TT for task execution.
- Storage is handled by DN for storing/retrieving data.



Hadoop Distributed File System (HDFS)

when data outgrows the storage capacity of a single machine, partition and store it across cluster of machines with streaming data access pattern.

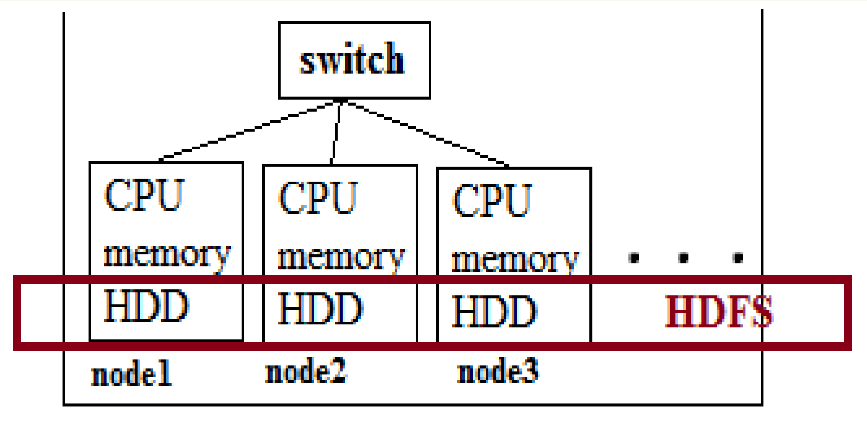
Streaming data access pattern means that a data block is read with no arbitrary/random access.

HDFS is based on both distributed and clustered filesystem.

HDFS has three components: Name Node (**NN**), Secondary Name Node (**SNN**), Data Node (**DN**).

Employs distributed storage to provide single disk view and distributed file system to provide unique global name space on distributed storage.

It is specially designed file system with functionalities such as distribution, fault tolerance, replication…



Use HDFS when you want

- to store large data (over TB).
- to use cheap commodity hardware.
- to process small number of large files than large number of small files.
- batch reads (streaming access) rather than random reads/writes.

Don’t use HDFS when you want

- transaction process, and low latency access (in ms, HBase does this).
- lot of small files (HDFS takes more meta-data, takes more resources).
- parallel write (HDFS supports only append mode).
- arbitrary read (HDFS provides only batch read, from the beginning till the end)

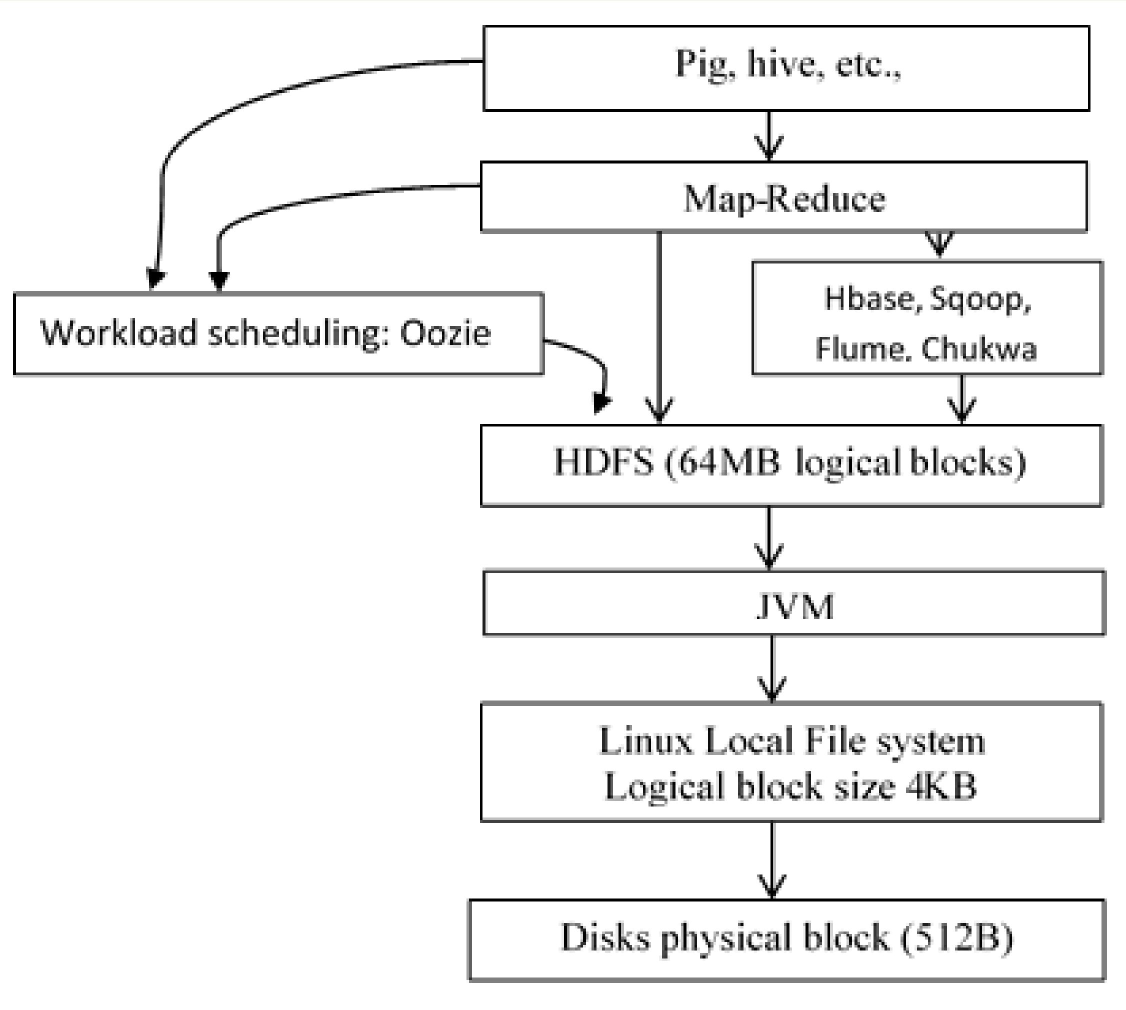
Name Node (NN)

Centralized service that acts as cluster storage manager. Primary responsibilities of NN are:

- Managing file system namespace (path) in tree structure and meta-data for all the files and directories in memory.
- Controls and coordinates DNs for file system operations such as creating files, folders, read/write files… from clients.
- Clients communicate NN in order to perform common file system operations such as read, write…. In turn, NN gives clients the location of DNs to carry operations on data blocks.
- NN suffers from single point of failure (SPOF), which means if NN is down, then the entire cluster becomes inaccessible because of the meta-data unavailability.

Data blocks

- Unit of data storage and access in HDFS is block. It denotes a minimum amount of data that you can store and retrieve.
- Data we want to load onto HDFS is divided into multiple equal sized chunks (blocks), and stored across DNs in the cluster. The DN stores each data block as a file on its local filesystem.


Pig, hive, etc.,
Map-Reduce
Workload scheduling: Oozie
Hbase, Sqoop, Flume. Chukwa
HDFS (64MB logical blocks)
JVM
Linux Local File system
Logical block size 4KB
Disks physical block (512B)

- A file which is bigger than block size is broken up into multiple blocks. Only block is indexed not the contents inside the blocks. By default, block size is 64 MB in HDFSv1.
- you can use block size 128/256 MB if dataset is over PB. A file which is smaller than a block size does not occupy a complete block.
- If our file size is just 10 MB, then only 10 MB is occupied from 64 MB, the remaining 54 MB of a block may be utilized by other blocks unlike Linux file system that occupies entire 4 KB even if file size is 1KB.

- At this case, arbitrary file size (<64 MB) itself is considered to be a block. Every file we upload tends to have minimum 1 block. Directories won't get any blocks. Directories are just meta-data.
- HDFS block size is large compared to disk blocks, and the reason is to minimize the cost of seeks in disks than block transfer rate.
- By making a block large enough, the time to transfer a large block operates at the disk transfer rate.
- Ex: if the seek time is around 10 ms and the transfer rate is 100 MB/s, then to set seek time 1% of transfer rate, block size should be around 100 MB. Therefore, transferring blocks are faster.

Meta-data

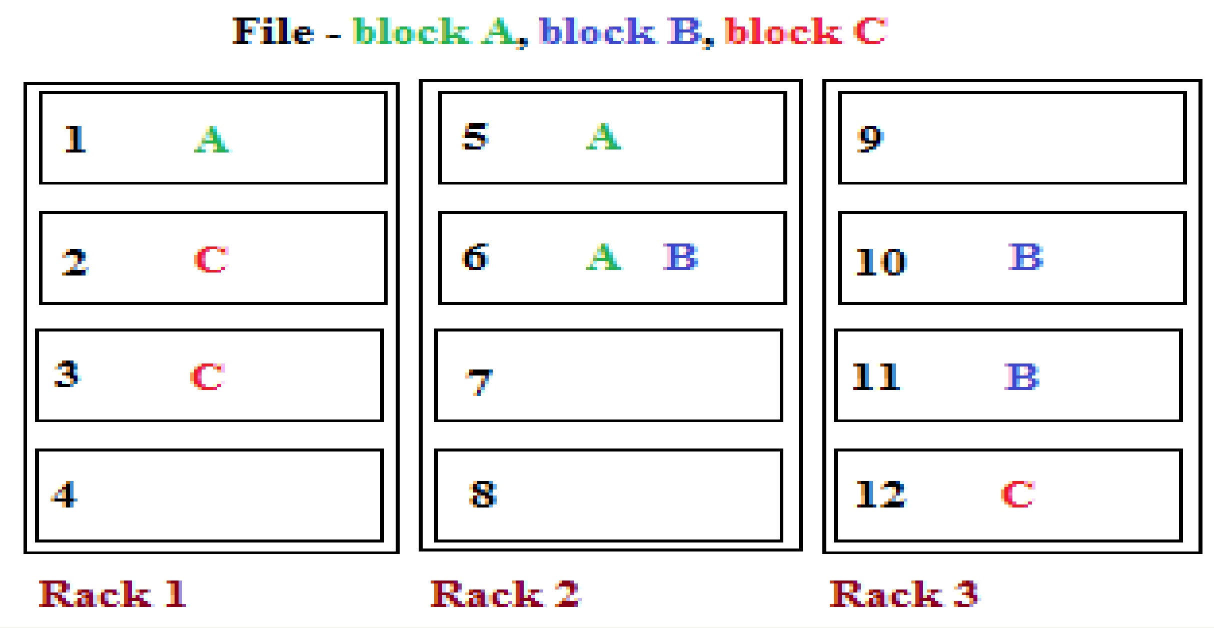
The primary job of NN is to manage file system namepspace in tree structure and meta-data for all the files and directories in memory for faster access.

Namespace is just a path to any object (file/block) in distributed file system. HDFS metadata contains various attributes such as block ID, ownership, permissions, quotas, replication factor, date created of directories, files and blocks…

Meta-data for objects such as file, blocks, and directory take 150 B by default. Attempting to modify meta data will cause HDFS downtime and even permanent data loss.

Replication placement using rack awareness

- As Hadoop deployment is mostly done on unreliable commodity machines, the failure of nodes is more likely. Therefore, computation, and data failure are inevitable. How does Hadoop take care of block loss? HDFS doesn’t rely on RAID (RAID only solves HDD failure and mirroring is expensive for big data).
- Consider an input file having three blocks: A, B, C. if you store all these blocks in a single node, then if that node crashed or disk failed, then it is not possible to get files back. Therefore, HDFS itself ensures fault tolerance by replicating blocks.



File - block A, block B, block C

Rack 1	Rack 2	Rack 3
1 A	5 A	9
2 C	6 A B	10 B
3 C	7	11 B
4	8	12 C

- Number of replications of data blocks is denoted as replication factor (**RF**), by default it is 3. Replication helps to achieve fault tolerance, network performance and improves data parallel processing. Please note that only blocks of a file are replicated not directories.
- We can run same task of a job on block A, B, C at a time, so that data parallelism is achieved. But, same task will not be executed on two copies of block A simultaneously. On the other hand, tasks of two different jobs can be run two copies of block A.
- If you have three copies in the same machine, if machine is down, then none of the copies will be accessible. Therefore, there is no point in making more than one copies in a same machine. It wastes memory and provides no fault tolerance.

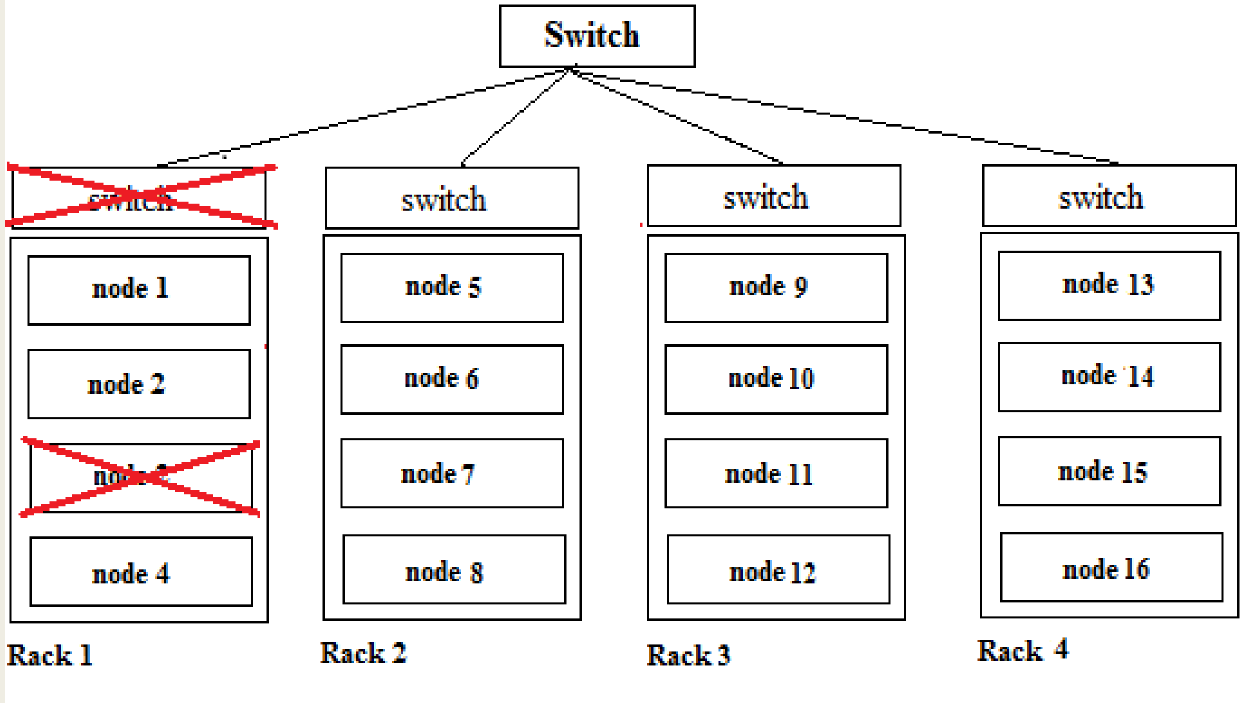
- So, each copy is distributed to more than one nodes in the cluster. Rule of thumb is, RF must be less than or equal to number of nodes. NN decides where to place replicated blocks based on cluster topology. It is also called as rack awareness. Each block is replicated with 3 copies.
- First copy of block A is stored in node5 rack2.
- Second copy of block A is stored in same rack, but different node (node6).
- Third copy is stored in node stacked in some other rack (rack1).

- It is called rack/topology aware data block placement.
- why such a placement scheme is required? Consider the following cases as shown in Figure:

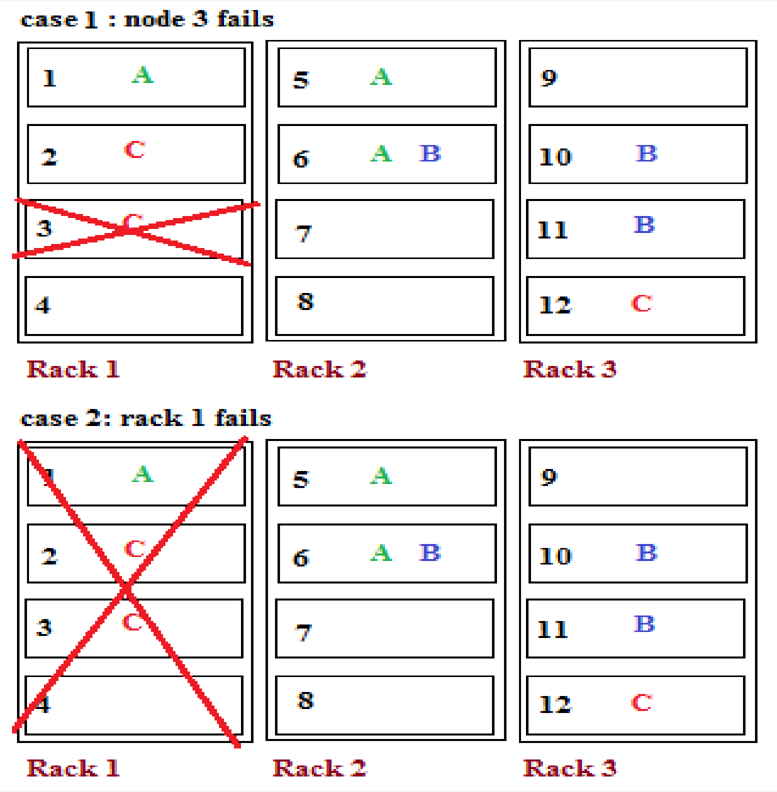
Case 1: What if a node is failed or disk failed or data block corrupted (node3)?

Case 2: What if the entire rack (rack1) is down due to network switch failure?

If any one of the cases happened, then how to reconstruct original file?



- Case 1: HDFS replication solves node failure, disk failure, block corruption. If node3 in rack1 is crashed or disk failed or block corrupted, then the same block C is available in node2 of rack1 to reconstruct original file with block A and B.
- Case 2: HDFS replication solves network failure (rack failure) also. If you have all 3 copies in different machines of same rack, none will be accessible if top of the rack switch failed.
- Therefore, by default, first two copies are stored in two different machines of same rack and third copy is stored in node of some other rack. Thus, rack awareness helps to avoid loss of entire rack.



Secondary Name Node (SNN)

NN suffers from Single Point Of Failure (SPOF). If NN is lost, then meta-data namespace and meta-data are lost.

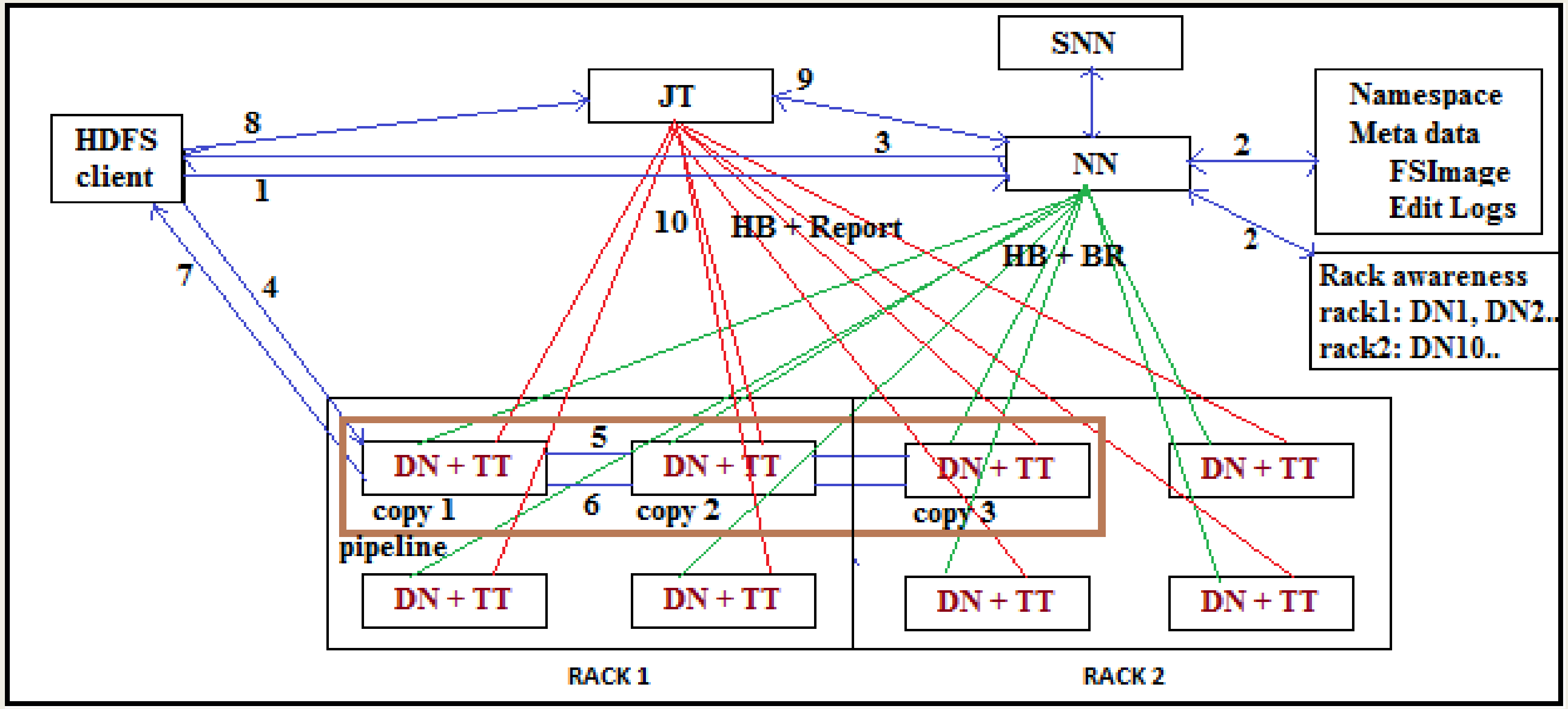
So, entire cluster becomes inaccessible. Therefore, it is recommended to run NN in enterprise grade server in production environment.

However, failure of a server is inevitable. Therefore, copy of namespace and meta-data are lost are maintained in another machine called SNN.

Meta-data can be even maintained in shared file system called Network File System (NFS) to recover NN from failure.

Data Node (DN)

- DN daemon is responsible for managing local storage disks. It handles file system operations such as creating, reading, opening, closing, deleting blocks…
- The DN has no knowledge about what data is inside the blocks. It stores each block in a separate file in its local file system.
- Blocks are loaded/deleted in DNs based on the instructions of NN, which validates and processes requests from clients. NN doesn’t perform any read/write operations for client.



Who splits files into blocks?

Users submit file upload/download requests to a daemon in HDFS, called “HDFS client”. HDFS client is responsible to divide input file into blocks.

Block report (BR)

Block Report is a message sent by DN to NN about all facets of blocks in that corresponding DN.

A Block report contains a list of all blocks in a DN and its meta-data. NN updates meta-data of block upon receiving BR. While DN’s start up, it scans through all the blocks available at present and create a BR to send to NN.

Heartbeat (HB)

HB is very similar to ping command. It is required for scalable architecture. NN periodically (by default 3 seconds) receives HB from each DN in the cluster.

It is a periodic health check to ensure whether a slave node is alive or not.

Map-Reduce (MR)

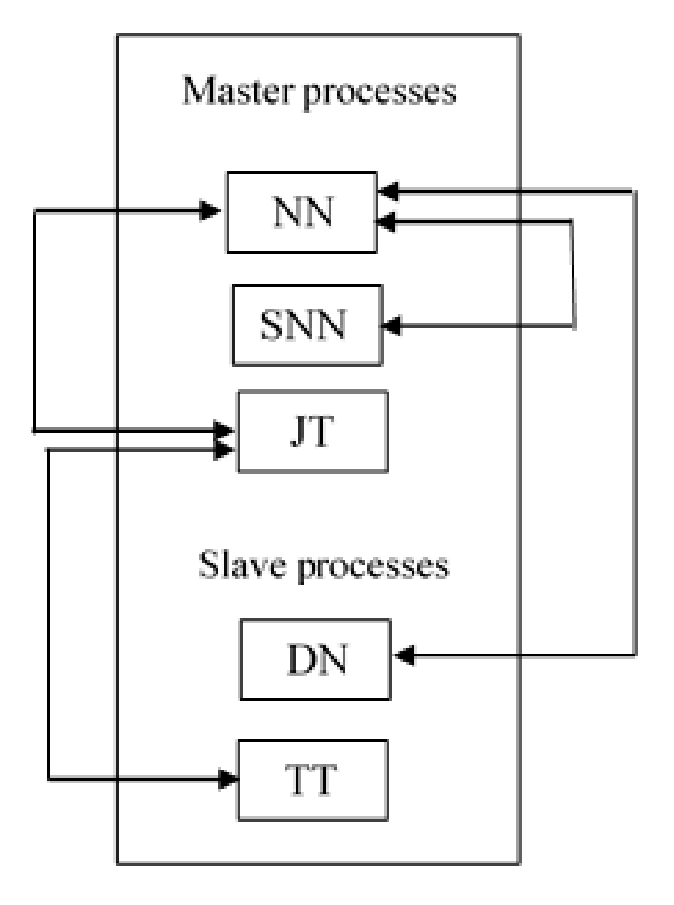
- MR is highly distributed, scalable, fault tolerant data parallel batch processing tool of Hadoop ecosystem that runs on cluster of commodity, unreliable nodes to process big data.
- Data is collected, and stored onto HDFS before launching jobs. It is used to develop scalable algorithms. Program designed for single node can be used for 1000's machines without remodifying the code.
- MR is termed as embarrassingly (parallel + concurrent) parallel framework.
- MR programming is based on LISP (functional programming). It processes list of input values into list of output values.
- Both map and reduce are higher order functions in LISP. A function taking another function as argument is called higher order function.
- It treats the computation as the evaluation of mathematical functions. Functional programming doesn't maintain state and doesn't support blocking/synchronization.
- Hence, it is scalable. MR tasks run independently, so it is easy to handle partial failure. Ex: LISP, Haskell, scala, clojure, smalltalk, ruby…

- Every application is not suitable to implement in MR. Only applications such as counting, ranking, aggregation, searching, sorting… can be accomplished with MR.
- MR doesn't have any predefined queries unlike OLAP for DWH. It is all about writing adhoc algorithms to process entire dataset.
- MR performance is lower than general programming languages to process small dataset like few 100 MBs, because MR has a sequence of steps to carry out.
- However, to experience the true power of MR, one needs to have data in TBs, because this is where RDBMS takes hours and fails, whereas Hadoop does the same in couple of minutes.
- As you know MR has two core-components: JT and TT. MR has two tasks: map and reduce

MR applications are

- Simple statistics: count, ranking…
- Complex statistics: PCA, covariance matrices…
- Classification: naïve bayes, random forrest, perceptron, regression…
- Clustering: k means, hierarchical, density, bi-clustering…
- It can pre-process huge data for applying machine learning algorithms.
- Sorting, searching, text processing, index building, graph creation and analysis, pattern recognition, collaborative filtering, sentiment analysis…

Communication between master-slave daemons



Master daemons talk to each other. Slave daemons don’t talk to each other. JT communicates with all TT, similarly NN talks with all DNs and SNN.

Job Tracker (JT)

Primary responsibilities of JT are

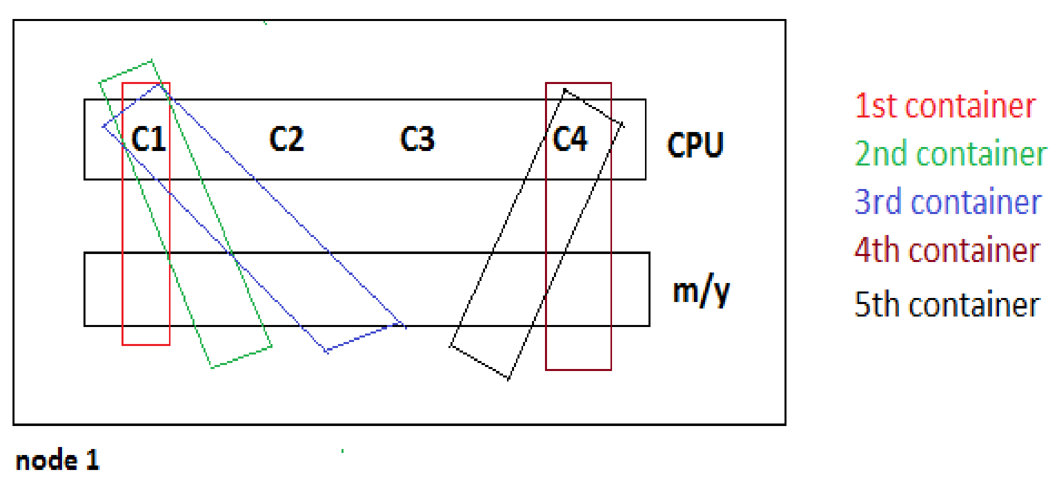
- managing and monitoring resources (CPU and memory) of TTs.
- prepares execution plan and phase co-ordination.
- scheduling map/reduce tasks to TT.
- jobs life cycle management (from the time of submission till its completion).
- maintaining job history (job level statistics).
- providing fault tolerance.

JT is also rack aware while scheduling tasks. Most of the time, computing takes place at nodes where data block is physically available to reduce network traffic. It is called data locality/gravity.

Since MR reads/writes data from standard input/output, many languages (Java, C++, Ruby, Python…) can be used to write MR programs. JT also suffers from SPOF.

Task Tracker (TT)

- TT is a daemon that runs in every slave nodes. It manages the local resources (CPU and memory) as a slot.
- A slot is a fixed logical pack of portion of memory and CPU to run map/reduce task as shown in Figure.
- TT is responsible for starting and managing individual map/reduce tasks in slots.



- CPU in a node may contain more than one cores (C_1, C_2…). A slot is formed as, for example, 1 virtual core + 1 GB memory. There can be many slots in a TT and number of slots in a TT is determined by the available CPU and memory resource.
- Number of map and reduce task slots in a node is fixed. Ex: if there are 4 slots in a node, we can assign two slots for map and two slots for reduce tasks.

- JT monitors the availability of slots to assign tasks in TT. TT executes map/reduce tasks of a job on block of data taken from HDFS.
- TT periodically reports to JT about resource availability, task progress, and its status. It sends HB to JT to ensure liveness of TT.

- As you add more nodes in Hadoop cluster, the number of task slots increases linearly and data blocks can be processed highly in parallel.
- Therefore, number of blocks processed per unit of time (throughput) is increasing linearly.

Map and Reduce tasks

- When you code for MR, map and reduce are called as functions. When they are executed, the same map and reduce are called as tasks: map/mapper task, and reduce/reducer tasks.
- There are two phases in MR: Map phase and Reduce phase.
- MR is highly asynchronous. There should not be any communication/blocking/synchronization among map tasks and among reduce tasks across nodes.
- However, there is a synchronization point between map and reduce tasks. Unless all mappers are completed, reduce task can't be executed. So, Map tasks have higher priority than reduce tasks as it has to complete before reducer can start.
- Map tasks are scheduled to achieve data locality to the extent. There can be non-local execution too. Reduce tasks don't have the advantage of data locality.
- JT dynamically decides where to run reduce tasks based on slot availability and load of the TT. However, JT tries to reduce network consumption while assigning reduce tasks.
- One can implement business logic such as filtering, projection, transformations in map phase and aggregations, joins, sorting… in reduce phase in MR life cycle.

MR related Terminology review

NN: manages file system namespace and meta-data, controls and co-ordinates DNs.

DN: manages local disk, stores and retrieves data blocks upon NN instructions.

JT: prepares execution plan, launches map/reduce tasks, and performs phase coordination.

TT: manages CPU and memory of slaves, runs map/reduce tasks, tracks task status and reports to JT.

Job client: You submit MR job to job client (a daemon), which interacts with JT for you. It can be run in any of the nodes in Hadoop cluster.

Job: a unit of work that the client wants to perform. It comprises the input data, MR program, and configuration information.

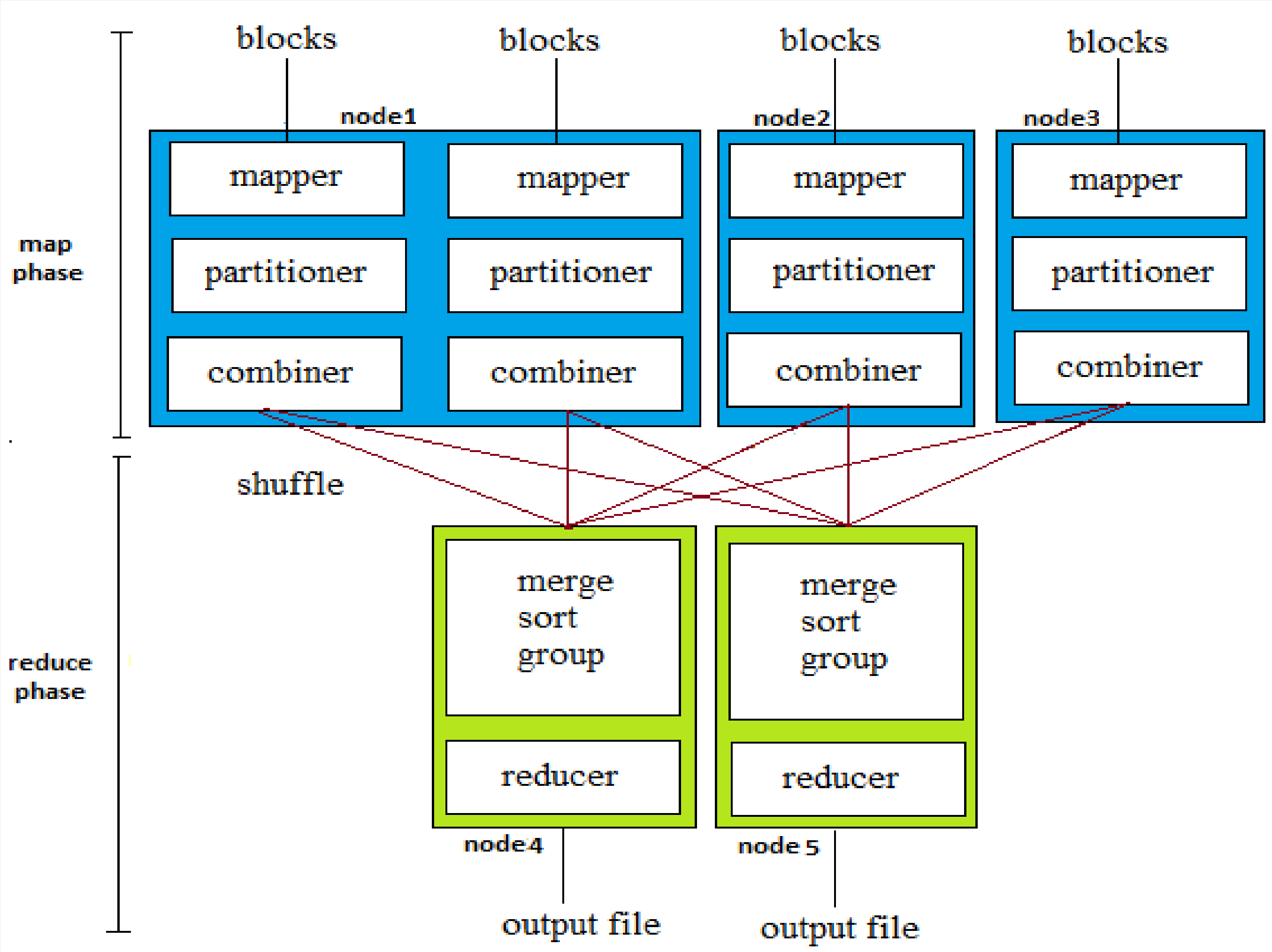
Task: a piece of code running within a single thread of execution. It can be map/reduce task.

Map task: executes a user defined function that reads data from HDFS as key/value pairs and produces arbitrary number of intermediate key/value pairs.

Reduce task: executes a user defined function that gets list of values that belong to a key as input and produces output key-value pairs.

Driver program: program that initiates the execution of map and reduce tasks.

MR Phases


blocks
blocks
blocks
blocks
node1
node2
node3
map
phase
mapper
mapper
mapper
mapper
partitioner
partitioner
partitioner
partitioner
combiner
combiner
combiner
combiner
shuffle
reduce
phase
merge
sort
group
merge
sort
group
reducer
reducer
node4
node 5
output file
output file

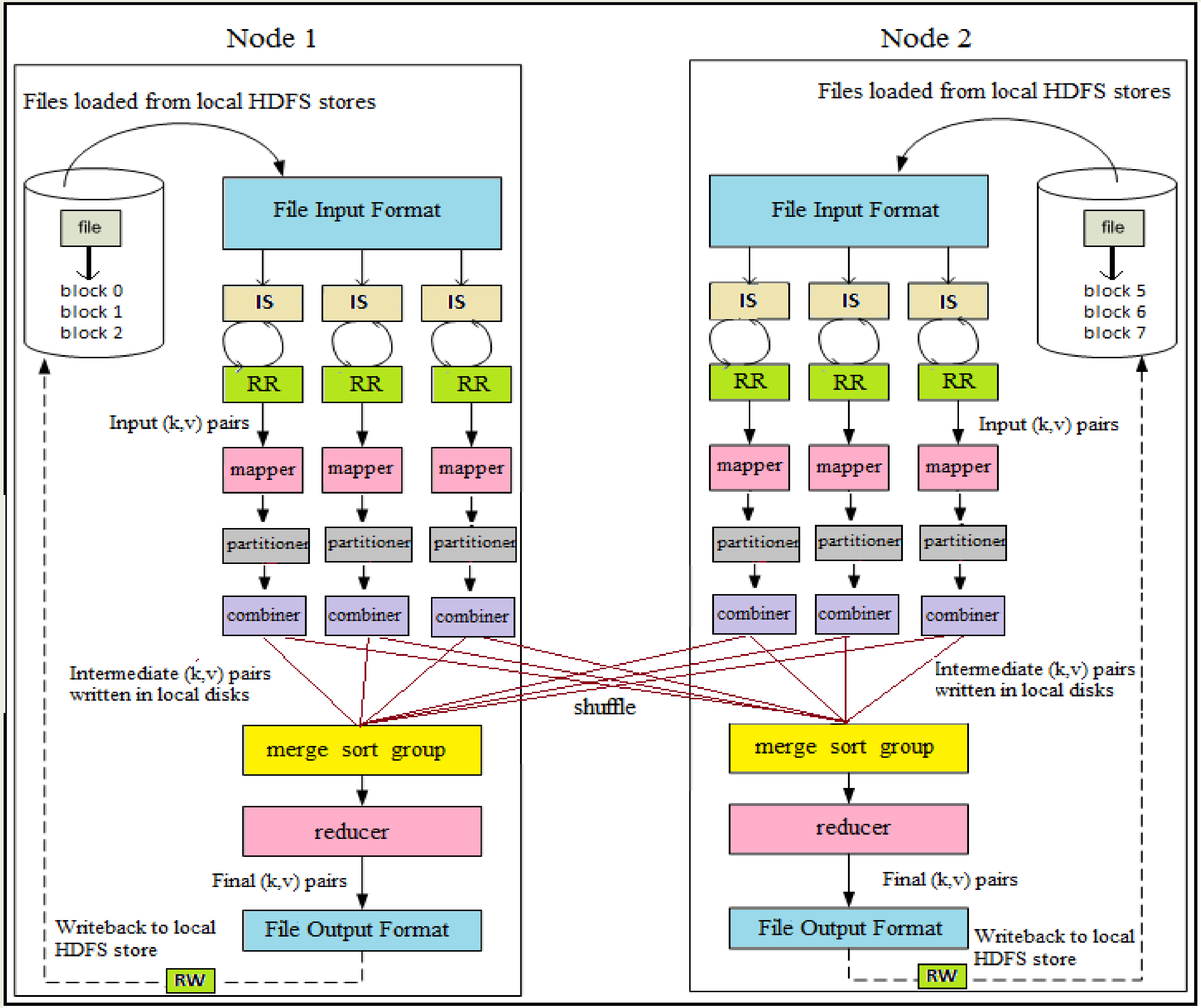
Each phase contains sequence of steps to be carried out for data processing. Every step in the MR phase is based on key-value pair.

Map phase

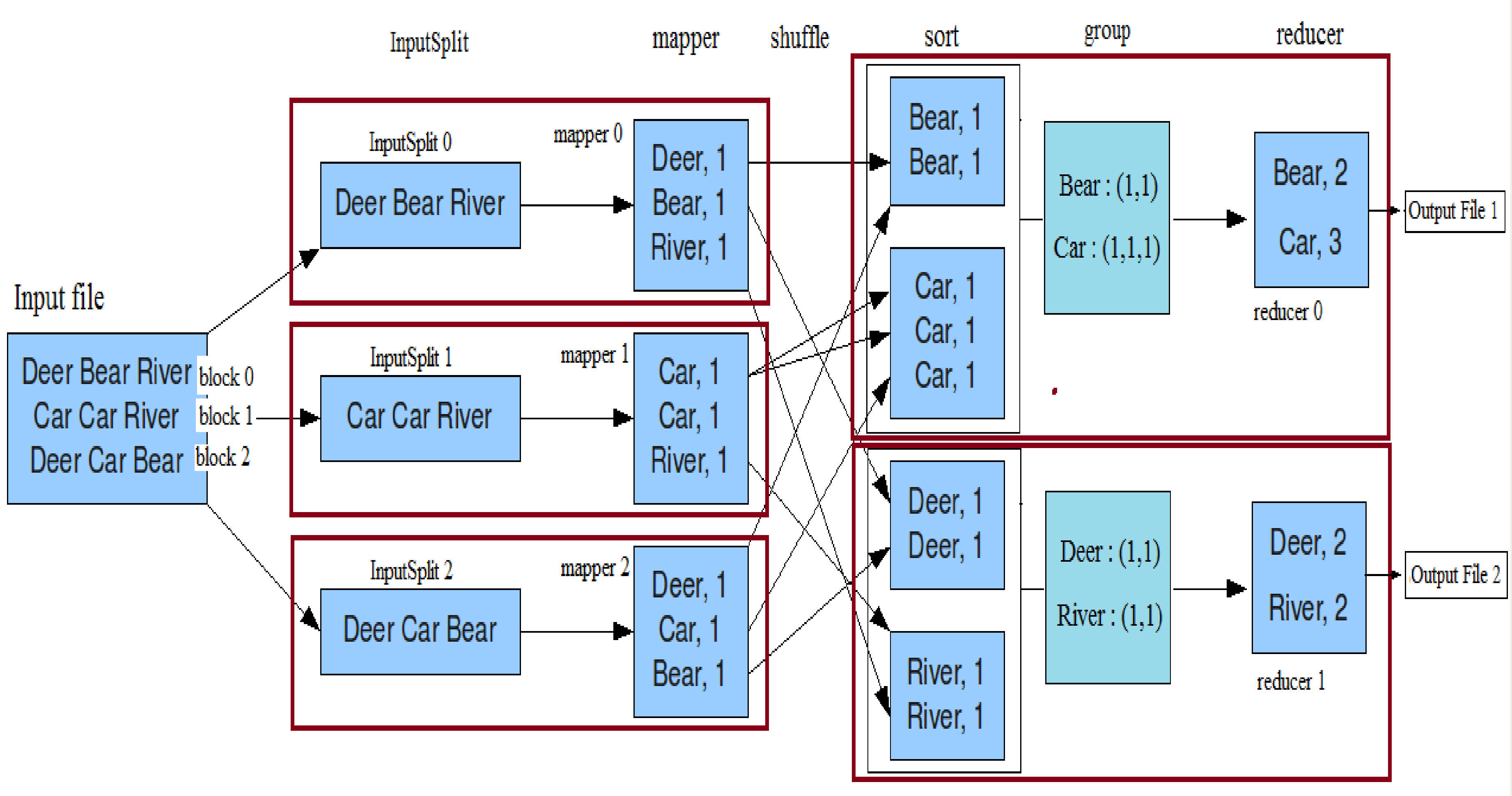
Input file→blocks→file input format→input split→record reader→mapper→partitioner→ combiner

Reduce phase

shuffle→merge→sort→group→reducer→file output format→record writer→output file


Node 1
Node 2
Files loaded from local HDFS stores
file
block 0
block 1
block 2
File Input Format
IS
RR
Input (k,v) pairs
mapper
partitioner
combiner
Intermediate (k,v) pairs written in local disks
shuffle
merge sort group
reducer
Final (k,v) pairs
File Output Format
Writeback to local HDFS store
RW
Files loaded from local HDFS stores
File Input Format
file
block 5
block 6
block 7
IS
RR
Input (k,v) pairs
mapper
partitioner
combiner
Intermediate (k,v) pairs written in local disks
merge sort group
reducer
Final (k,v) pairs
File Output Format
Writeback to local HDFS store
RW

wordcount


InputSplit
mapper
shuffle
sort
group
reducer
Input file
Deer Bear River
Car Car River
Deer Car Bear
block 0
block 1
block 2
InputSplit 0
Deer Bear River
mapper 0
Deer, 1
Bear, 1
River, 1
InputSplit 1
Car Car River
mapper 1
Car, 1
Car, 1
River, 1
InputSplit 2
Deer Car Bear
mapper 2
Deer, 1
Car, 1
Bear, 1
Bear, 1
Bear, 1
Car, 1
Car, 1
Car, 1
Bear : (1,1)
Car : (1,1,1)
Bear, 2
Car, 3
reducer 0
Output File 1
Deer, 1
Deer, 1
River, 1
River, 1
Deer : (1,1)
River : (1,1)
Deer, 2
River, 2
reducer 1
Output File 2

Customer purchase calculation

We have 2 GB of sales information system. We are going to calculate the total money spent by each customer. The entries of sales input file are like below:

Name	price	item	date	
Ram	7000	mobile	12-2-2014	….
Ravi	1000	headset	10-2-2015	….
Raja	10000	laptop	2-1-2015	…..
Ram	2000	charger	3-2-2014	….
Rahim	1000	books	5-3-2011	….
Ravi	500	pizza	23-11-2015	….
Ram	100	pen	12-12-2015	….

…….

For our simplicity to understand, we are going to use only two blocks to work out.

Map phase

Step 1: file → blocks → FileInputFormat → IS → RR

Step 2: RR (k1, v1) → mapper

Step 3: mapper (k1, v1) → (k2, v2) → partitioner

Step 4: partitioner (k2) → gives reduce node number → spilling process → combiner

Step 5: combiner (k2, list(v2)) → (k3, v3) → shuffle

Reduce phase

Step 6: shuffle → merge → sort

Step 7: sort (k3) → (k3, v3) → group

Step 8: group (k3)→ (k3, list (v3)) → reducer

Step 9: reducer (k3, list (v3)) → (k4, v4) → OutputFileFormat → RW

Step 10: RW (k4, v4) → output file

Note: moving from one step to another step, if any changes in (key:value) pairs then (k#, v#) is incremented by 1. Ex: key and value after mapper are incremented from (k1, v1) to (k2, v2).

Map phase

Step 1: file → blocks → FileInputFormat → IS → RR

- User submits 2 GB input file to HDFS client. HDFS client divides file into 32 blocks (2GB/64MB). Then HDFS client communicates NN to upload blocks onto HDFS. NN assigns blockID and gives list of three DNs for each block. HDFS client moves blocks onto HDFS.
- For our simplicity, I don’t replicate blocks in this example and consider only two blocks (B1, B2) for further discussion.

B1 in node1

Ram	7000	mobile	12-2-2014	….
Ravi	1000	headset	10-2-2015	….
Raja	10000	laptop	2-1-2015	…..
Ram	100	fine	12-12-2015	….

B2 in node2

Ram	2000	charger	3-2-2014	….
Rahim	1000	books	5-3-2011	….
Ravi	500	pizza	23-11-2015	….

- Once DN received the blocks, it sends block report to NN to update meta data and sends an acknowledgement to HDFS client.
- Write a MR code to find total money spent by each customer and submit to the job client. Job client communicates to JT to get job ID, determines IS (2 here), and copies all job-related information onto HDFS.
- Job client now submits job to JT. Then, JT prepares execution plan. Map task is launched to the respective nodes which has required data blocks.
- IS prepares content of the blocks as records to check for record boundaries, so that RR can read records comfortably.

Step 2: RR (k1, v1) → mapper

RR is used to read record in key:value from IS. By default, text input format is used to read records. Therefore, key is byte offset, and value is entire line (until new line).

Preparing records in IS does not mean that key is attached with each line. Keys are created only when RR reads a line. Therefore, when a RR reads records from IS:

node1:

byte offset	name	price	item	date	
0	Ram	7000	mobile	12-2-2014	....
70	Ravi	1000	headset	10-2-2015	....
180	Raja	10000	laptop	2-1-2015	.....

node2:

byte offset	name	price	item	date	
2010	Ram	2000	charger	3-2-2014	....
2120	Rahim	1000	books	5-3-2011	....
2290	Ravi	500	pizza	23-11-2015	....
2340	Ram	100	fine	12-12-2015	....

Byte offset is a count of number of characters (each 1 byte size) of previous line + 1. Ex: first row key starts with 0 character. Second row key is number of characters in the first row + 1.

Third row key is number of characters in the first two rows+1… I have given key arbitrarily here as I am reluctant to calculate the number of characters in each line manually.

Step 3: mapper (k1, v1) → (k2, v2) → partitioner

RR gives the details of every customer transactions line by line to the map function. Particularly, our program filters out only the customer name and amount paid for each transaction.

Therefore, map task gives output as (customer name, price) key:value pairs. As we are going to calculate the total money spent by each customer, we have chosen customer name as map output key.

Mappers of node1 and node2 buffer their output in local memory. If overflew, then spills into local file system files as part-m-00001 in node1 and part-m-00002 in node2. Output of mapper is as follows in both nodes:

node1		**node2**	
part-m-00001		part-m-00002	
Ram	7000	Ram	2000
Ravi	1000	Rahim	1000
Raja	10000	Ravi	500
Ram	100		

Step 4: Partitioner (k2) → gives reduce node number → spilling process → combiner

We have only one reducer by default, therefore only one partition is formed from mapper output. So, each map task has only one partition corresponding to one reducer.

Partitioner gives reducer number as output if there are more than reduce tasks. There happens map output spilling process. There will be no change in key, value pairs in partition process.

Step 5: combiner (k2, list(v2)) → (k3, v3) → shuffle

Combiner is explained later. Further explanation is done with output of map task, not the output of combiner for more clarity. Still, you can follow the same procedure with combiner output.

Reduce phase

Step 6: shuffle → merge → sort

It doesn’t change the previous phase (key, value) output. It just transfers partitions to the node running reduce task (say node2) over network.

Reducer node is decided by JT either node1 or node2 or some other node… based on the load and task slots availability. Assume node1 partition is moved to node2. Output of shuffle in node2 is:

node2	
Ram	2000
Rahim	1000
Ravi	500
Ram	100
Ram	7000
Ravi	1000
Raja	10000

Step 7: sort (k3) → (k3, v3) → group

node 2

Rahim	1000
Raja	10000
Ram	2000
Ram	100
Ram	7000
Ravi	1000
Ravi	500

Step 8: group (k3)→ (k3, list (v3)) → reducer

node 2

Rahim	[1000]
Raja	[10000]
Ram	[7000, 2000, 100]
Ravi	[1000, 500]

Step 9: reducer (k3, list (v3)) → (k4, v4) → OutputFileFormat → RW

Since our aim is to find total money spent by each customer, we are going add all the values that belong to each key (customer).

node 2

Rahim	1000
Raja	10000
Ram	9100
Ravi	1500

Step 10: RW (k4, v4) → output file

Final output is written onto HDFS with file name part-r-00000. Output key and value are written with tab space as delimiter based on TextOutputFormat. Finally, client is notified about the job completion and job statistics with counters are displayed on the console.

node 2

part-r-00001

Rahim	1000
Raja	10000
Ram	9100
Ravi	1500

The delimiter “tab space” can be changed to any other symbol. Ex: to change delimiter to “,” you have to use mapred.textoutputformat.separator property.

Disk and network IO optimization

Combiner function is run during map output spill onto disk. Objective is to minimize the output before spilling into disks and move onto network to reach reduce node by using reducer/user defined code.

Step 5: combiner (k2, list(v2)) → (k3, v3) → shuffle

Following steps are carried out for combiner, which are similar to original MR flow.

mapper (k1, v1) → (k2, v2) → partitioner

partitioner (k2) → local sort

local sort (k2) → (k2, v2) → local group

local group (k2) → (k2, list(v2)) → combiner

combiner (k2, list(v2)) → (k3, v3) → shuffle

Combiner in detail

mapper (k1, v1) → (k2, v2) → partitioner

node 1		**node2**	
Ram	7000	Ram	2000
Ravi	1000	Rahim	1000
Raja	10000	Ravi	500
Ram	100		

partitioner (k2) → local sort

node 1		**node2**	
Ram	7000	Ram	2000
Ravi	1000	Rahim	1000
Raja	10000	Ravi	500
Ram	100		

local sort (k2) → (k2, v2) → local group

node 1	node2
Raja 10000	Ram 2000
Ram 7000	Ram 100
Ravi 1000	Rahim 1000
Ravi 500	

local group (k2) → (k2, list(v2)) → combiner

node 1	node2
Raja 10000	Ram [2000 100]
Ram 7000	Rahim 1000
Ravi 1000	Ravi 500

combiner (k2, list(v2)) → (k3, v3) → shuffle

node 1	node2
Raja 10000	Ram 2100
Ram 7000	Rahim 1000
Ravi 1000	Ravi 500

Go to step 5

Map phase

Step 1: Input Split (IS) or Splits

Submitted input file is physically divided into equal sized blocks and stored in various DNs based on rack awareness. DN brings blocks into memory to feed map task for execution. IS represents set of blocks to be processed by an individual Map task.

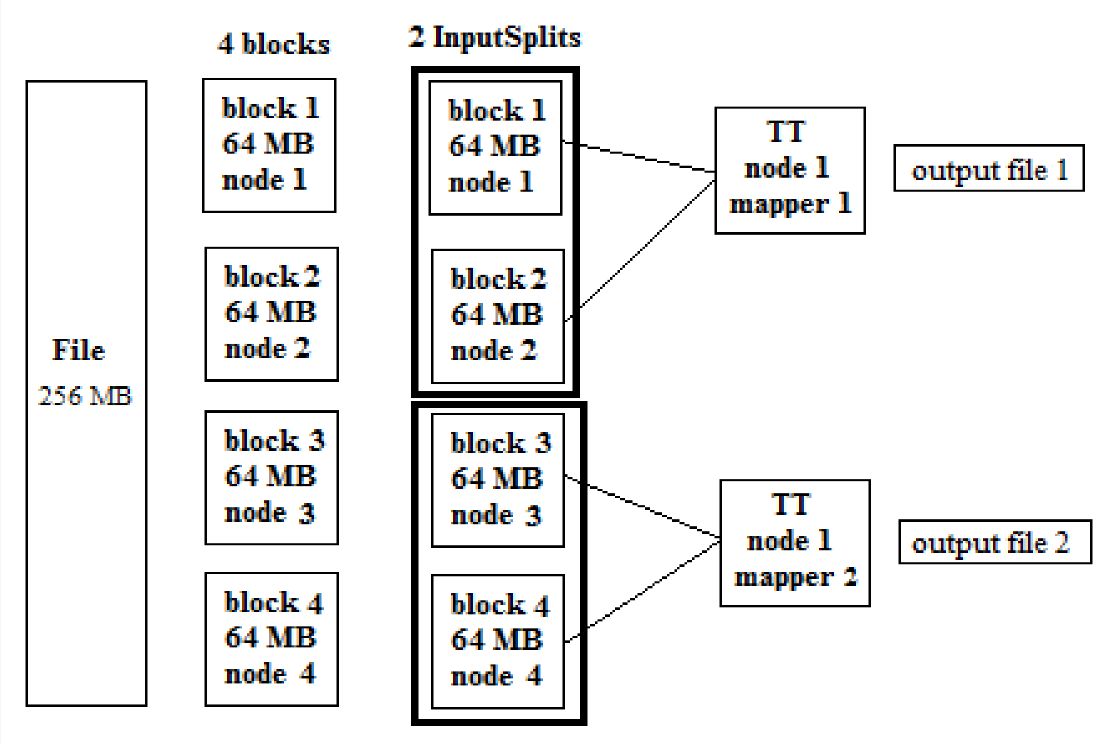
Block vs IS

- block is HDFS concept.
- IS is MR concept.

IS is the logical grouping of one or more physical blocks.

IS cannot be less than block size.

By default, block size and IS size are same.



Please note that IS doesn't contain copy of physical blocks. IS just contains the location of physical blocks and its meta-data. A map task can process only one IS at a time.

The number of map tasks for a given job is driven by the number of IS. Therefore, number of mapper tasks launched is equal to number of IS formed.

Example: number of map tasks launched for a job can be tuned by modifying block size. Assume input file size is 1 GB with varying block size.

Number of IS = file size / HDFS block size

= 1 GB / 64 MB = 16 map tasks

= 1 GB / 128 MB = 8 map tasks

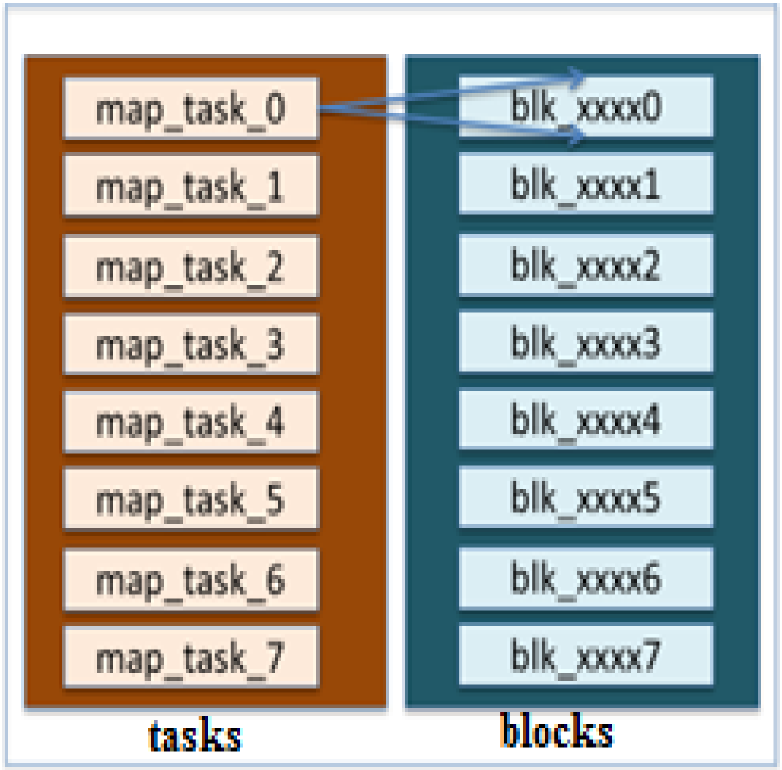
= 1 GB / 256 MB = 4 map tasks

As we know, by default, IS size is same of block size. If you modify IS size, then number of blocks for map input is affected. Ex: say IS size is 128 MB.

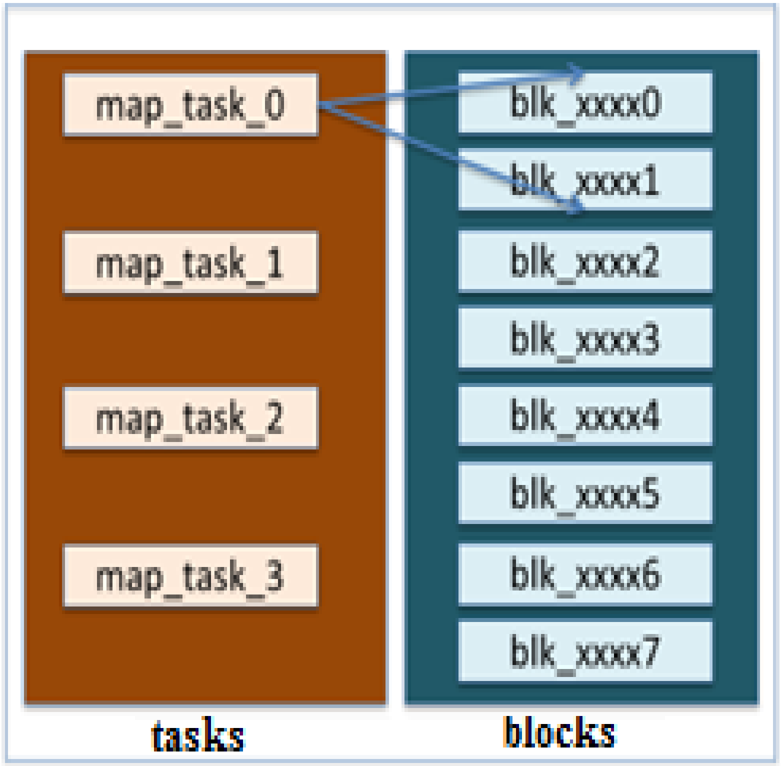
Number of IS = 1 GB / IS size = 128 MB = 8

Therefore, 8 map tasks are launched each processing 2 blocks. As you increase IS size, number of map tasks are reduced. However, latency of a task might increase due more blocks as there is no much parallelism.

File size : 1 GB
Block size: 128 MB
# blocks : 8
split size : 128 MB
# map tasks : 8



File size : 1 GB
Block size: 128 MB
# blocks : 8
split size : 256 MB
# map tasks : 4

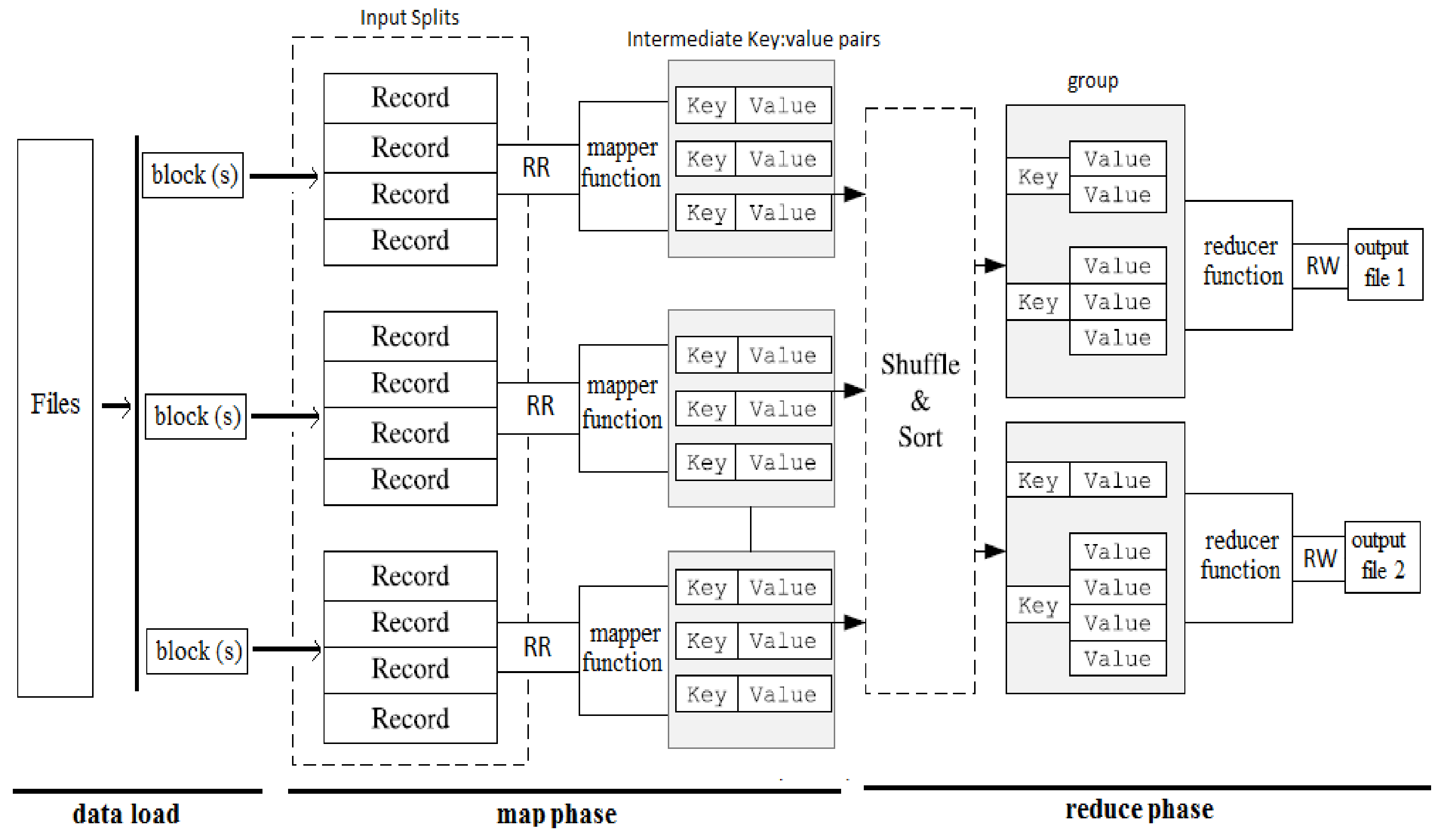


key-value pair

In MR, no data stands on its own. Every data is associated with key. Every key-value pair is called as record.

Splits and records are logical entities used at the time of job execution. They don't affect physical blocks.

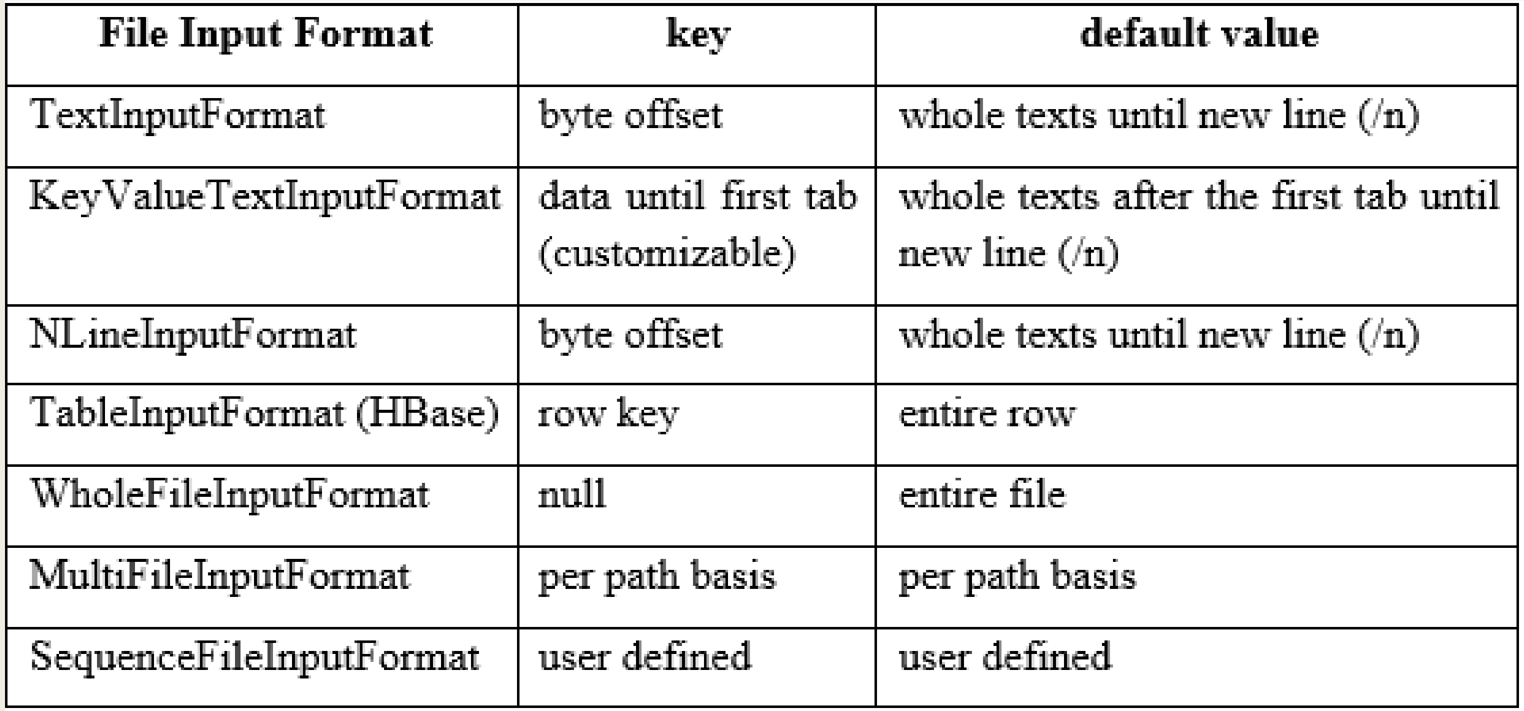
Every step in MR flow takes key-value pair as input and outputs key-value pairs as shown in Figure.



File Input Format

It is responsible for forming IS and preparing records from the contents of blocks.

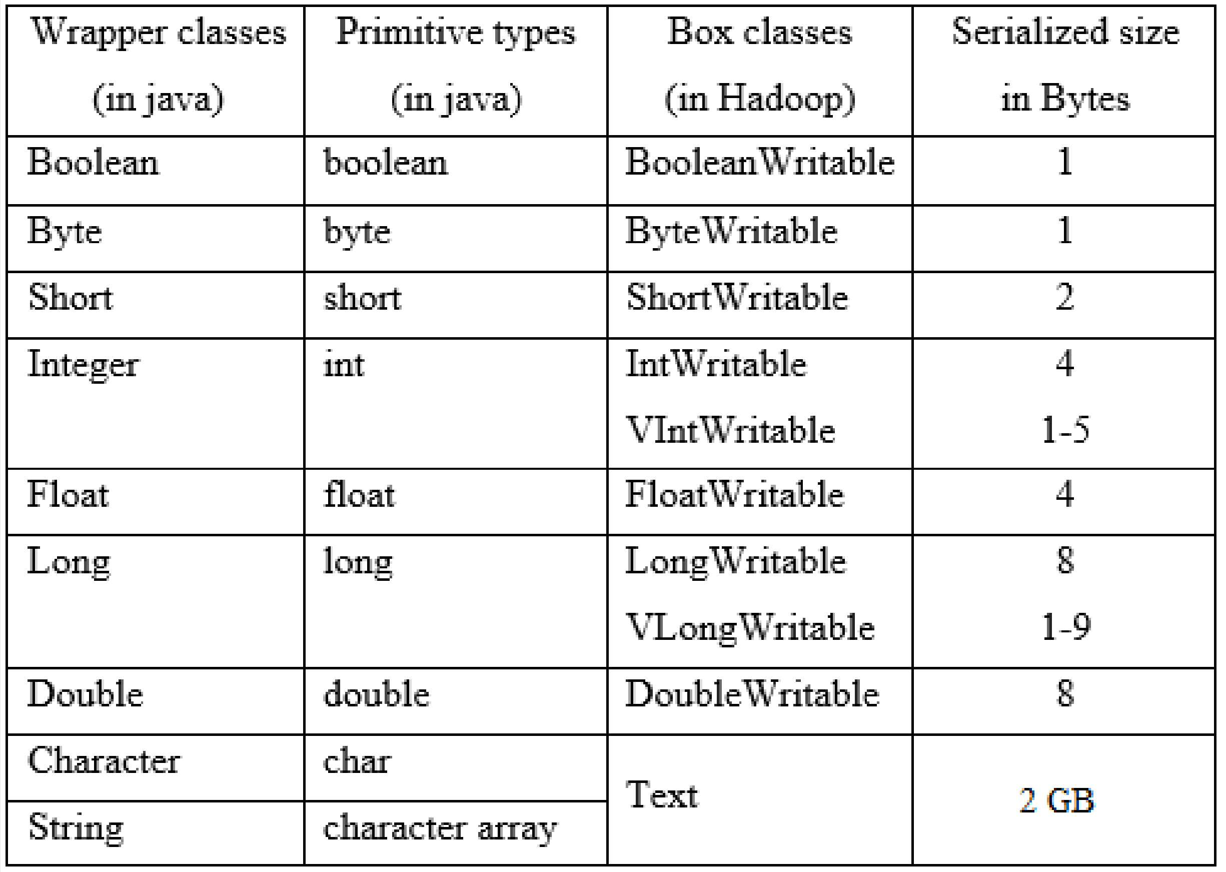
Every file input format defines what should be the datatype for key and value. Following table lists out key-value pairs of different file input format.



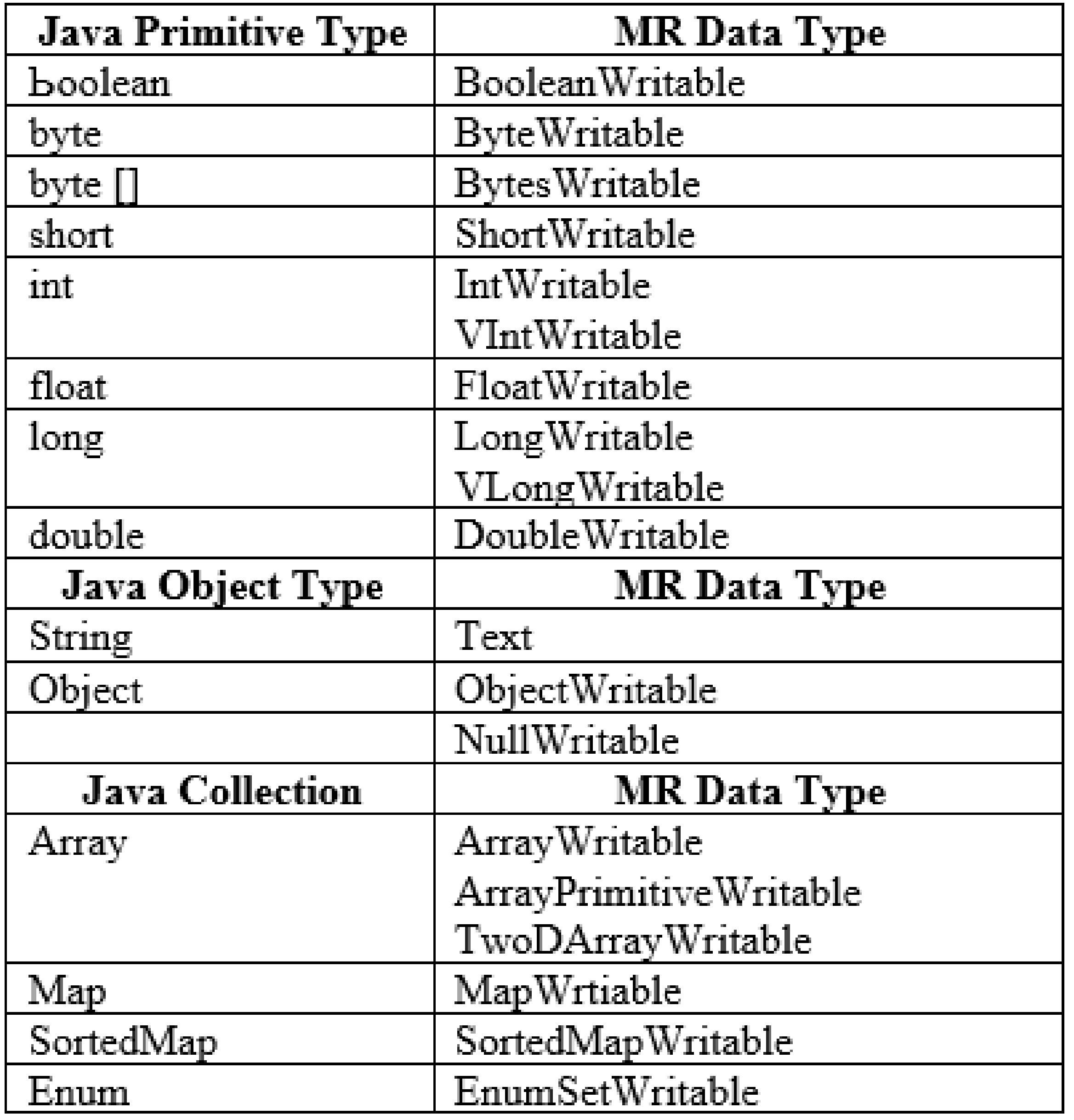
File Input Format	key	default value
TextInputFormat	byte offset	whole texts until new line (/n)
KeyValueTextInputFormat	data until first tab (customizable)	whole texts after the first tab until new line (/n)
NLineInputFormat	byte offset	whole texts until new line (/n)
TableInputFormat (HBase)	row key	entire row
WholeFileInputFormat	null	entire file
MultiFileInputFormat	per path basis	per path basis
SequenceFileInputFormat	user defined	user defined

Step 2: Record Reader (RR)

- Like every programming languages, MR programming also has data types to deal different data.
- While InputFormat and OutputFormat deal at file level, key-value types deal with data level. Both key and value are declared with a data type.

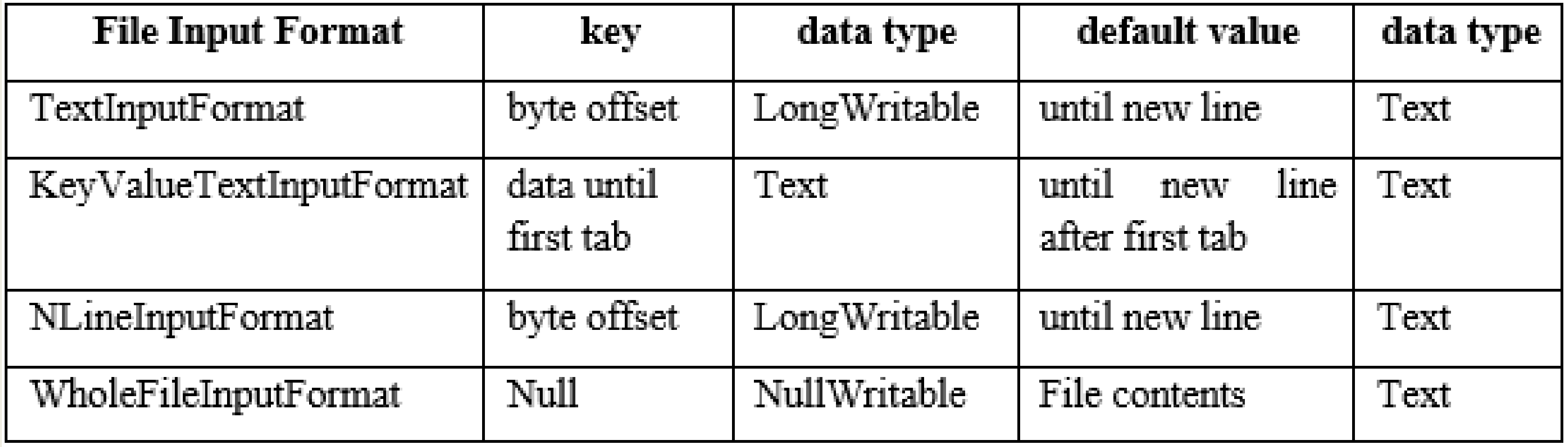


Wrapper classes (in java)	Primitive types (in java)	Box classes (in Hadoop)	Serialized size in Bytes
Boolean	boolean	BooleanWritable	1
Byte	byte	ByteWritable	1
Short	short	ShortWritable	2
Integer	int	IntWritable VIntWritable	4 1-5
Float	float	FloatWritable	4
Long	long	LongWritable VLongWritable	8 1-9
Double	double	DoubleWritable	8
Character	char	Text	2 GB
String	character array		



Java Primitive Type	**MR Data Type**
boolean	BooleanWritable
byte	ByteWritable
byte []	BytesWritable
short	ShortWritable
int	IntWritable VIntWritable
float	FloatWritable
long	LongWritable VLongWritable
double	DoubleWritable
Java Object Type	**MR Data Type**
String	Text
Object	ObjectWritable
	NullWritable
Java Collection	**MR Data Type**
Array	ArrayWritable ArrayPrimitiveWritable TwoDArrayWritable
Map	MapWrtiable
SortedMap	SortedMapWritable
Enum	EnumSetWritable

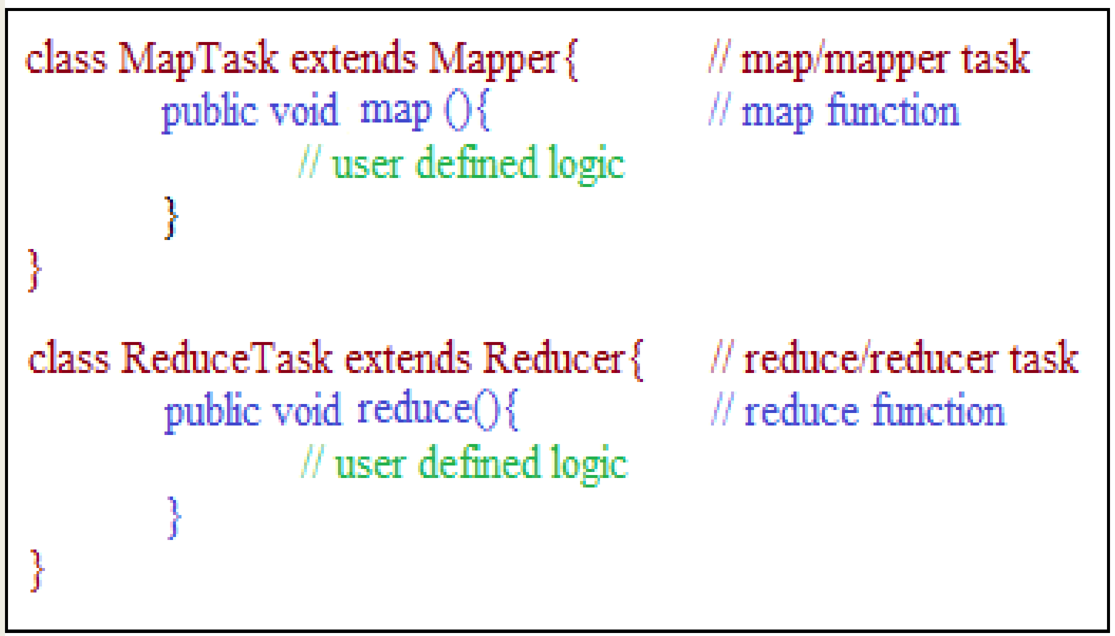
- IS prepares key:value pairs (records) from content of the block based on the file input format and gives byte oriented view. Map task does not know how to read those records from IS.
- RR reads <key, value> pairs from IS and converts byte oriented view of records to Hadoop data types and then feeds to map function.
- Therefore, data type of key and value in RR is determined based on file input format.
- Each InputFormat has to provide its own RR implementation to read key/value pairs.
- Ex: for TextInputFormat, IS forms byte offset as key and entire line (until new line) as value. RR will convert key (byte offset) to LongWritable and value (entire line) to Text and feed to map task.



File Input Format	key	data type	default value	data type
TextInputFormat	byte offset	LongWritable	until new line	Text
KeyValueTextInputFormat	data until first tab	Text	until new line after first tab	Text
NLineInputFormat	byte offset	LongWritable	until new line	Text
WholeFileInputFormat	Null	NullWritable	File contents	Text

Step 3: Mapper

- Mapping is also a kind of pre-processing most of the time. Therefore, output of mapper is called as intermediate key-value pairs, but not the result of MR Job.
- You can parse records and extract only relevant fields (projection) or something of interest, filtering unwanted/bad records, transform the incoming record…
- While coding for MR, we call map/reduce as function. When map/reduce is executed we call them as task.
- However, map/reduce task contains a function to override to include our logic.
- Map task is invoked only once at the time of launch. But, map function is invoked for every input record.



```
class MapTask extends Mapper{           // map/mapper task
        public void  map (){            // map function
                // user defined logic
        }
}

class ReduceTask extends Reducer{       // reduce/reducer task
        public void reduce(){           // reduce function
                // user defined logic
        }
}
```

Step 4: Partitioner (load balancing map outputs among reducers)

- Partitioner load balances map output across multiple reduce tasks. Partition is a portion of map output that goes to a reducer.
- Partitioner splits the output of a map task (or combiner if executed) into several partitions. The number of partitions is equal to the number of reduce tasks launched.
- Partitioner is meaningful only when we launch more than one reduce tasks.
- Objective of partitioning is to bring the same key from different map tasks into one reducer. Ex: there are 10 map tasks for wordcount, say map task 1, 4, 6, and 10 produces word “Hadoop” as output.
- Now, in order to find count of word “Hadoop”, that particular word should be brought from map tasks 1, 4, 6, and 10, to reducer task 1. Therefore, word “Hadoop” can be counted as 4.

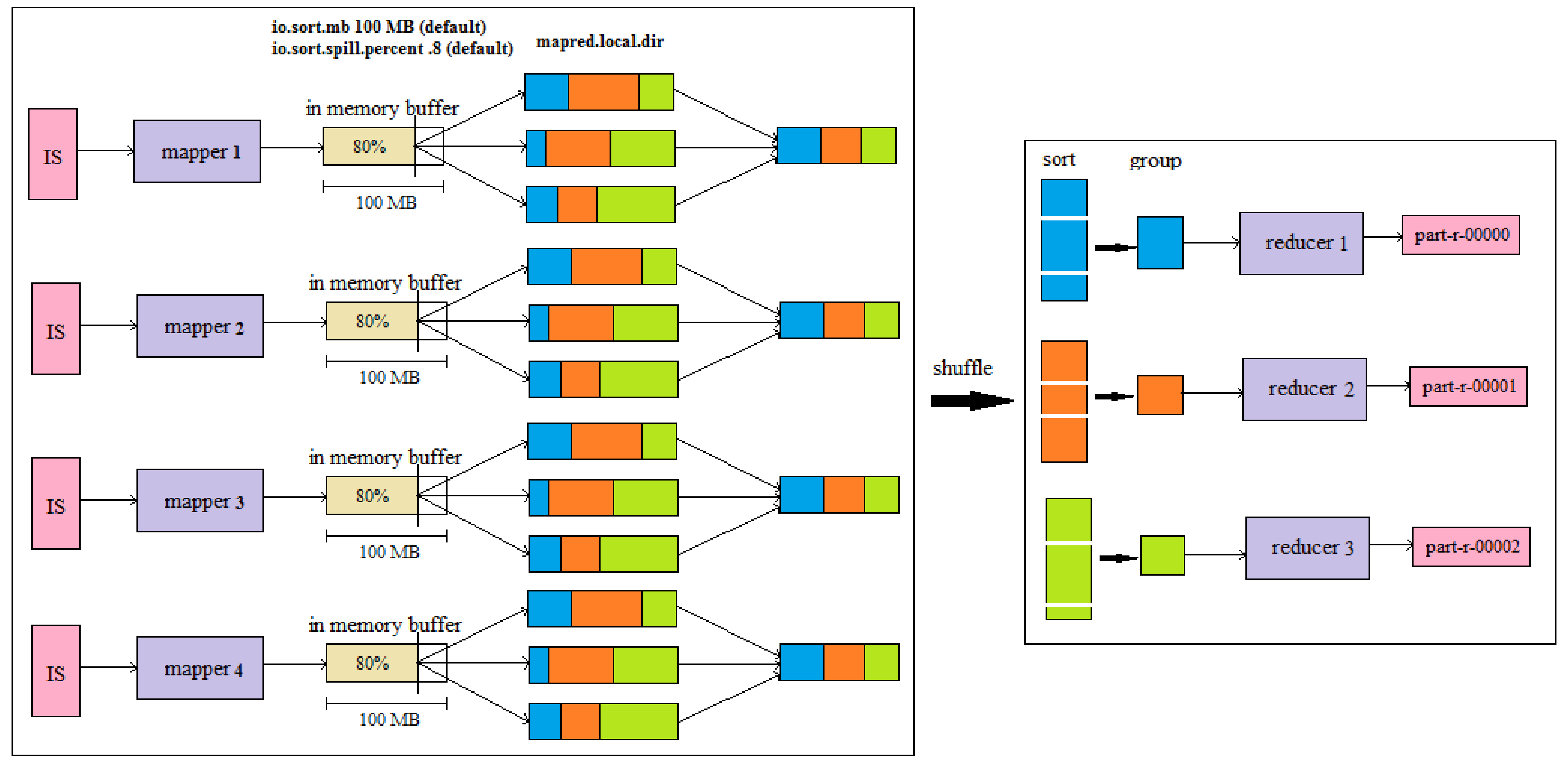
- Default paritioner is HashPartitioner, which works so well with any number of partitions and ensures each partition has a good mix of keys, leading to more evenly sized partitions.
- Sometimes, it might cause skewness. We can define our own partitioner programmatically to decide which partition to go which reducer. There are different other partitioners:

BinaryPartitioner	– partitions binary data
KeyFieldBasedPartitioner	– taking any field of a record to partition
TotalOrderPartitioner	– helps to equally distribute the map outputs to reducers.

Step 5: Combiner (network and disk IO optimization)

- Combiner task reduces network traffic, disk IO, and number of records processed by reduce task. If map task generates huge output, then to transfer them to reduce task over network introduce huge network traffic.
- Secondly, until map outputs transferred over network, they have to be spilled onto local file system in the disks. It requires more disk IO. Finally, reduce task also will have to process huge data, so latency will be high.
- Therefore, responsibility of combiner task is to process map output and reduce the size locally, so that very less data to spill into disk and transfer over network to reducer.

- It is also called mini/local reducer. If there is no way to reduce the map output, then programmer should not enable combiner. Therefore, it is application specific and enabled by programmer.
- When map output size goes beyond the in-memory buffer threshold, it is divided into partition, and each partition is sorted. Then, values belong to same key are grouped to remove key redundancy.
- Partitioning, sorting, and grouping are done based on the map output key. Combiner task is executed as a separate thread simultaneously with map task on these partitions. Then combiner output is spilt into local disk. Each mapper has its own combiner function to execute.


io.sort.mb 100 MB (default)
io.sort.spill.percent .8 (default)
mapred.local.dir
IS
mapper 1
in memory buffer
80%
100 MB
IS
mapper 2
in memory buffer
80%
100 MB
IS
mapper 3
in memory buffer
80%
100 MB
IS
mapper 4
in memory buffer
80%
100 MB
shuffle
sort
group
reducer 1
part-r-00000
reducer 2
part-r-00001
reducer 3
part-r-00002

After map task is over, there can be several spill files in local disk over period of time. If there are at least three spill files, then again those files are merged, sorted, grouped, and combiner is run one more time.

If there are just two spills for a map task, then the potential reduction in map output size is not worth than the overhead of invoking combiner.

If combiner task is enabled, it is executed at least once in each map task. Combiner runs a user-defined combiner function or can be same as reduce task.

While writing user defined combiner task , the signature of combiner task must be identical to reduce task.

However, all applications cannot use reduce task as combiners.

The number of times combiner task launched is equal to number of spills +1.

The number of times combiner function invoked is sum of number of key:(list of values) in each partition.

Only application that satisfies additive and commutative property can use reduce task itself as combiner. Ex: sum, max, min…

Commutative: a + b = b + a (swapping operands result the same)

Associative: a + (b + c) = (a + b) + c (changing operands in group result the same)

Ex: 1. Max function works: max (max(a,b), max(c,d,e)) == max (a,b,c,d,e)

2. Mean function does not work: mean(mean(a,b), mean(c,d,e)) != mean(a,b,c,d,e)

For mean function, associative property is failed. So, reduce function cannot be used as combiner function. Combiner can be mainly used for aggregation scenarios (sum, count…), not for all applications like joining datasets and sorting...

Combiner task is not automatically invoked. We have to specify combiner in job to execute.

Reduce phase

- This phase includes the magical steps (shuffle + merge + sort + group) that show the power of Hadoop MR. Magical steps are executed only if reduce task is enabled and executed where reduce task will be run.
- These are completely managed by MR framework itself. However, programmers can customize. Please note that shuffle, sort, group are logical operations only. They don’t change the original output of mappers.

- The reduce phase begins when some % of map tasks completed. It does not mean that the reduce method can begin.
- In general, reduce phase moves output (partitions) of mapper/combiner to reducer nodes, merges all the partitions, sorts based on key, groups all the values that belong to the same key to eliminate key redundancy, and finally records (key:list(values)) are fed into reduce function.

Step 6: Shuffle

- Shuffle is the process by which partitioned mapper/combiner output is transferred over HTTP network to 1 or more TT where reducer is going to be run.
- Each reducer node receives 1 or more partitions from all map tasks. Each reduce task has 5 parallel threads running to fetch map output by default.
- If reduce task runs in same node where map task completed, then shuffle doesn’t have any role.
- But, how do a reducer node knows which node to query to get its partitions? This happens through the JT.
- As each Mapper task completes, it notifies the JT about the partitions. Each reducer periodically queries the JT for Mapper hosts until it receives its partitions.
- Consider 2 TB input file. What if mapper output size is same as its input? Then moving this 2 TB to reducer across network requires huge bandwidth. That is the reason, combiner is run to reduce the size of map output to move across network.
- To reduce size of map output further, it can be compressed. Snappy is a compressor used in MR most commonly. Other options are: DEFLATE, LZO, LZ4… More detail on compression is given in MRv2 section.

Step 7: Sort

Firstly, the map output partitions are copied to the reduce task’s JVM. If memory is not sufficient, then they are spilt into local disk.

As soon as partitions from all map tasks arrived to reducer nodes, all partitions should be merged into one file for further processing. Every 10 spilled files are merged by default.

Merged file should be sorted based on key.

Step 8: Group

Set of values that belong to a same key is grouped to eliminate key redundancy.

Number of times reduce function executed is equal to number of key: list(values) pairs after grouping.

If there exist duplicate keys, then reduce function is repeatedly invoked for every duplicate key.

If same key is present in in multiple spilt files, then, for a single key so many spilt files have to be brought into memory (needs multiple passes).

It involves lot of IO. That is the reason, we sort and group keys before calling reduce function. Therefore, only one pass is required for each key and only one time reduce function is invoked for each key.

Step 9: Reducer

- Reduce function processes list of values for a key and produces zero or more output records.
- Core algorithm is implemented in reduce function. Aggregation and join operations are performed here.
- The number of times reduce function is executed is equal to the number of (key: list of values) pairs after grouping.
- Mapper output is not deleted until reducer picks up map result. Mostly, map output is deleted only upon the successful completion of reducers.

- If node running map task fails before the map output has been consumed by reduce tasks, then Hadoop will automatically rerun the map task to re-create the map output.
- JT decides in which node reducer to be run. Whether in the node where map tasks done or in some other node in the same rack or some other rack?
- It depends on the task slots availability and the load of TT.

- If you don’t specify number of reduce tasks, then by default one identity reduce task is launched. It results the same of map output, but in sorted manner as it passes through shuffle, sort, and group.
- If you set number of reduce tasks as zero, then map output itself is considered to be job output and will be stored in HDFS. There will be no shuffle, sort and merge process.
- To decide the number of reducers, one need to understand the map output and based on the amount of parallelism required.
- There should not be too many reduce tasks doing very little work and should not be too few also.
- Reduce tasks don’t have the advantage of data locality unlike map tasks, because, it has to receive partitions from various mappers running in different nodes. But, it is rack aware.
- Output of reduce task is generally stored in HDFS by default with replication.
    - first copy is stored locally where reducer task runs
    - second copy is stored in the local rack
    - third copy is stored in any node in some other rack in the cluster.
    - Therefore, writing reduce output consumes network bandwidth, as much as normal HDFS write pipeline consumes.

Key points

- If your application does not require sorting, grouping, aggregation, then you need not use reduce task at all. Some applications will require only map tasks as pre-processing.
- Don’t perform any aggregation in map task and don’t filter any records in reduce task.
- Number of mappers is determined by MR framework and number of reducers cannot be determined by MR framework, but user can decide.
- If no mapper/reducer task is specified, then identity mapper/reducer task is run which results the input key and input value as output key and output value.
- Combiner and reduce functions get unique keys in sorted order as input.

By default,

- number of blocks == number of IS == number of mapper == number of combiner
- number of partition == number of reducer == number of output file.

Step 10: Record Writer (RW)

- Reduce function feeds its output key-value to RW, which writes them in HDFS with tab separation by default based on TextOutputFormat.
- Each reducer writes an output file onto HDFS with desired RF and rackawareness.
- Output of map task is spilt into local file system, because writing intermediate results onto HDFS will lead to unnecessary replication and leads extra work to delete them later.
- But, output of reduce task is written onto HDFS as it is final result and should be fault tolerant. Output files of map/reduce are follow naming convention.

  part-m-00000 → "m" stands for map/combiner output file.

  part-r-00000 → "r" stands for reduce output file.

- Part denotes part of a file. 5 digits indicate that upto 99999 output files can be created.
- Please note that output of combiner file also is in the name of mapper output.

MR Execution flow

Before we discuss more about MR, we need to understand how MR jobs are carried out as shown in below Figure.

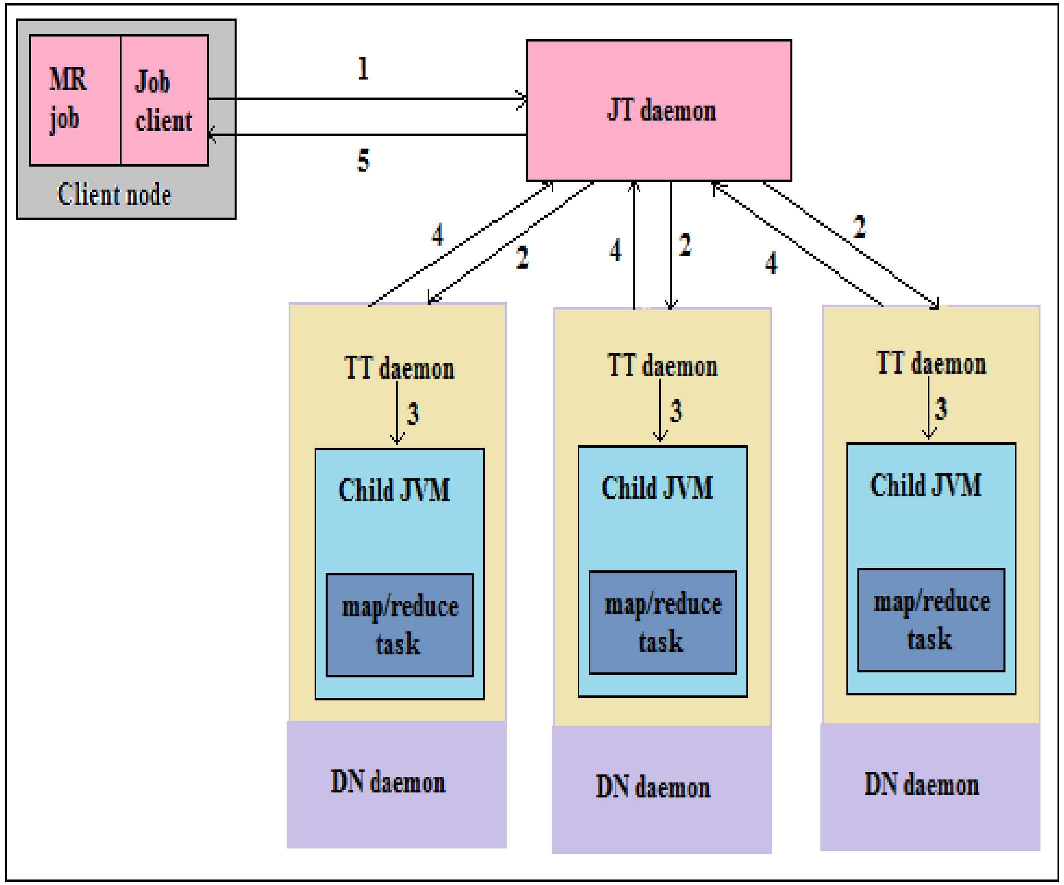
1. User submits a job to Job client, which in turn submits to JT.

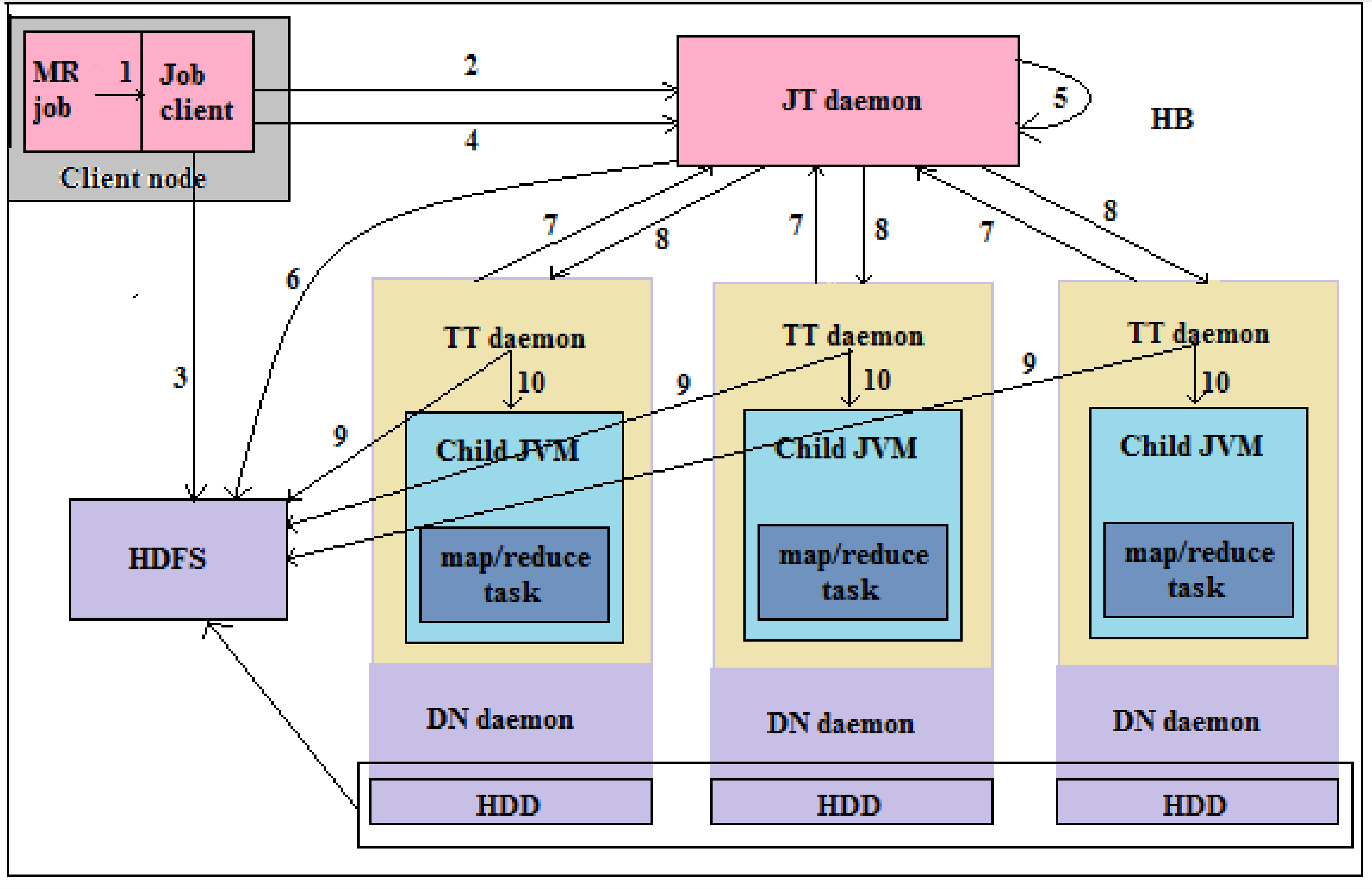
2. JT prepares execution plan, distributes tasks, and co-ordinates phases.

3. TT launches map/reduce tasks.

4. TT sends progress of map/reduce tasks and HB to JT.

5. JT sends job progress status and completion message to client.




MR job
1
Job client
Client node
2
4
JT daemon
5
HB
6
7
8
7
8
7
8
3
TT daemon
10
9
Child JVM
map/reduce task
TT daemon
10
9
Child JVM
map/reduce task
TT daemon
10
9
Child JVM
map/reduce task
9
HDFS
DN daemon
DN daemon
DN daemon
HDD
HDD
HDD

Job Submission

1.Upload data onto HDFS before you launch job. User submits a job to Job client.

2. Job client goes for various checks whether input file and output directory in HDFS exists or not by interacting NN. If input file doesn’t exist and/or output directory is already available, then Exception is thrown and job is terminated. If it is successful, then it interacts with JT to get jobID and calculates input splits.

3. All job-related information such as job.xml, job file, and IS information are copied to 10 DNs by default to be highly available for execution.

4. Once all information is moved onto HDFS, Job client creates an instance of JobSubmitter, submits job to JT to begin job life cycle, and output directory is created. However, output files of MR job are created only at the end of job life cycle.

Job Initialization

5. Calculating IS by Job client object may be slower. So, you can set JT to determine IS by communicating NN. Once, this is done job is put into job queue. Then, MR job scheduler picks the job from job queue and creates an object which encapsulates its tasks.

6. JT retrieves IS information from HDFS (if calculated by Job client) and prepares execution plan based on IS, locality of data blocks. Every block has three copies by default. So, JT prepares a list (for block1 say DN_2, DN_1, DN_3) for each block. DN_2 comes first in the list as it is very near to JT based on rack awareness.

Task Assignment

7. TT runs an infinite loop that periodically sends Heartbeat and task status information to JT.

8. Scheduler attempts to launch map task on block1 in TT_2. If there is no free slot in TT_2, then TT_1 is attempted… if there are no free slots in three TTs mentioned in the list for block1 to achieve data locality, then block1 is moved to any other TT which has free map/reduce slots.

Job Assignment

9. TT localizes job related information such as jar file, and job.xml file from HDFS.

10. TT launches JVM for running map/reduce tasks. NN instructs DN to bring required block from HDFS and loads into JVM.

Percentage of tasks of a job completed is updated regularly, which a user can see on WebUI. Once all map and reduce tasks are done, JT updates the job status as success, and notifies the Job client. In the end, WebUI will present all statistical and runtime information… Finally, JT instructs TT to remove JVM and reclaim resources safely.

Job’s metrics

- Upon job completion, JT displays set of information/statistics on the console. These statistics are collected using counters.
- There are two types of counters: built-in counter (job, task, and file level), and user defined counters.
- TT, DN aggregates all counters and send to their masters. These information’s are retained in job history for later use.
- Counters are used to determine data quality, to diagnose and debug problems. It is implemented using Enums.
- User can create their own counters to enumerate something of interest. Counters can be accessed using WUI, java API’s, and CLI. Several built-in counters are:

  Job counters: number of map and reduce tasks launched, number of failed tasks…

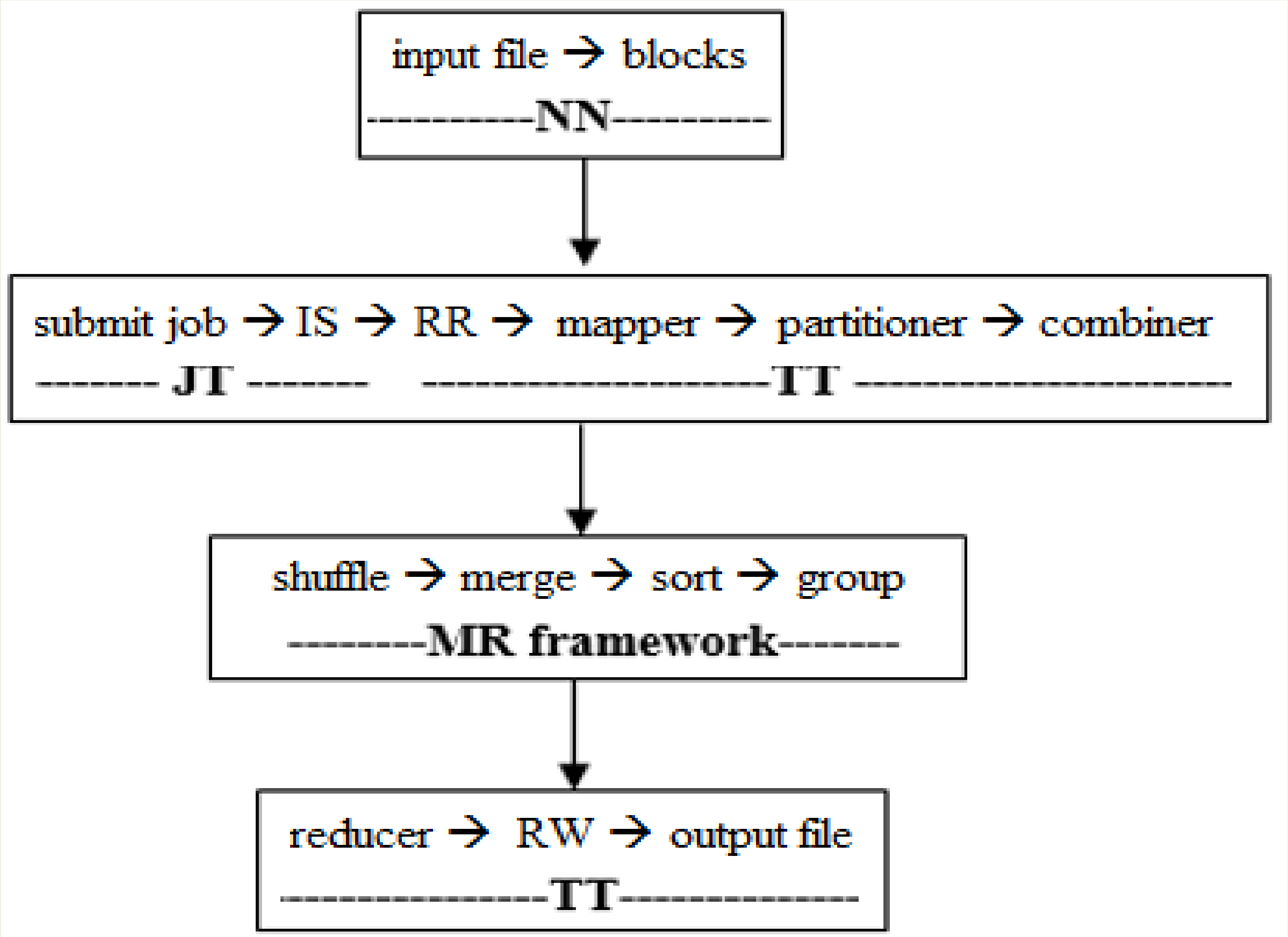
  File system Counters: number of bytes read and written…

  MR framework counters: network, and memory related statistics…

Job progress calculation

- map phase: the number map tasks completed in a job.
- reduce phase: 0-33% means shuffle (shuffle), 34-66% is sort (sort + group), 67%-100% is reduce task.
- If only half of the reduce tasks completed, then it is 1/3+1/3+1/6=83%.

Daemons that execute MR steps



input file → blocks

----------NN---------

submit job → IS → RR → mapper → partitioner → combiner

------- JT ------- --------------------TT ---------------------

shuffle → merge → sort → group

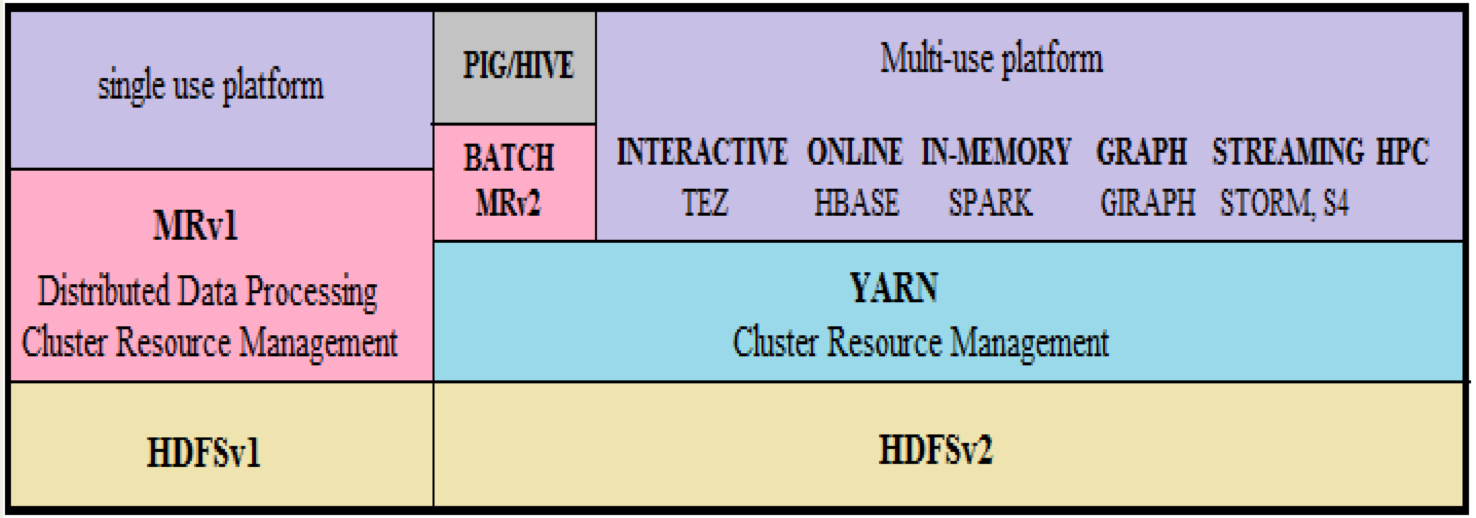
--------MR framework-------

reducer → RW → output file

---------------TT--------------

Hadoop 2.x

- If MRv1 is installed in a cluster, no other data processing framework can’t be deployed.
- It doesn’t share the cluster resources to other co-located data processing tools such as stream processing, in-memory computing…
- Therefore, MRv2 splits MRv1 functionalities into two components to exploit more scalability and resource sharing.
- Hadoop 2.x enables a cluster of commodity servers to share HDFS data and cluster resources dynamically between MR and other data processing frameworks using Yet Another Resource Negotiator (YARN).
- It allows fine-grained (sharing resources in any proportion upon its availability) resource sharing for better cluster utilization.



single use platform

MRv1

Distributed Data Processing

Cluster Resource Management

HDFSv1

PIG/HIVE

BATCH MRv2

Multi-use platform

INTERACTIVE TEZ

ONLINE HBASE

IN-MEMORY SPARK

GRAPH GIRAPH

STREAMING STORM, S4

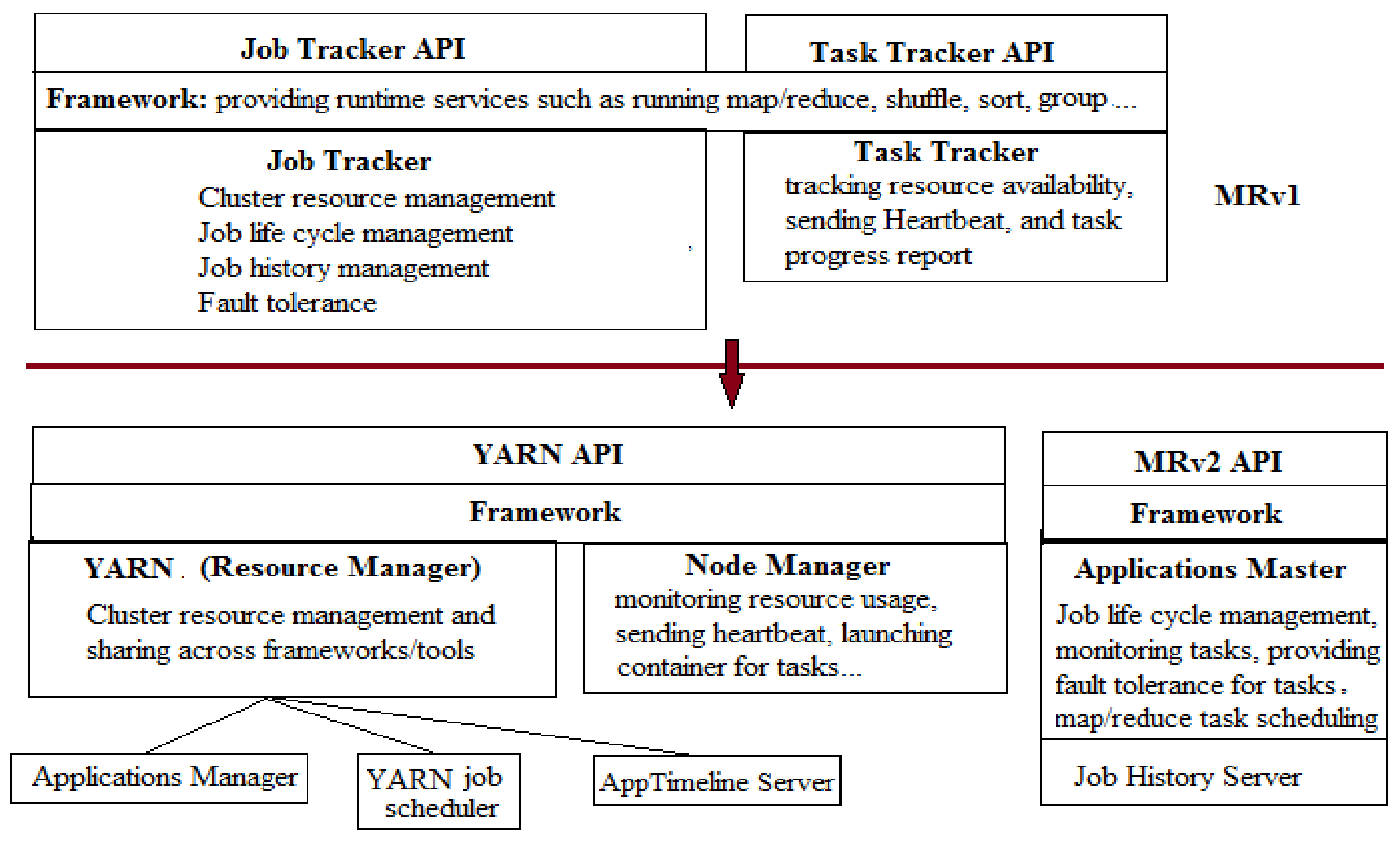
HPC

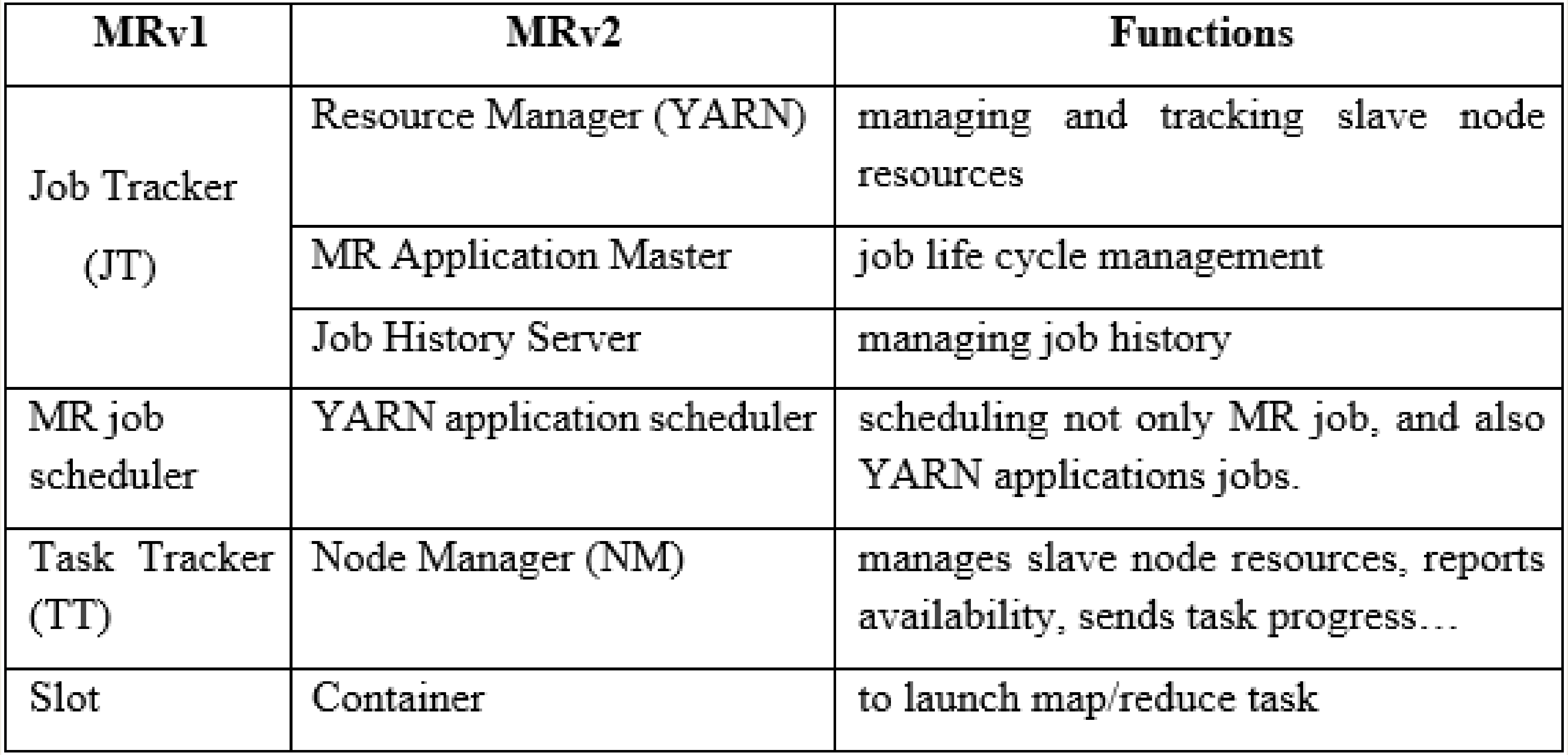
YARN

Cluster Resource Management

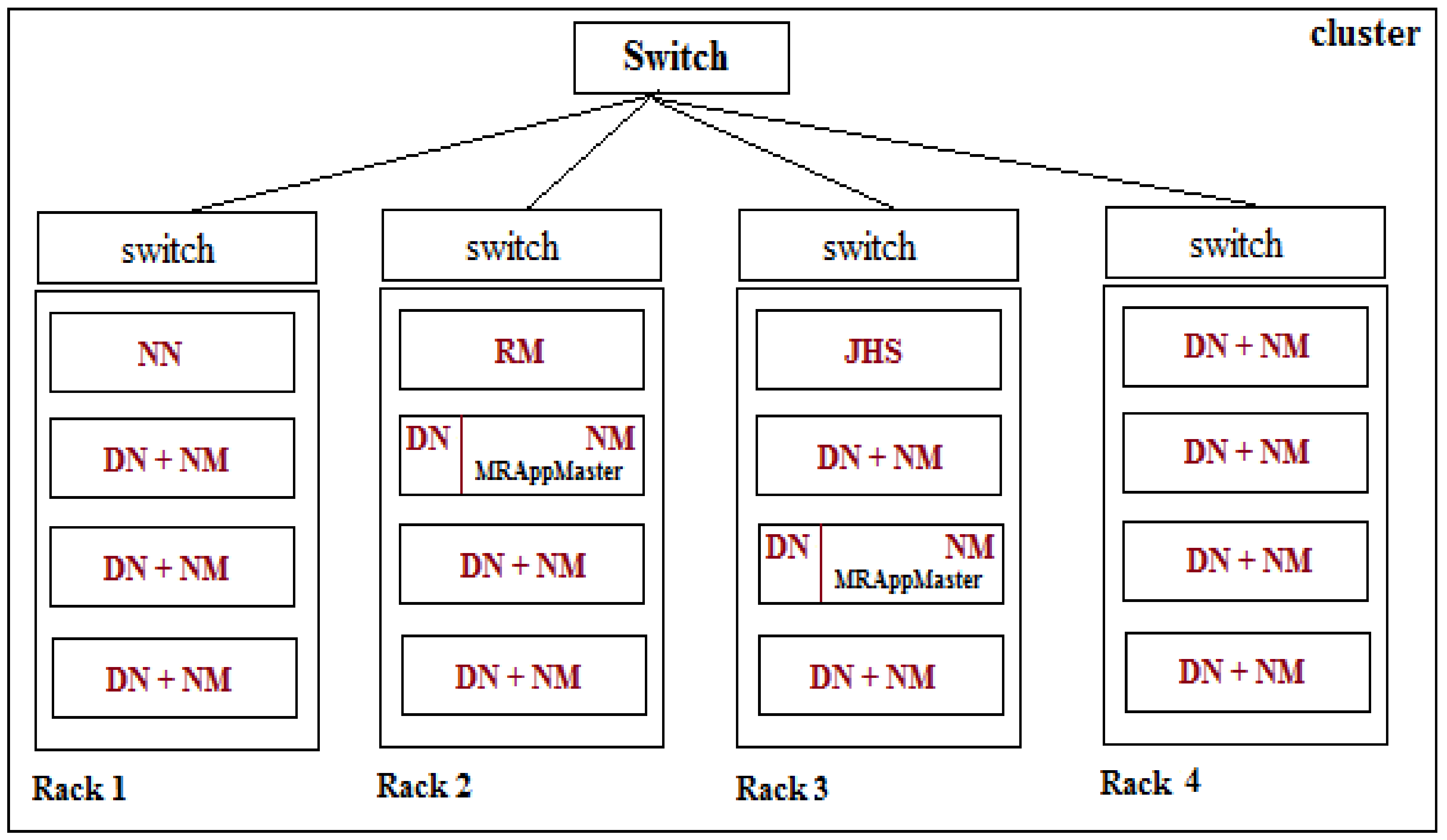
HDFSv2

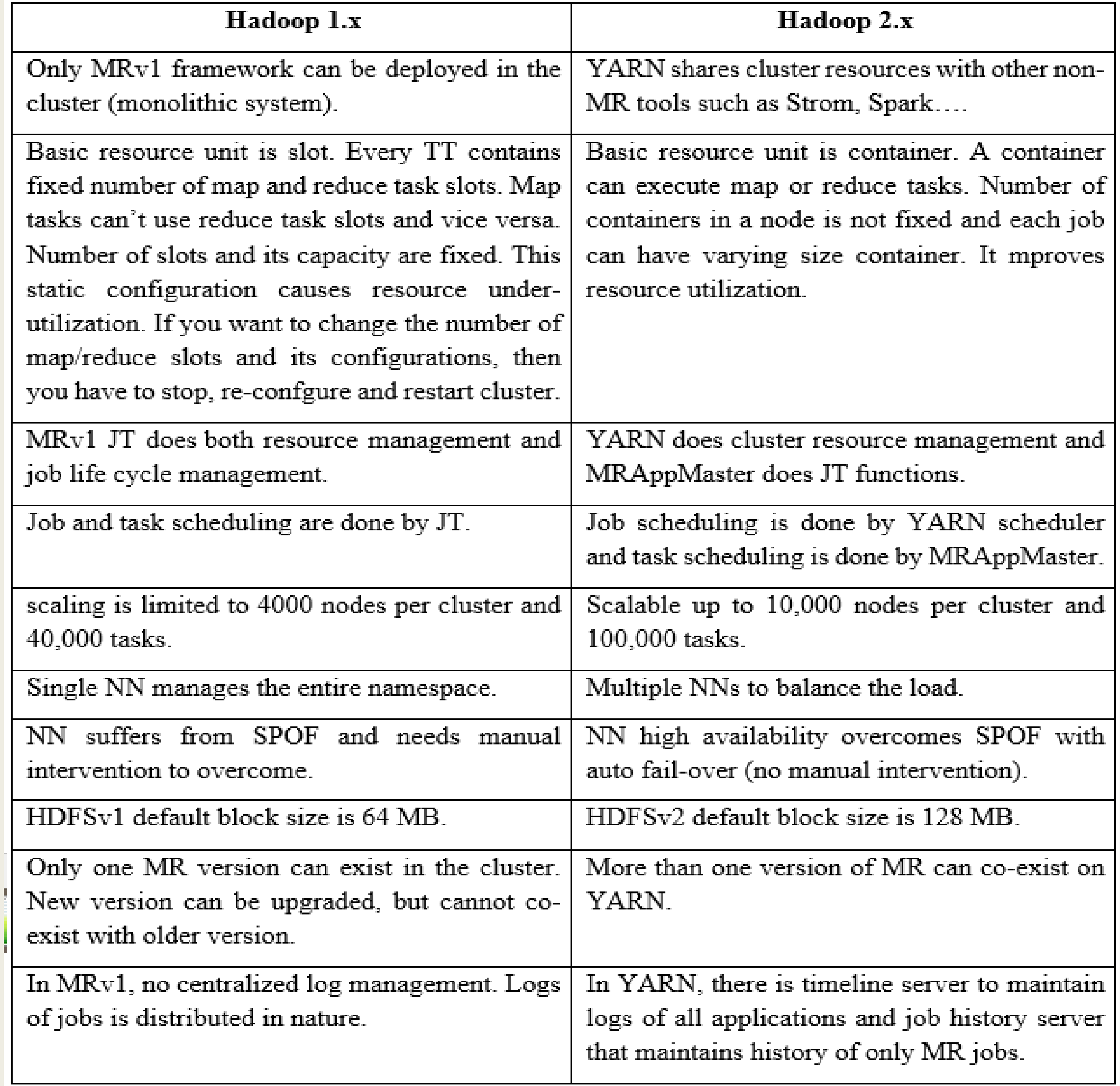
MRv1 vs YARN components


Job Tracker API
Task Tracker API
Framework: providing runtime services such as running map/reduce, shuffle, sort, group ...
Job Tracker
Cluster resource management
Job life cycle management
Job history management
Fault tolerance
Task Tracker
tracking resource availability, sending Heartbeat, and task progress report
MRv1
YARN API
Framework
YARN . (Resource Manager)
Cluster resource management and sharing across frameworks/tools
Node Manager
monitoring resource usage, sending heartbeat, launching container for tasks...
MRv2 API
Framework
Applications Master
Job life cycle management, monitoring tasks, providing fault tolerance for tasks , map/reduce task scheduling
Job History Server
Applications Manager
YARN job scheduler
AppTimeline Server



MRv1	MRv2	Functions
Job Tracker (JT)	Resource Manager (YARN)	managing and tracking slave node resources
	MR Application Master	job life cycle management
	Job History Server	managing job history
MR job scheduler	YARN application scheduler	scheduling not only MR job, and also YARN applications jobs.
Task Tracker (TT)	Node Manager (NM)	manages slave node resources, reports availability, sends task progress…
Slot	Container	to launch map/reduce task


cluster
Switch
switch
switch
switch
switch
NN
DN + NM
DN + NM
DN + NM
RM
DN
NM
MRAppMaster
DN + NM
DN + NM
JHS
DN + NM
DN
NM
MRAppMaster
DN + NM
DN + NM
DN + NM
DN + NM
DN + NM
Rack 1
Rack 2
Rack 3
Rack 4



Hadoop 1.x	Hadoop 2.x
Only MRv1 framework can be deployed in the cluster (monolithic system).	YARN shares cluster resources with other non-MR tools such as Strom, Spark….
Basic resource unit is slot. Every TT contains fixed number of map and reduce task slots. Map tasks can't use reduce task slots and vice versa. Number of slots and its capacity are fixed. This static configuration causes resource under-utilization. If you want to change the number of map/reduce slots and its configurations, then you have to stop, re-confgure and restart cluster.	Basic resource unit is container. A container can execute map or reduce tasks. Number of containers in a node is not fixed and each job can have varying size container. It mproves resource utilization.
MRv1 JT does both resource management and job life cycle management.	YARN does cluster resource management and MRAppMaster does JT functions.
Job and task scheduling are done by JT.	Job scheduling is done by YARN scheduler and task scheduling is done by MRAppMaster.
scaling is limited to 4000 nodes per cluster and 40,000 tasks.	Scalable up to 10,000 nodes per cluster and 100,000 tasks.
Single NN manages the entire namespace.	Multiple NNs to balance the load.
NN suffers from SPOF and needs manual intervention to overcome.	NN high availability overcomes SPOF with auto fail-over (no manual intervention).
HDFSv1 default block size is 64 MB.	HDFSv2 default block size is 128 MB.
Only one MR version can exist in the cluster. New version can be upgraded, but cannot co-exist with older version.	More than one version of MR can co-exist on YARN.
In MRv1, no centralized log management. Logs of jobs is distributed in nature.	In YARN, there is timeline server to maintain logs of all applications and job history server that maintains history of only MR jobs.

Yet Another Resource Negotiator (YARN)

- YARN is essentially a software system for managing different distributed framework.
- It manages and shares cluster resources in a fine-grained manner among different frameworks deployed in the same cluster.
- YARN doesn’t know what kind of applications it is running. It can be MR or spark or storm…
- YARN pools and shares resources across frameworks (as done in cloud computing).
- It is based on master slave architecture.
- YARN sub components are: Resource Manager (RM), Node Manager (NM)

Resource Manager (RM)

It is a master service and centralized resource management in the cluster.

It manages resources of a cluster, instructs NM to launch containers for map/reduce tasks.

There is only one RM in the cluster.

Specific RM functionalities are:

- manages slave’s resources.
- handles resource requests.
- provides security with kerberos.

RM sub components are:

- Applications Manager
- AppTimeLine Server
- YARN Scheduler

Applications Manager (AsM):

it is responsible for accepting job-submissions, negotiating a container for executing MRAppMaster and releases that container when job gets over.

It provides fault tolerance to MRAppMaster.

YARN scheduler:

YARN scheduler is responsible for allocating resources to the various running frameworks subject to familiar constraints of capacities, queues…

MRv2 is just a programming paradigm and doesn’t have any job scheduling mechanism.

Since all frameworks hosted on YARN is called YARN applications, MR jobs is called as applications with globalID.

AppTimeline Server: YARN has global job history server that records job history of all frameworks (not only MR) running in the cluster.

Node Manager (NM)

Every slave node runs a NM, which manages slave resources such as memory, CPU, IO… NM communicates with the RM to register itself to be part of YARN cluster.

Then, NM sends resource availability to RM via heartbeat. NM responds to the requests from RM and MRAppMaster.

Its job is to create, monitor, and kill containers for MRAppMaster and map/reduce tasks on receiving instrctions from RM.

It monitors resource usage of containers, and kills them if trying to consume more resources.

It provides local logging services to applications.

NM runs auxiliary services such as shuffle, sort, and group during task execution.

YARN client: It is a light weight process that interacts with AsM to submit job and to store job meta-data in HDFS.

YARN child: It is a light weight process that runs to help containers.

Its particular responsibilities are to localize task information from HDFS for map/reduce, and to send task status information to MRAppMaster periodically.

In contrast, in MRv1, TT itself sends task report to JT.

Container

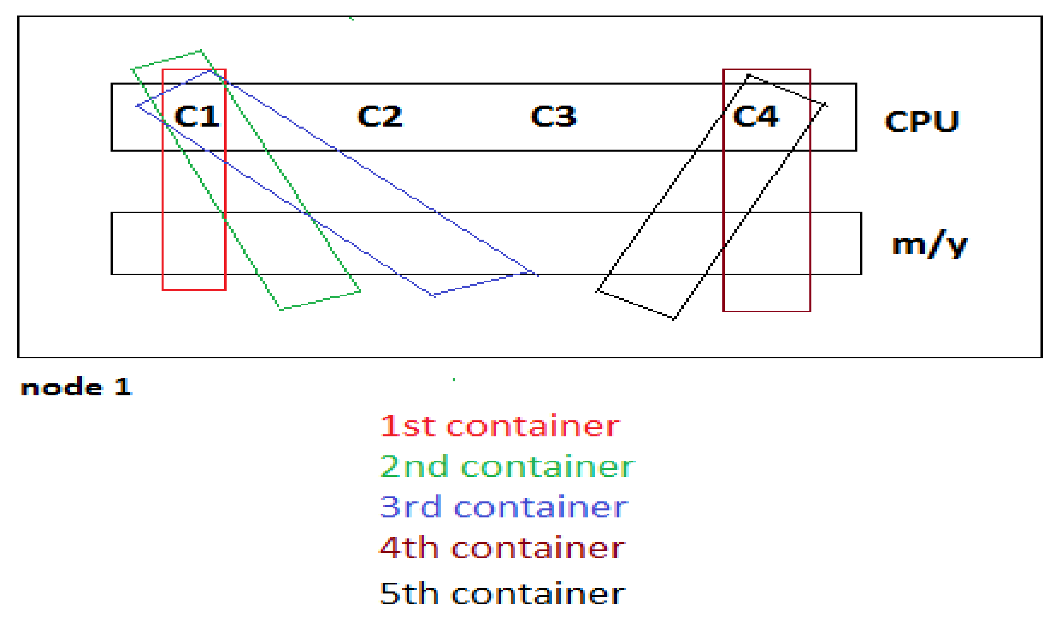
Container is a basic unit of resource allocation in YARN, which is also called slot in MRv1.

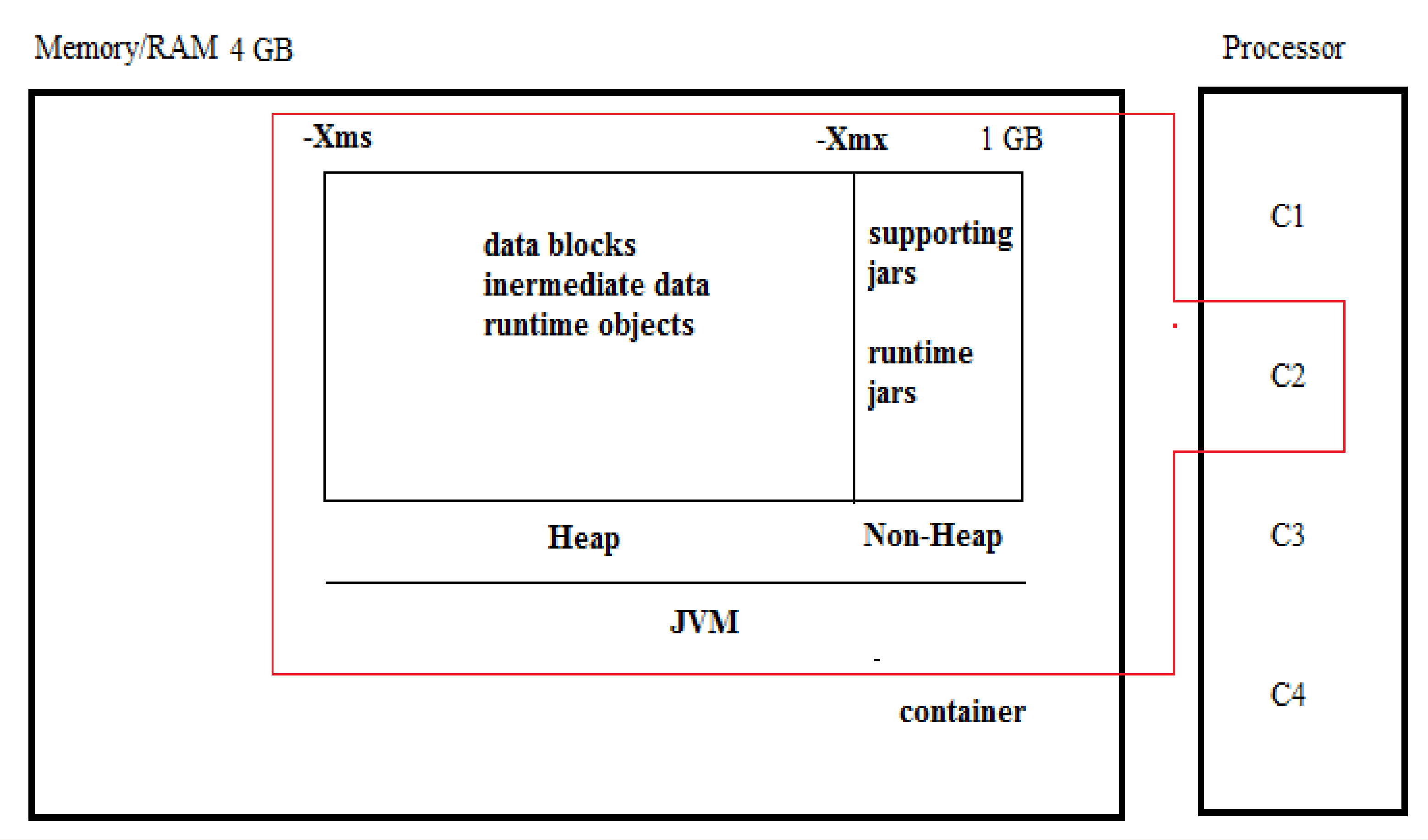
Container is composed of virtual CPU and portion of memory for map/reduce tasks, and MRAppMaster to execute.

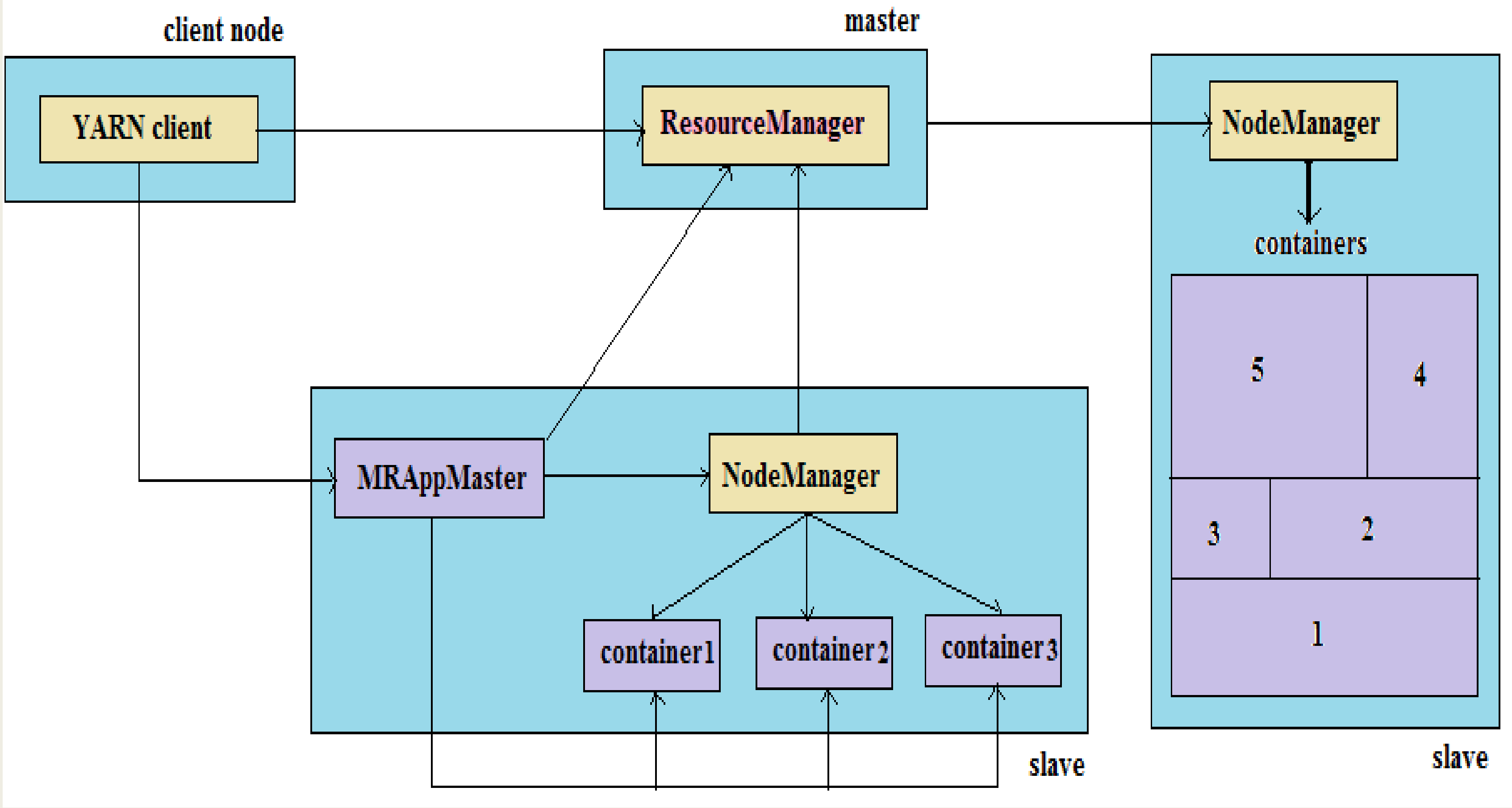
In MRv1, user can set number of map/reduce slots and its configuration in every TT.

But, the disadvantage of MRv1 slot based scheme is, to change the number of slots and its configuration, MR service should be stopped to re-configure and restarted.

Therefore, hard coding number map/reduce task slots and its configurations lead to cluster under utilization.




Memory/RAM 4 GB
Processor
-Xms
-Xmx
1 GB
data blocks
inermediate data
runtime objects
supporting jars
runtime jars
Heap
Non-Heap
JVM
container
C1
C2
C3
C4


client node
YARN client
master
ResourceManager
NodeManager
containers
5
4
3
2
1
slave
MRAppMaster
NodeManager
container1
container2
container3
slave

MR Application Master (MRAppMaster)

MRAppMaster is managed by AsM. It is created when MR job begins and destroyed when MR job ends.

MRAppMaster manages MR job life cycle management (from job submission till it end) unlike MRv1, where JT does job management.

For every MR job, a separate MRAppMaster is assigned to take care of them individually.

It requests RM for containers to launch map/reduce task via heartbeat message.

Job History Server (JHS)

It is a daemon that serves historical information about completed MR jobs.

It archives jobs’ metrics and meta-data.

Every NM records task activities locally.

Whenever a MR job is submitted, JHS will give you a MRAppMaster URL (in the console at the starting time of job) to track the running application.

Because, client does not know in which node MRAppMaster is running.

YARN job scheduler

YARN scheduler schedules and allocates resources for not only MRv2 jobs and also all other YARN application jobs.

YARN Scheduler is also pluggable as in MRv1. YARN scheduler doesn’t monitor task progress and provide fault tolerance for tasks, because it is not application specific.

So, MRAppMaster is designed to track task progress and provide fault tolerance for MR tasks. Therefore, YARN scheduler performs only job/application scheduling activity.

MRv1 and YARN scheduler come with the choice of FIFO, capacity, and fair schedulers.

Default scheduler of MRv1 is FIFO and for YARN is capacity scheduler.

In MRv1, job and task scheduling are done by MR scheduler itself.

In MRv2, job scheduling is done by YARN scheduler and task scheduling is done by MRAppMaster.

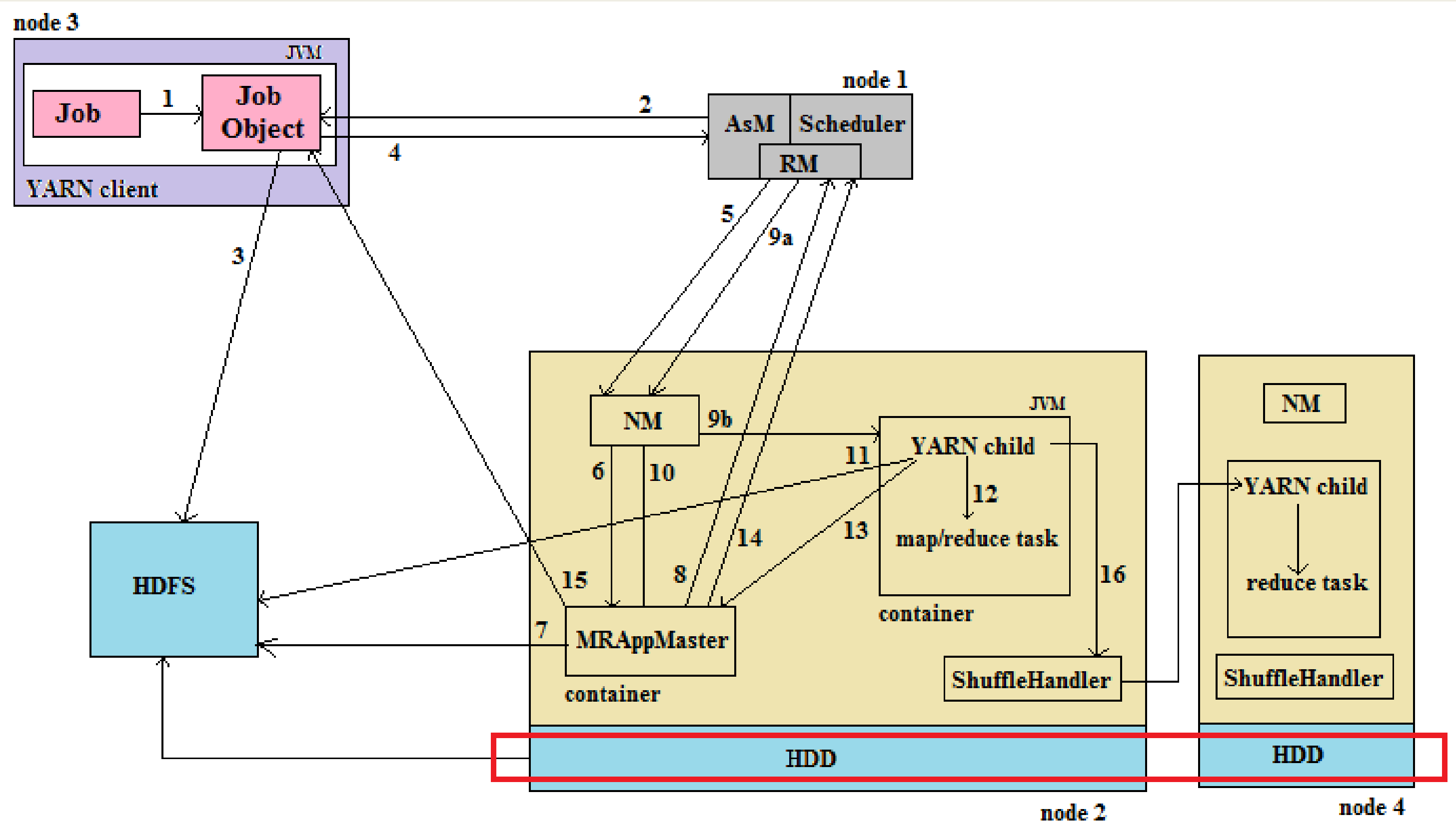
MRv2 job execution flow

Each MR job on YARN is executed as separate YARN application. MRv2 job execution flow is very similar to MRv1.

Only the responsibilities are split into different components to achieve scalability and support multiple framework.

Following figure illustrates the execution flow of MRv2 jobs in YARN.

Major steps are: Job submission, Job Initialization, Task Assignment, Task Execution, Status Updates and Job Completion.



Job Submission

1. Client submits MR job by interacting with Job object (JobSubmitter).

2. Job object interacts with AsM and acquires application meta-data such as application ID. Job object verifies whether input files and output directory already exists by interacting NN. If output directory exists or input files not available, then it throws an exception.

3. Job object calculates input splits by interacting NN and moves all the job-related resources such as job jars, job configurations, and input split information to HDFS to make them available for all NMs. These job-related information is replicated to DNs across cluster.

4. Job object submits the application to AsM. Output directory is created right after job submission is successful, but output files are written only at the end of job completion.

Job Initialization

5. Right after job submission, AsM interacts with RM to launch container for MRAppMaster. RM directs scheduler to create a container for MRAppMaster in a NM. MRAppMaster registers itself with the RM on its boot-up.

6. Calculating input splits in step 3 is slower. So, you can set MRAppMaster to calculate input splits. MRAppMaster also decides how to run map/reduce tasks of a job. If the job is small, MRAppMaster may choose to run tasks in the same JVM itself (such job is called uber job).

7. MRAppMaster grabs required job related resources from HDFS, such as input splits (if already calculated by job client), job configurations…

8. MRAppMaster retrieves block locations of input file from NN for input splits. MRAppMaster assigns task ID for both map and reduce tasks. Then, it sends resource request to RM to launch containers for map/reduce tasks. The resource-request carries the following information:

- list of three (replication factor by default 3) DN locations for each block. Ex: block1 resides in DN2, DN1, and DN3. DN2 comes first in the list, as MRAppMaster has determined that block1 in DN2 is close to MRAppMaster based on rackawareness.
- Priority
- Task ID
- CPU and memory requirements…

Task Assignment

- Task assignment is only applicable to non-uber jobs. RM gives control to scheduler to prepare execution plan based on resource request and launch container for map/reduce tasks.
- Scheduler attempts to launch container in NM2 to achieve data local execution. If there is not enough resource available to launch container in NM2, then scheduler tries NM1 to achieve data local execution….
- If no data locality is possible, then scheduler goes for non-local execution by moving block1 to the NM which has enough resources to launch container.
- Reduce tasks don’t have locality constraints to satisfy. Reduce tasks are scheduled to NM’s, so as to reduce network bandwidth. By default, map/reduce tasks are assigned with 1024 MB memory and 1 virtual CPU core.

Task Execution

9. RM instructus specific NM to start container. For each task, NM start container (a java process with YarnChild) with specified containerID. Each YarnChild is executed on dedicated JVM.

10. Then, NM gives container control to MRAppMaster.

11. YarnChild acquires and localizes job resources (configurations, jars…) by creating a local working directory. Data blocks are brough by DN in the respective slave node.

12. YarnChild creates an instance of TaskRunner to run the task. TaskRunner launches map/reduce task. A particular task will be attempted at least once, possibly more times if it crashes. If same record causes task to crash over and again, then particular record is abandoned. Multiple attempts for a task may occur in parallel with speculative execution turned on.

16: Shuffle phase in MR is responsible for sorting mapper outputs and distributing them to the reducers. Every NM runs shuffle service that takes map output to the reducer task.

Status Updates

13. YarnChild provides necessary information (task progress, counters...) to its MRAppMaster via an application-specific protocol (every 3 seconds). MRAppMaster accumulates and aggregates logs and counters from YarnChild to assemble current status of the job to update clients.

14. Once the application is complete, MRAppMaster deregisters with the RM and shuts down.

15. Client (Job object) periodically polls MRAppMaster for job status updates. MRAppMaster receives task status information from each container and assembles to update job status.

Job Completion

- Job is marked as success or failed upon its completion. RM WUI displays details of all the running jobs and its tasks. Every job is given URI of MRAppMaster to track job status.
- Because, job object doesn’t know in which node MRAppMaster of every job is running. However, once job is done successfully, job history information from MRAppMaster is moved to Job History Server and job object displays job counters. After MR application/job completes (or fails).
- Containers for tasks and MRAppMaster are removed. Intermediate working directory handling intermediate data also is removed from memory. Reclaimed resource is not immediately marked as free space. It can take several 10 seconds to be available for next call.
- Log and metrics information are sent from MRAppMaster to the MapReduce History Server.

MR weaknesses and solutions

1. Lack of selective access to input data

For every MR job, entire input data is processed. It is not possible to access only subset of data. There are tools to achieve this based on Hadoop:

- Indexing data: Hadoop++, HAIL
- Intentional data placement: CoHadoop
- Data layout: Llama, Cheetah, RCFile, CI

2. Redundant processing and re-computation

Different jobs perform similar processing on same input data. There is no way to reuse the results produced by previous jobs. When a future job requires those results, we have to re-compute the whole data. Some possible tools to overcome these problems are:

- MRShare
- Result sharing and materialization: ReStore
- Incremental processing: Incoop

3. Lack of early termination

All map tasks must be completed before any reduce task can start processing. To finish job before all map tasks completed, in case if we achived desire results, then folloing tools may be useful:

- Sampling: EARL
- Sorting: RanKloud

4. Lack of iteration

MR programmers need to write a sequence of MR jobs and coordinate their execution in order to implement machine learning algorithms. The intermediate result of each iteration is stored into HDFS and then reloaded for further processing. It involves huge IO bound processing. Solutions are:

- *Looping,* caching, pipelining: Stratosphere, Haloop, MapReduce on-line, NOVA, Twister, CBP, Pregel, PrIter
- Incremental processing: Stratosphere, REX, Differential dataflow

5. Lack of interactive and stream processing

MR leads to various overheads due to guarantee fault-tolerance that negatively impact the performance of job. Many applications require fast response time for interactive analysis, and stream analytics. Possible tools are:

- Streaming, pipelining: Dremel, Impala, Hyracks, Tenzing
- In-memory processing: PowerDrill, Spark/Shark, M3R
- Pre-computation*:* BlikDB

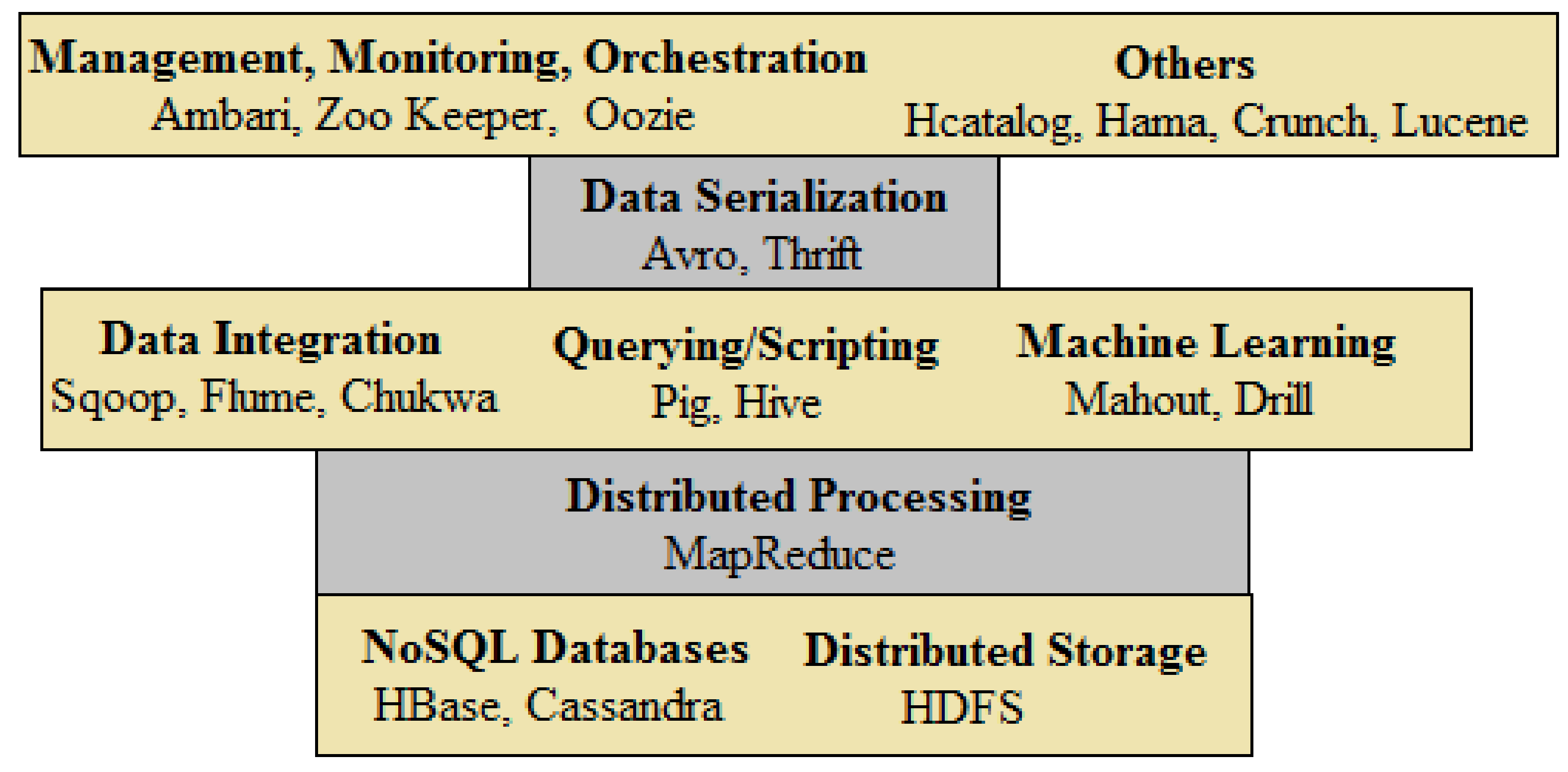
6. joins are slow in MR.

7. there is no specific optimal configuration for job efficiency.

Hadoop sub projects

Hadoop is mainly known for two core tools: HDFS and MR. Today, in addition to HDFS and MR, the term also represents several sub-projects as shown in Figure.

Head to incubator.apache.org and incubator.apache.org/projects to know more.



Management, Monitoring, Orchestration
Ambari, Zoo Keeper, Oozie

Others
Hcatalog, Hama, Crunch, Lucene

Data Serialization
Avro, Thrift

Data Integration
Sqoop, Flume, Chukwa

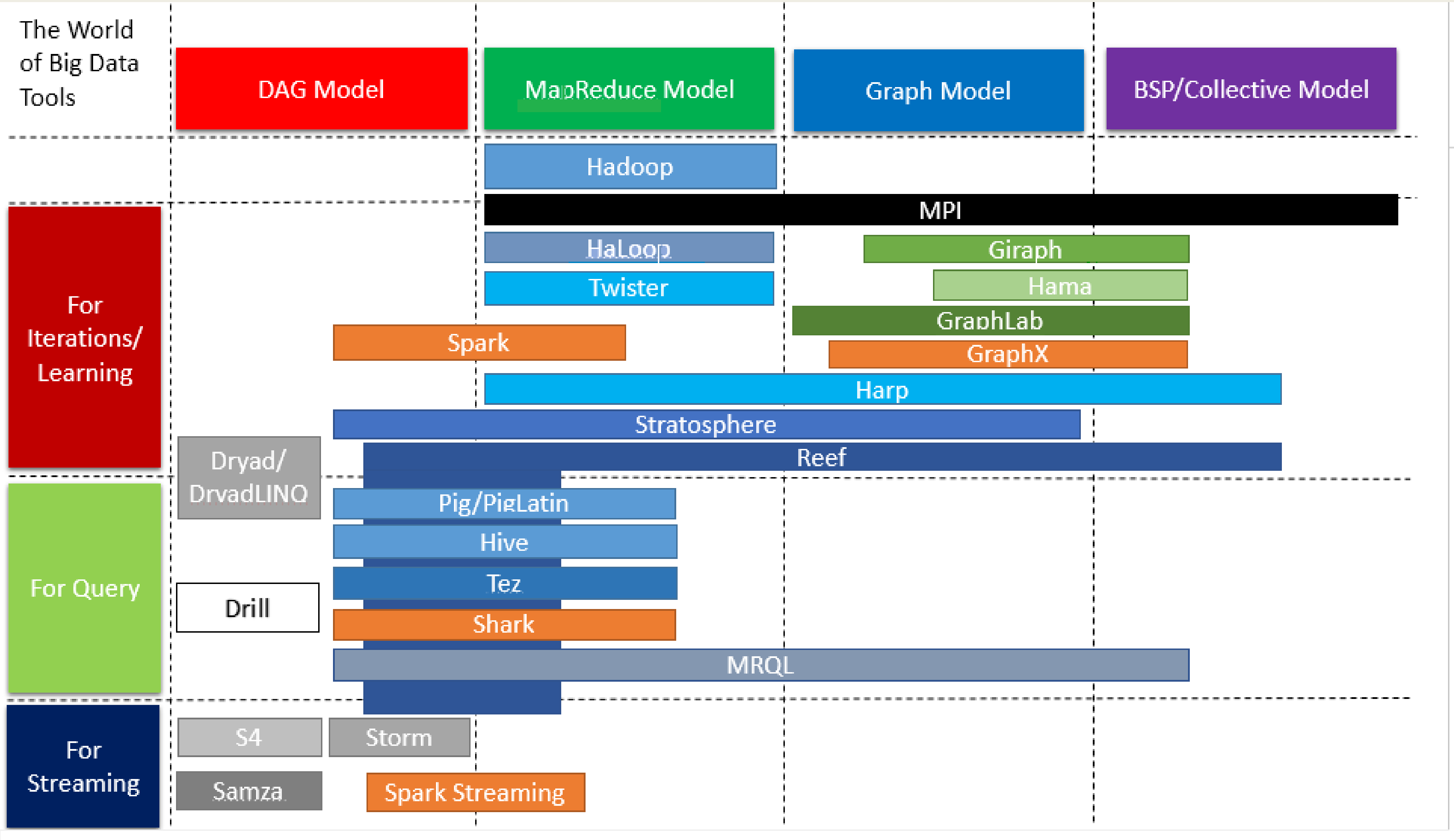
Querying/Scripting
Pig, Hive

Machine Learning
Mahout, Drill

Distributed Processing
MapReduce

NoSQL Databases
HBase, Cassandra

Distributed Storage
HDFS



The World of Big Data Tools

DAG Model

MapReduce Model

Graph Model

BSP/Collective Model

Hadoop

MPI

For Iterations/ Learning

HaLoop

Twister

Giraph

Hama

GraphLab

Spark

GraphX

Harp

Stratosphere

Dryad/ DryadLINQ

Reef

For Query

Pig/PigLatin

Hive

Drill

Tez

Shark

MRQL

For Streaming

S4

Storm

Samza

Spark Streaming

Querying and scripting

Pig

It is a home for non-java programmers to process big data. It was developed by Yahoo.

Yahoo writes 50% of jobs in pig. 100 lines of MR program in java can be written in just 10 lines using pig.

But, you don’t have much facility for writing user defined algorithms as there are limited number of pre-defined queries.

Pig stores and manages huge structured/semi-structured data on HDFS. It uses Pig Latin language. Pig run time compiles, converts pig queries to lower level MR jobs at run time.

Hive

It is mainly for SQL users who are non-java programmers.

It provides DWH facility on top of HDFS. Relational DWH is highly centralized. But, Hive is distributed DWH on top of HDFS.

It uses Hive Query Language (HQL), which is very similar to SQL to work with structured data.

You have to define schema and then load huge data onto it.

Hive runtime converts HQL to lower level MR jobs. Both pig and hive are not client-server architecture and perform ETL process before applying any algorithms.

Data storage (NoSQL)

NoSQL database systems were developed to provide operational functionality on real-time with interactive response on big data unlike Hadoop MR.

Hbase

It is column oriented database based on Google’s Big Table (huge structured database built on top of GFS).

Unlike pig and hive, it provides interactive response on HDFS.

It supports table with millions of columns and billions of rows on top of HDFS. Data can be indexed and perform analytics on them.

It is natively integrated with Pig and Hive. Only HBase in all NoSQL database systems is closely integrated with MR and HDFS.

It provides online read/write access to individual rows and batch operation for bulk data read/write access. It works on master-slave architecture.

It is horizontally auto-scalable, consistent, and supports auto failover. MR jobs can be run on HBase.

Data integration tool

HDFS commands are used to load data from local file system onto HDFS. There are tools to load data from other sources such as RDBMS, Streams…

Sqoop (**SQ**L+ Had**oop)**

Integrates Hadoop with relational database system.

You can’t perform huge data processing in relational database system.

Therefore, move huge structured data onto HDFS and perform analysis.

Sqoop also converts its queries to MR job and executes behind the screen.

Sqoop performs EL in ETL.

Flume

Log processing is one of the suitable applications of MR.

Logs are generated monotonically across many machines and moved dynamically onto HDFS on real-time using tools like flume.

Kafka

Apache Kafka is a fast, scalable, durable, and fault-tolerant publish-subscribe messaging system.

Kafka is often used in place of traditional message brokers like JMS and AMQP because of its higher throughput, reliability and replication.

Kafka works in combination with Apache Storm, Apache HBase and Apache Spark for real-time analysis and rendering of streaming data.

Kafka can message geospatial data from a fleet of long-haul trucks or sensor data from heating and cooling equipment in office buildings.

Whatever the industry or use case, Kafka brokers massive message streams for low-latency analysis in Enterprise Apache Hadoop.

what Kafka does

- Stream Processing
- Website Activity Tracking
- Metrics Collection and Monitoring
- Log Aggregation

Management, Monitoring, Orchestration

Zookeeper

- It performs distributed service co-ordination and synchronization.
- It is used in HBase, NN high availability….
- It is very similar to Chubby in google.

Oozie

- Manual effort is required to run and co-ordinate sequence of MR jobs.
- Oozie is a workflow scheduling library to manage Hadoop jobs in DAG.
- It helps in scheduling jobs one after other in a sequence.
- Jobs are triggered by time, data availability...
- Falcon is also a workflow scheduling library.

Talend

It is a software integration tool.

Because supporting increasing data volumes, users, and use-cases requires you to constantly evolve your data infrastructure, from Java and data warehouses to the Cloud and Spark and The Next Big Thing.

Only Talend allows you to effortlessly adopt new technologies so you can focus more on creating business value and less on data integration.

Thank you