Map Reduce

NSIT ⟶ marks ⇒ percentage

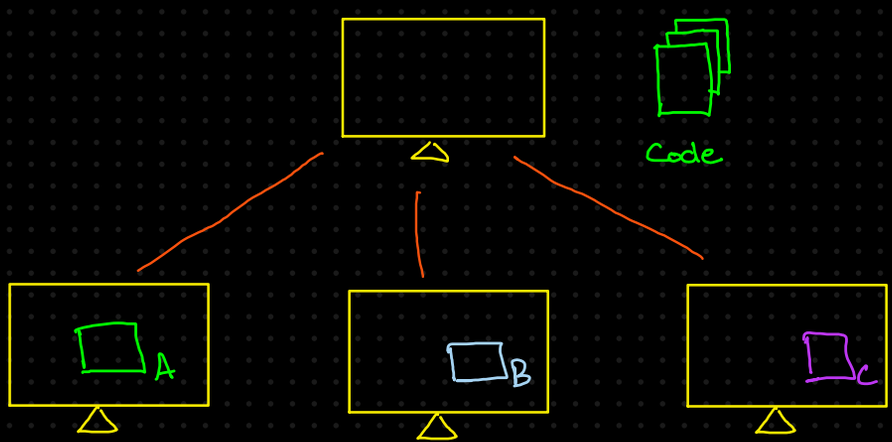
Maths	4	75
ED	4	19
Viva	2	60

Code to calculate %

data ⟶ process ⟶ percentage

google chrome
Voting

Delhi



data → code

★ data locality + parallel processing

≃ Divide & Conquer Paradigm

Map Reduce

1. it is difficult to write
2. Theory is a lot imp → distributed processing

Practical is not that imp.

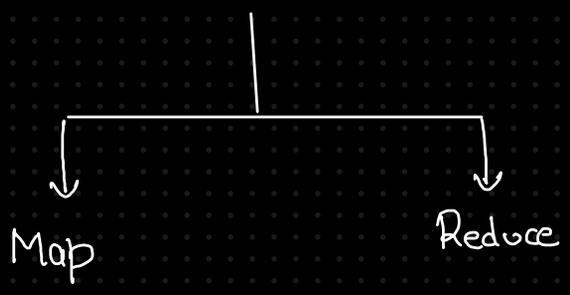
GMR → MR

Map Reduce is a programming model designed for processing large dataset in a distributed computing manner

Smaller task

- → Scalability
- → fault tolerance
- → parallelism

1. We will have data distributed parallely via hdfs which is required



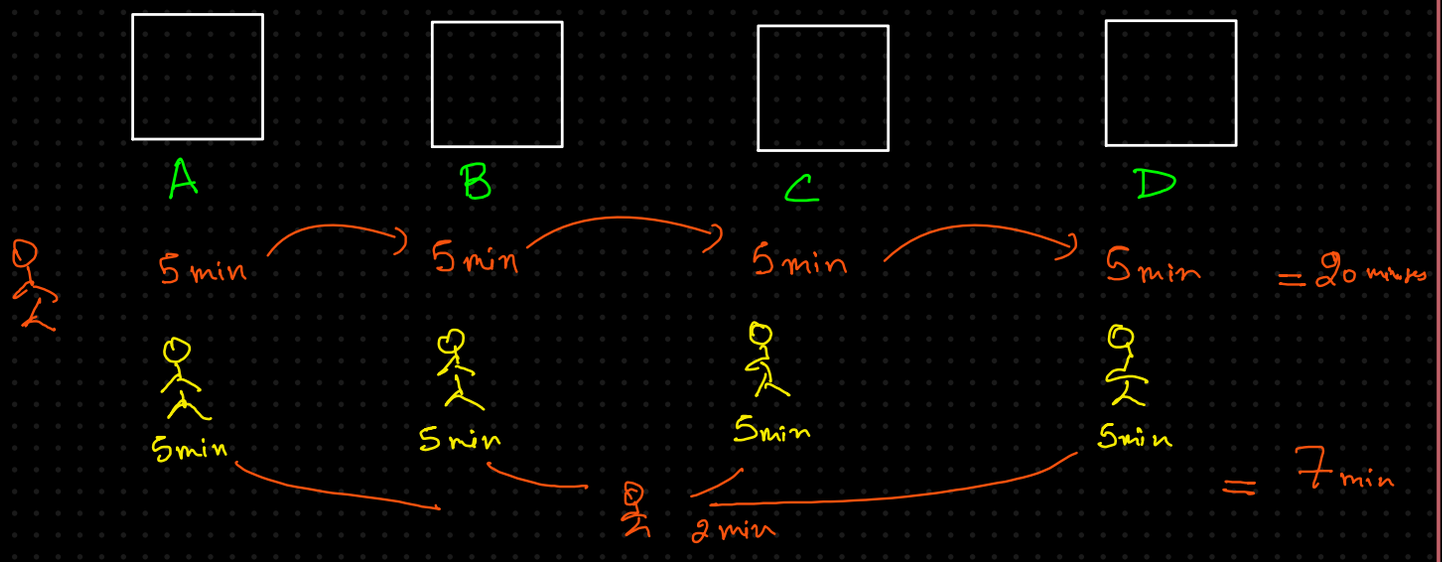
input data is split into chunks, & each chunk processed by a mapper

↓

Mapper produces intermediate output

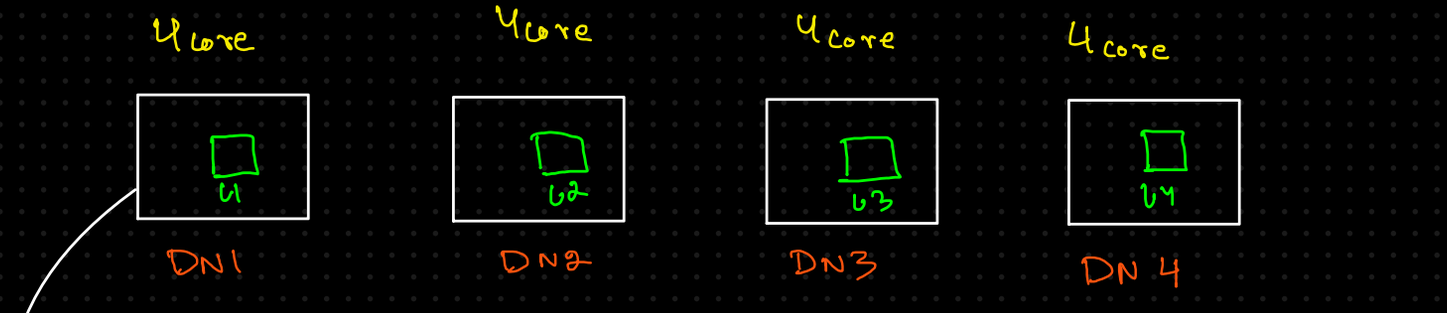
↓

Reducer combines it



Each and every Input or Output of MR job is a Key-value pair.

Say we have a 400 mb file in a hadoop cluster



1 cpu core can process 1 mapper job on 1 block / input split of data

4 cores

it can parallely process 4 blocks/IS of data

total 16 cores

4gb file → (128mb block size) → 32 blocks

So in this case as blocks >> cores, we will process the job in batches.

Word Count example

1. Input → text with sentences

2. Mapper → Value ('hello' world)

↓

('hello', 1) ('world', 1)

3. Combiner (optional)

4. Shuffle and sort

5. Reducer

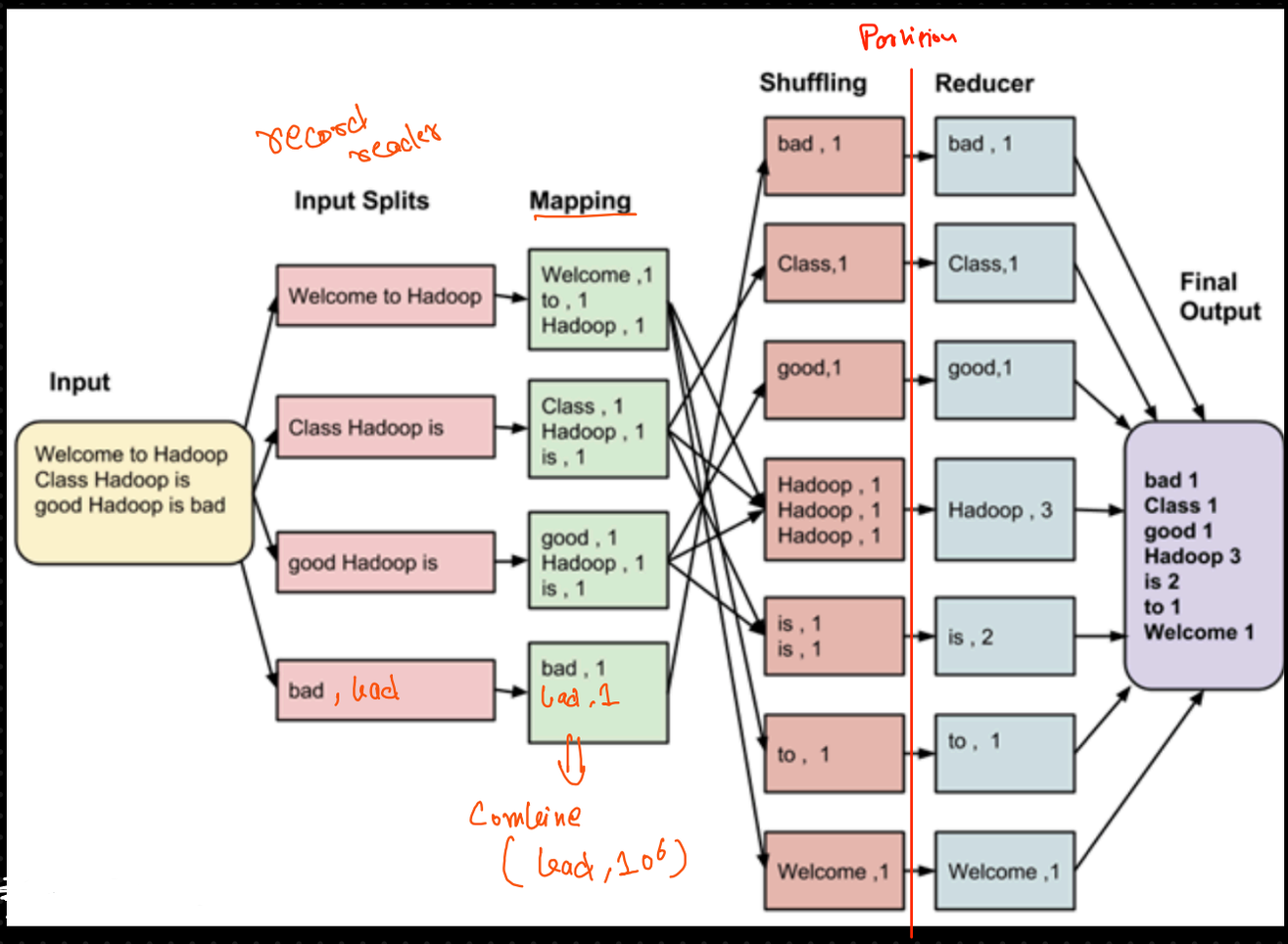
Aggregate the values

Goku, {1, 1, 1, 1}3

↓

(Goku, {4 3})

6. Final Output



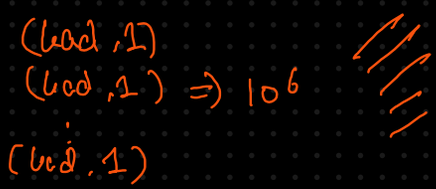
Gohu vegeta Gohan → (Gohu,1) (vegeta,1) (Gohan,1)

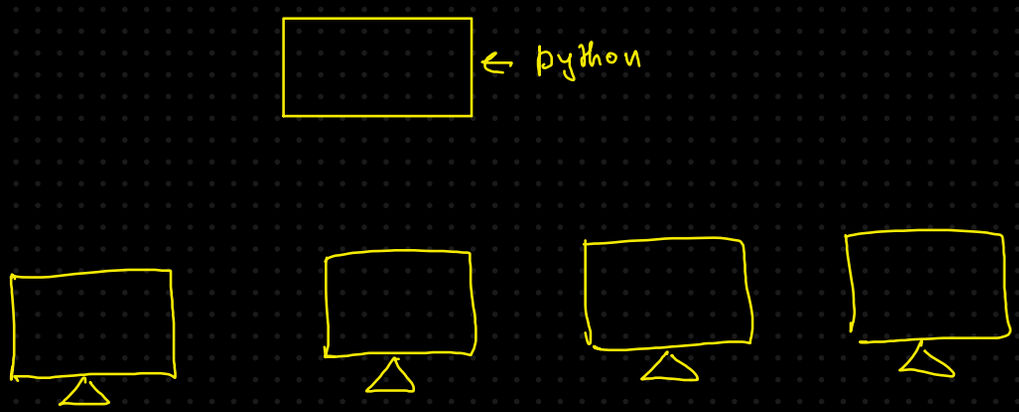
Gohan Frieza Gohu Gohu → (Gohan,1) (Frieza,1) (Gohu,1)
(Gohu,1)

↓↓ Combiner

(Gohu,2)

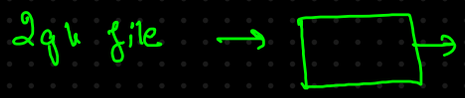
Step	Process	Output Example
Input Splitting	Splitting input file into HDFS chunks.	Split 1: `Goku Vegeta Gohan`, Split 2: `Goku Frieza Goku`, etc.
Mapper Phase	Generating key-value pairs (`word`, `1`) from each split.	`("Goku", 1), ("Vegeta", 1), ("Gohan", 1)`
Combiner Phase	Optional local aggregation at Mapper to reduce shuffle data.	`("Goku", 2), ("Frieza", 1)`
Shuffle and Sort	Grouping and sorting intermediate key-value pairs by key.	`("Goku", [1, 2, 1, 2]), ("Frieza", [1, 1, 1])`
Reducer Phase	Aggregating grouped data to produce final results.	`("Goku", 6), ("Frieza", 3)`
Final Output	Writing final aggregated results to HDFS.	`Frieza 3`, `Gohan 3`, `Goku 6`, `Vegeta 2`

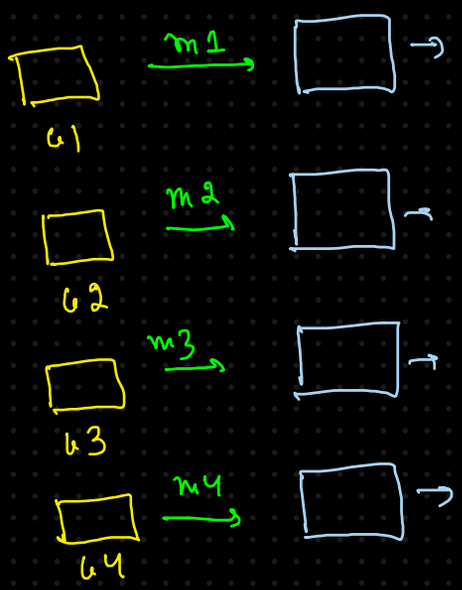




MR doesn't make sense for smaller data.

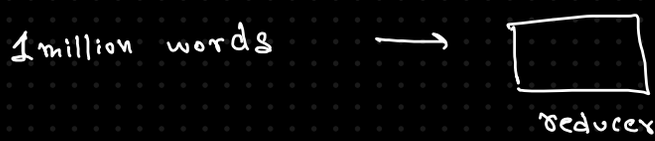
No. of Mappers ? = number of blocks

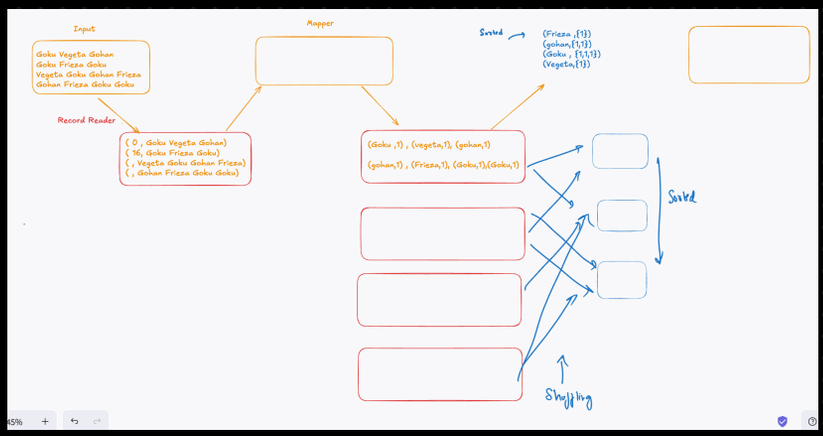




We can control the number of reducer

It can be 0, 1, ... n




Input
Goku Vegeta Gohan
Goku Frieza Goku
Vegeta Goku Gohan Frieza
Gohan Frieza Goku Goku
Mapper
Sorted
(Frieza ,{1})
(gohan,{1,1})
(Goku , {1,1,1})
(Vegeta,{1})
Record Reader
(0 , Goku Vegeta Gohan)
(16, Goku Frieza Goku)
(, Vegeta Goku Gohan Frieza)
(, Gohan Frieza Goku Goku)
(Goku ,1) , (vegeta,1), (gohan,1)
(gohan,1) , (Frieza,1), (Goku,1),(Goku,1)
Sorted
Shuffling
45%

19th Jan

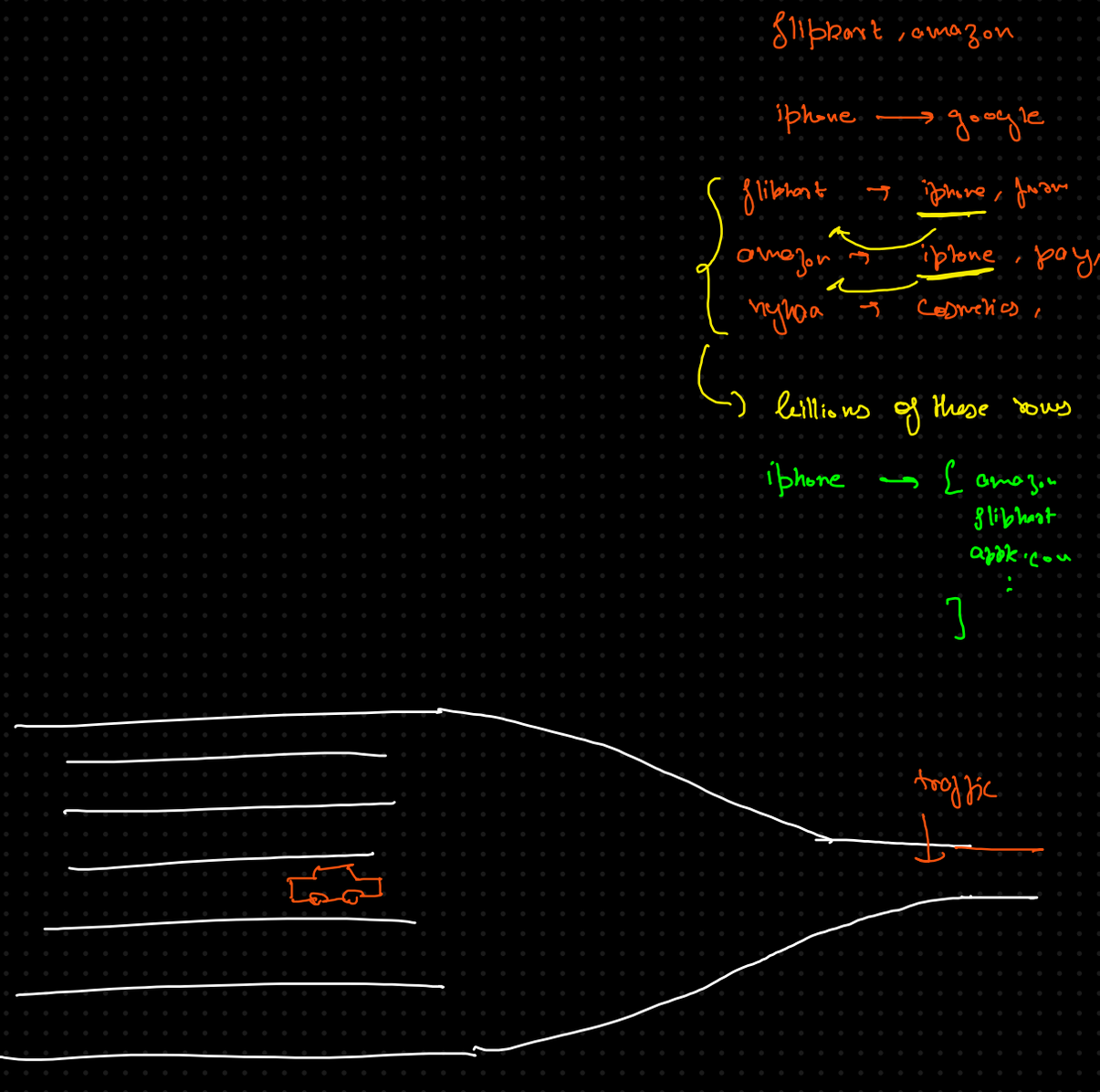
Map Reduce → 2 Reducers
0 reducers

Log file → 2 blocks
Input split

YARN

Spark ⟶ pyspark

2. Map Reduce with more than 1 reducer



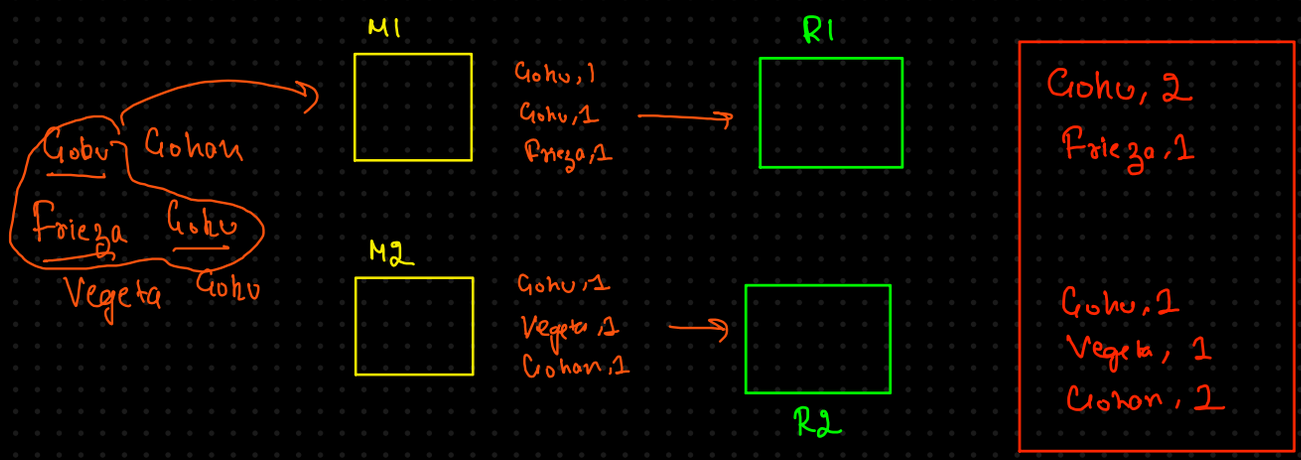
Always make sure that mapper handles the heavy lifting in terms of processing

Let us have more than one reducer

mapper jobs	→ 10 min		10 min
reducer	→ 30 min	↑↑ * 3	10 min
	40 min		20

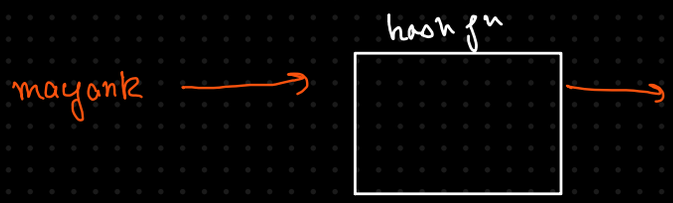
Partitioning

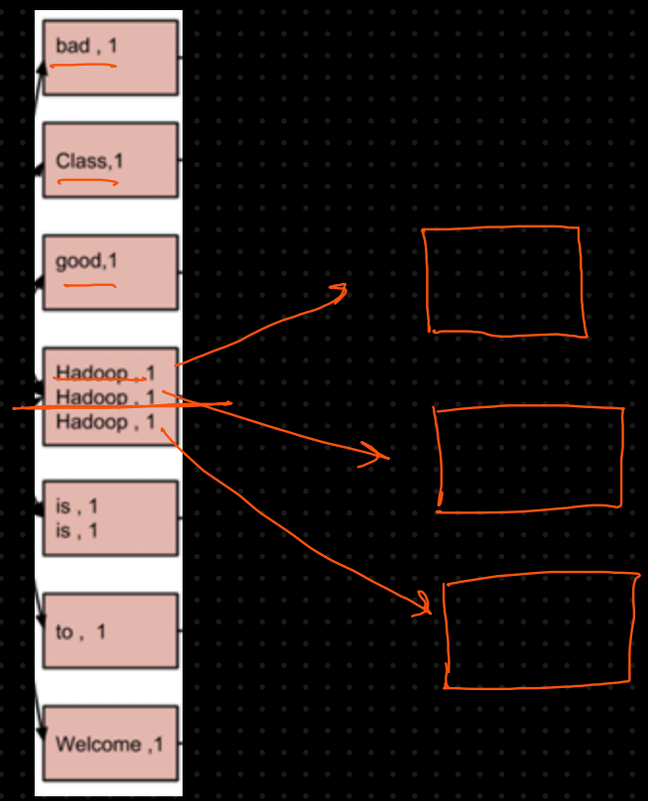
We can not have chain of reducers



Same keys should always go to same reducer

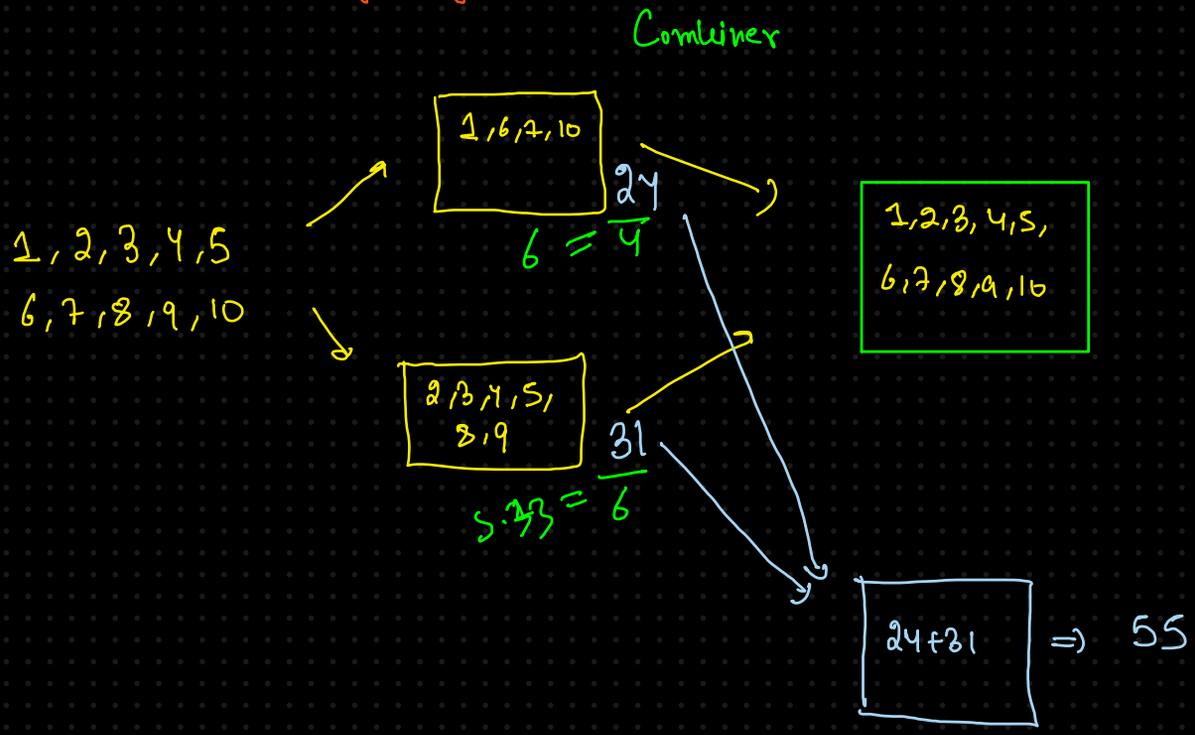
% modulus a very good hash function




bad , 1
Class,1
good,1
Hadoop , 1
Hadoop , 1
Hadoop , 1
is , 1
is , 1
to , 1
Welcome ,1

Combiners

We do local aggregation



Then, can we always use combiners?

Non associative or non commutative operation

eg: avg, median

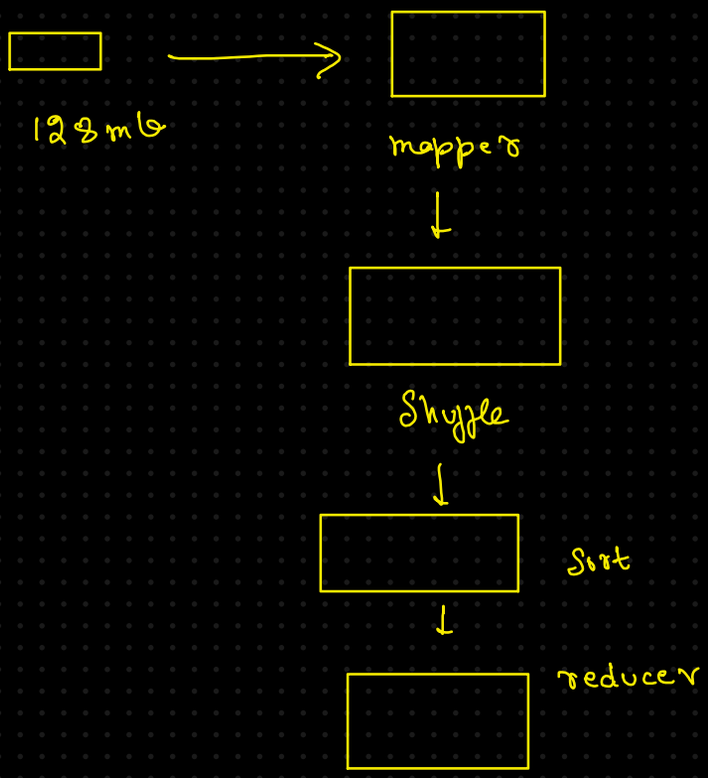
Map Reduce with zero Reducers

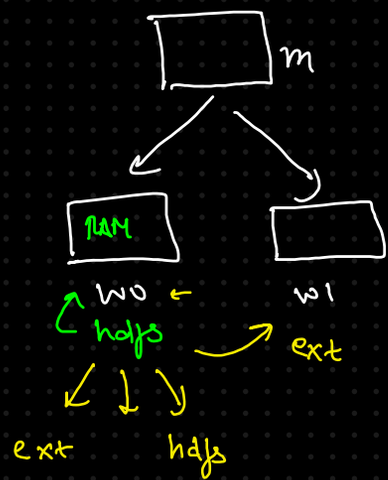
gohu 1

Examples → format conversion

Mapper O/P is final output

Map Reduce


128mb
mapper
Shyffle
Sort
reducer

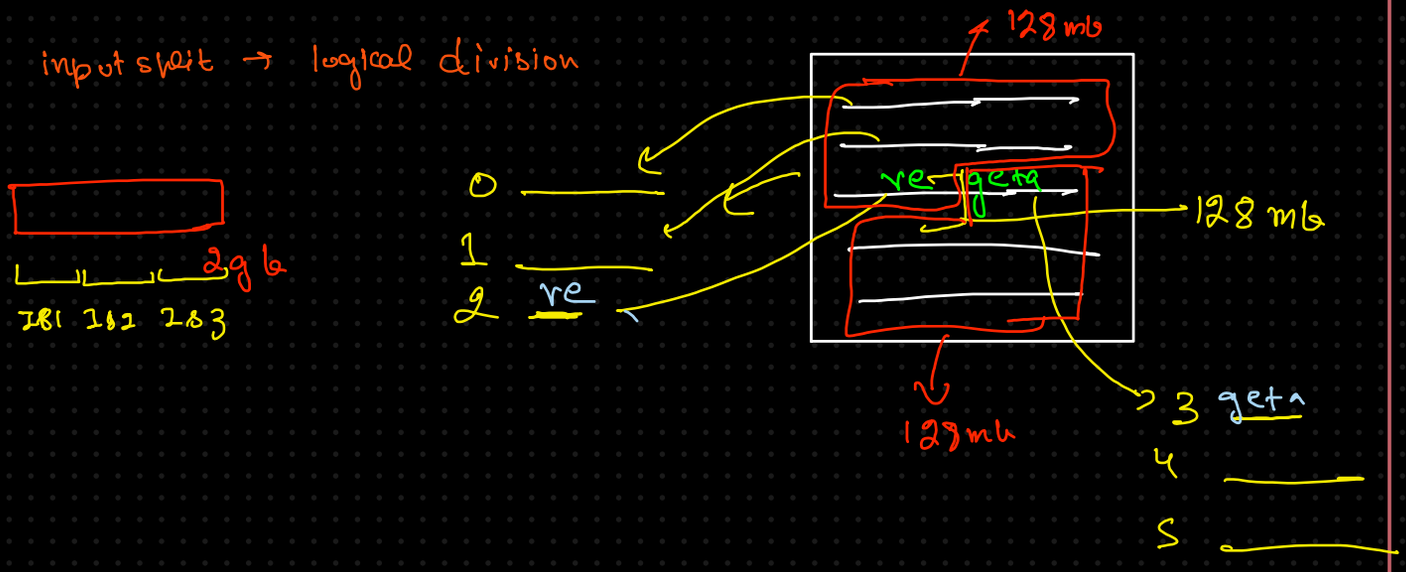

m
RAM
w0
w1
hdfs
ext
ext
hdfs

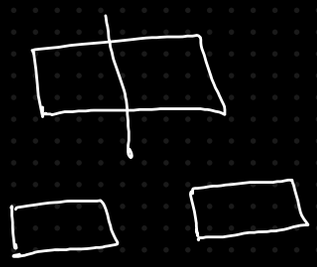
Input splits

Block/input split

blocks → physical division

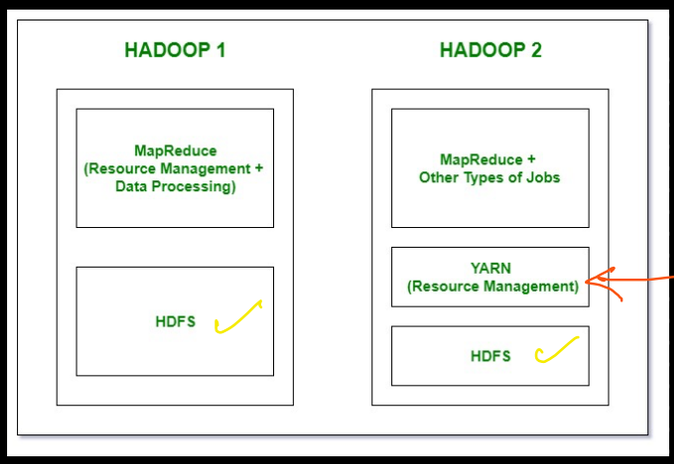
input split → logical division

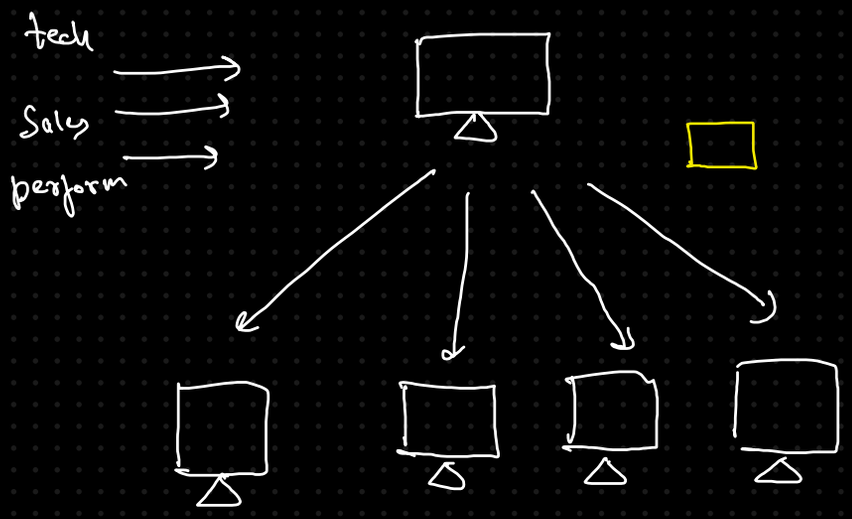




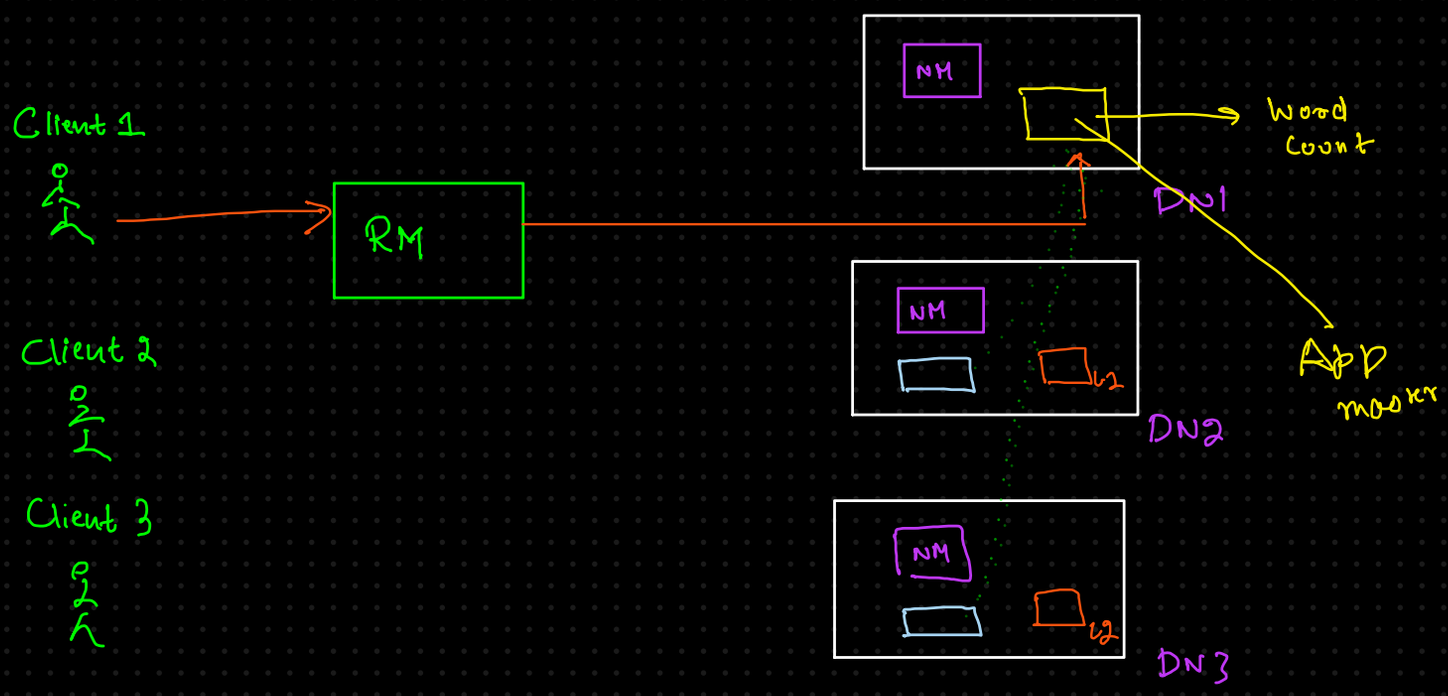
Record

YARN (Yet another resource Negotiator)





JP1 → Name node
JP2 → Datanode
JP3 → SNN
JP4 → RM
JP5 → NM

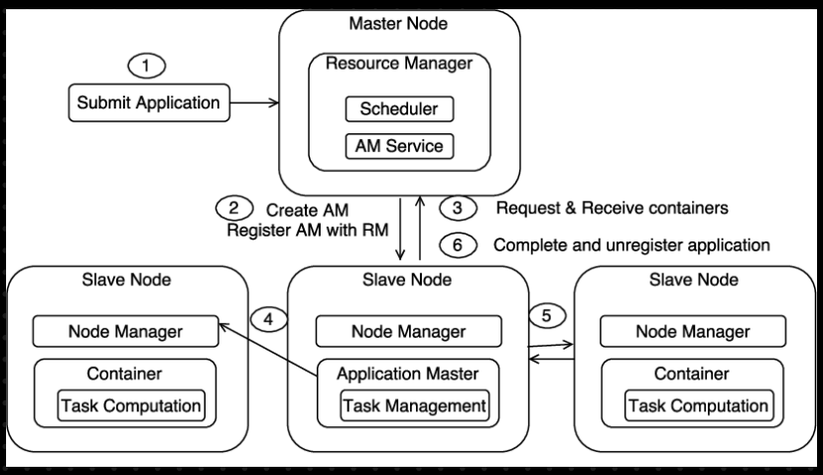


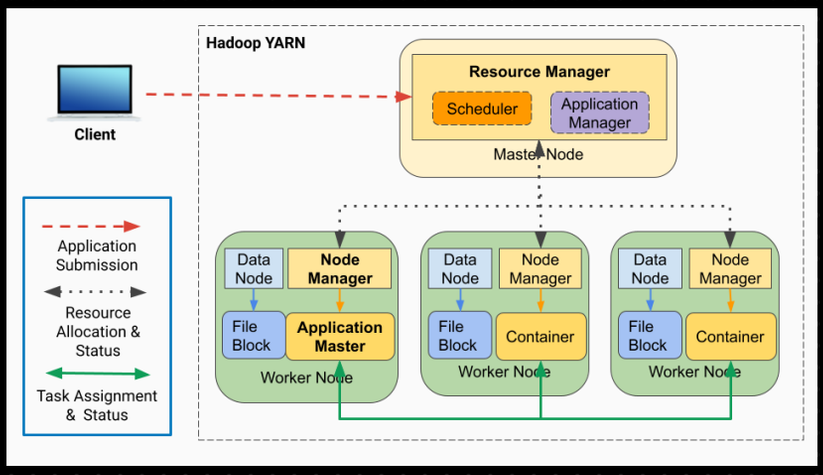
RM jobs

1. Scheduler
2. AM (application manager)

Step by step

1. Submission of Application → RM accepts and creates a App Manager ⇓ App master
2. request of resources
3. Allocation of resources → Scheduler
4. Launching of Container → RM communicate with NM
5. Task execution
   - 5a. Monitoring & Fault Tolerance
6. Completion → App master will free up the resources
   App master also killed & release its resources
7. Success →


Master Node
1
Submit Application
Resource Manager
Scheduler
AM Service
2
Create AM
Register AM with RM
3
Request & Receive containers
6
Complete and unregister application
Slave Node
Node Manager
Container
Task Computation
4
Slave Node
Node Manager
Application Master
Task Management
5
Slave Node
Node Manager
Container
Task Computation


Hadoop YARN
Client
Resource Manager
Scheduler
Application Manager
Master Node
Application Submission
Resource Allocation & Status
Task Assignment & Status
Data Node
Node Manager
File Block
Application Master
Worker Node
Data Node
Node Manager
File Block
Container
Worker Node
Data Node
Node Manager
File Block
Container
Worker Node