# MAP-REDUCE FRAMEWORK

-By Jagadish Rouniyar

# ACTIVITIES

- 10 students are given the task of counting words in a book “Big Data”

How they will perform it ? Identify the step . It denotes the map-reduce function

# MAP - REDUCE

- **sort/merge based distributed processing**
- **Best for batch- oriented processing**
- **Sort/merge is primitive**
  - Operates at **transfer rate** (Process+data clusters)
- **Simple programming metaphor:**
  - input | map | shuffle | reduce > output
  - cat * | grep | sort | uniq c > file
- **Pluggable user code runs in generic reusable framework**
  - log processing,
  - -- web search indexing
  - SQL like queries in PIG
- **Distribution & reliability**
  - Handled by framework - transparency

# MR MODEL

- **Map()**
  - Process a key/value pair to generate intermediate key/value pairs
- **Reduce()**
  - Merge all intermediate values associated with the same key
- **Users implement interface of two primary methods:**
  1. **Map: (key1, val1) → (key2, val2)**
  2. **Reduce: (key2, [val2]) → [val3]**
- ***Map - clause group-by (for Key) of an aggregate function* of SQL**
- **Reduce - *aggregate* function (e.g., average) that is computed over all the rows with the same group-by attribute (key).**

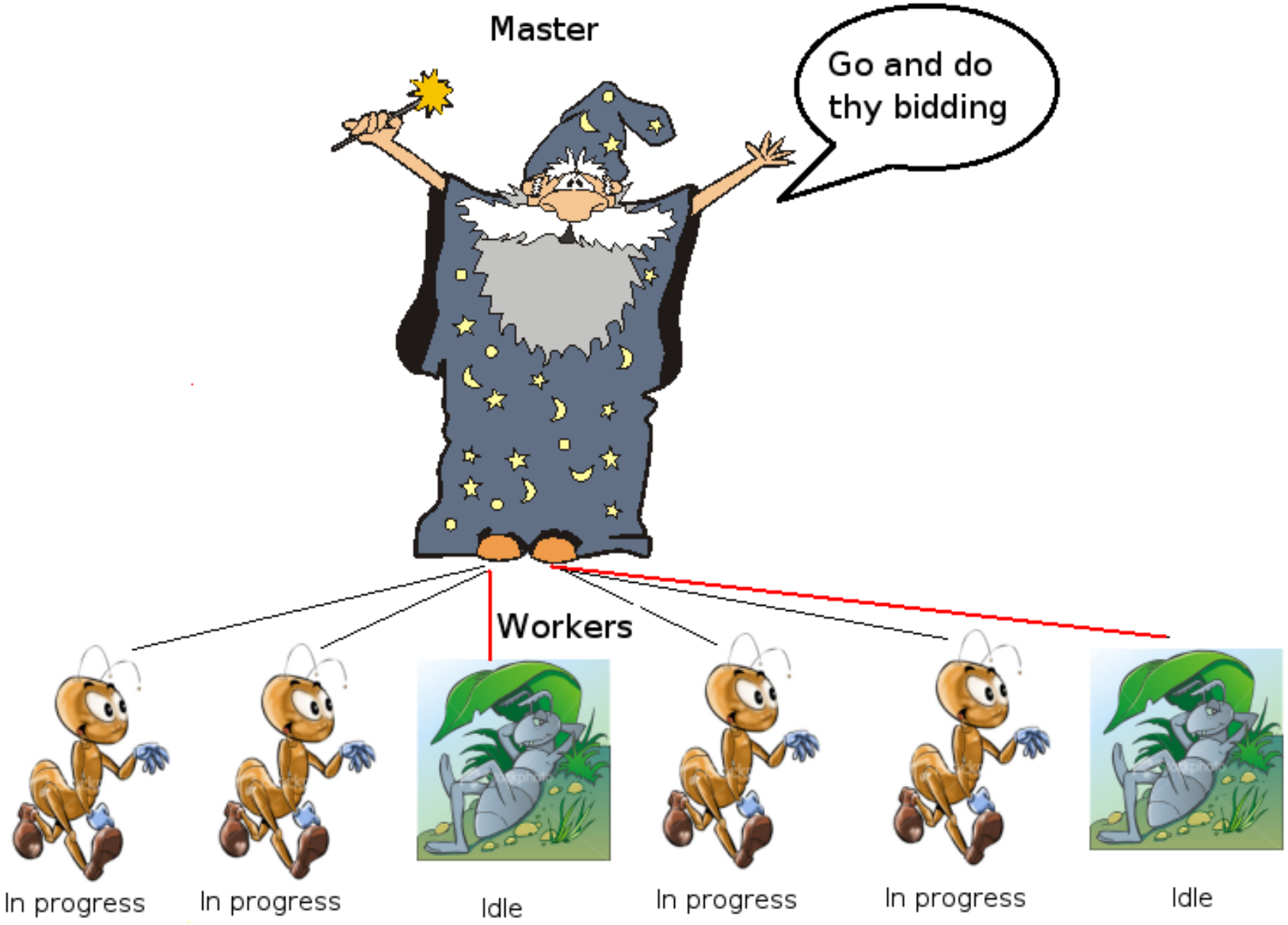
- **Application writer specifies**
  - A pair of functions called ***Map*** and ***Reduce*** and a **set of input files and submits the job**
- **Workflow**
  - *Input* phase generates a number of ***FileSplits*** from input files (one per Map task)
  - The *Map* **phase** executes a user function to transform input kev-pairs into a new set of kev-pairs
  - The framework sorts & ***Shuffles*** the kev-pairs to output nodes
  - The ***Reduce*** phase combines all kev-pairs with the same key into new kevpairs
  - The output phase writes the resulting pairs to files
- **All phases are distributed with many tasks doing the work**
  - Framework handles scheduling of tasks on cluster
  - Framework handles recovery when a node fails

## Data distribution

- Input files are split into M pieces on distributed file system - 128 MB blocks
- Intermediate files created from *map* tasks are written to local disk
- Output files are written to distributed file system

## Assigning tasks

- Many copies of user program are started
- Tries to utilize data **localization** by running *map* tasks on machines with data
- One instance becomes the Master
- Master finds idle machines and assigns them tasks

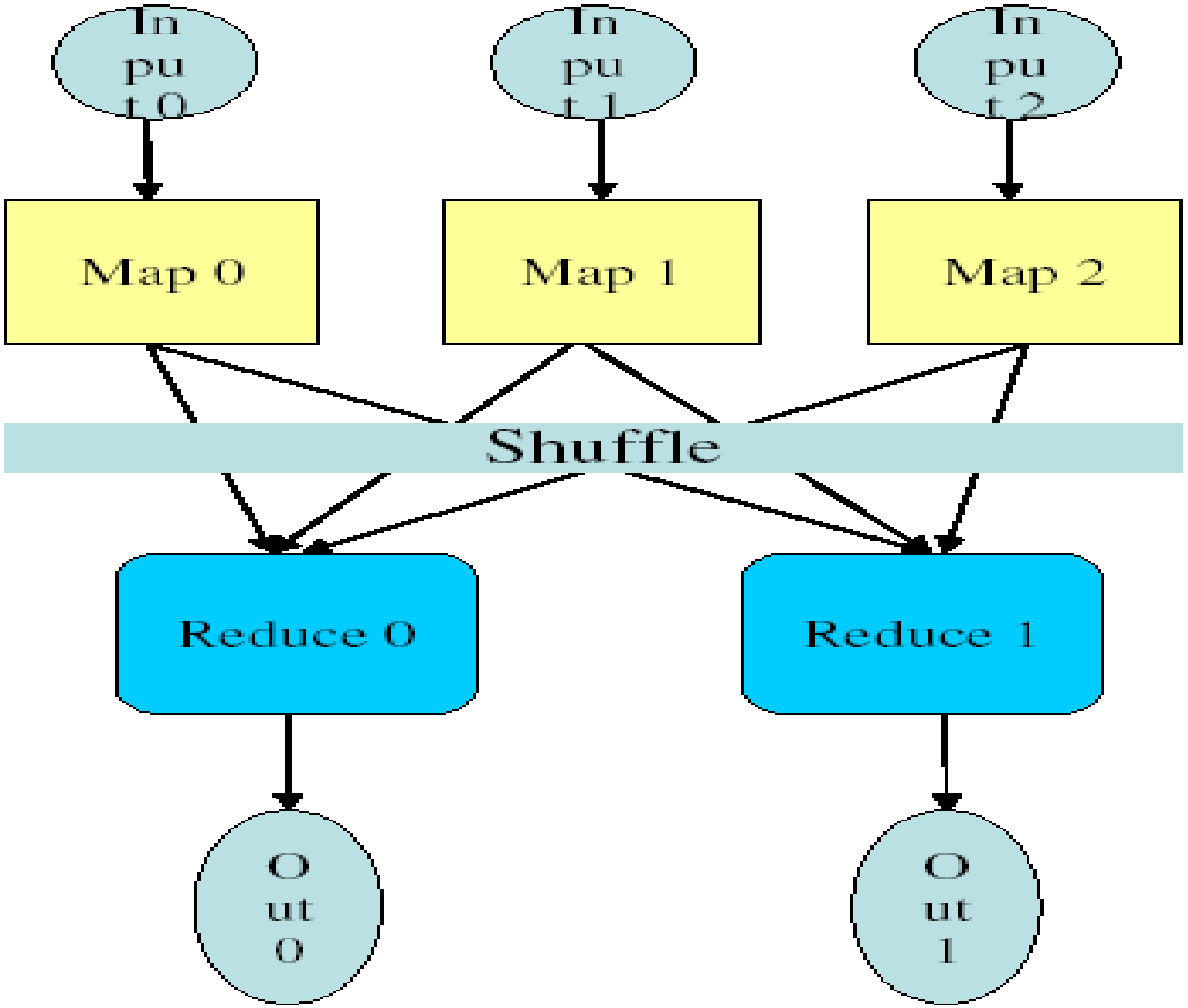

Master
Go and do thy bidding
Workers
In progress
In progress
Idle
In progress
In progress
Idle

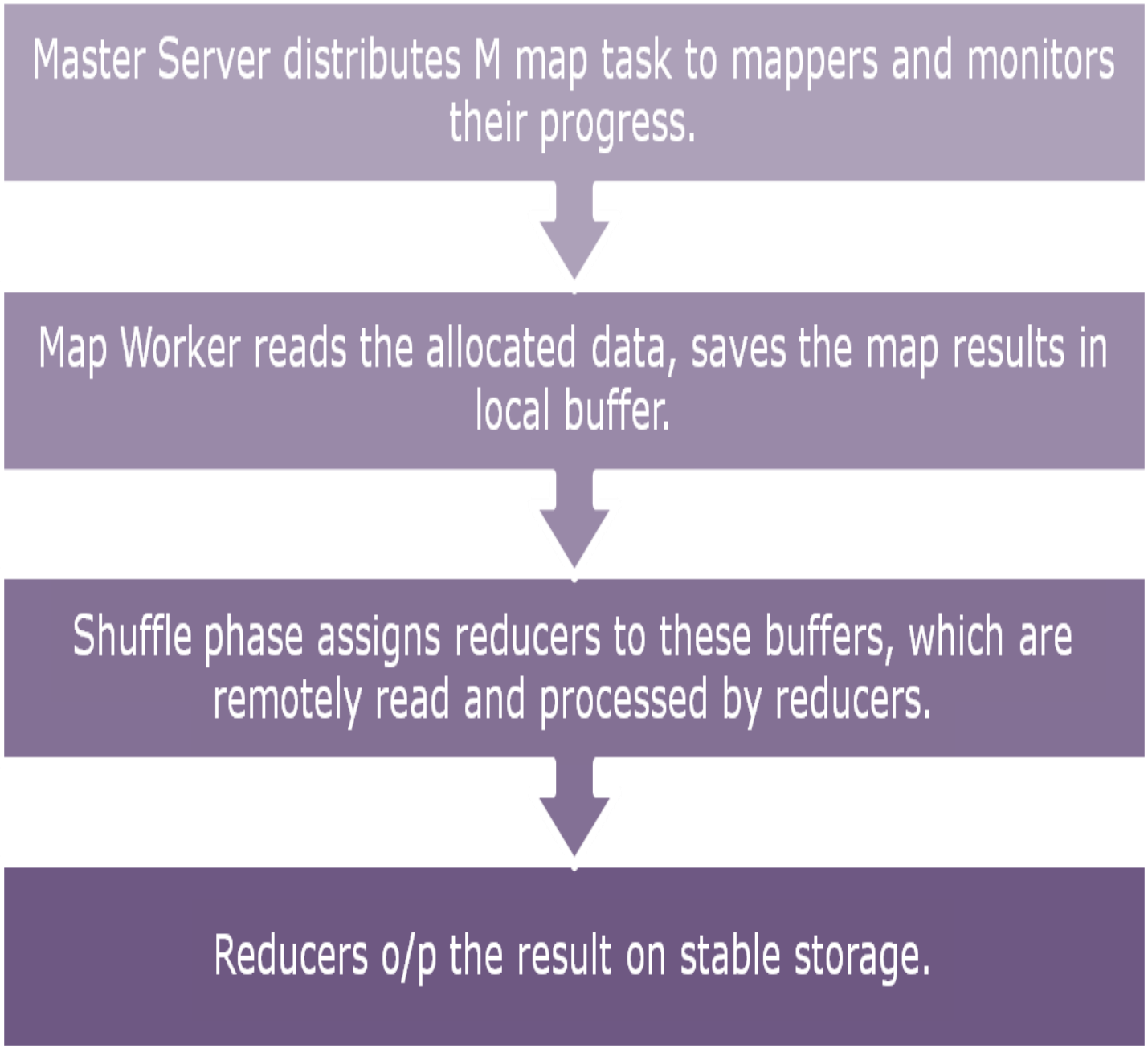
## Execution

- *Map* workers read in contents of corresponding input partition
- Perform user-defined *map* computation to create intermediate <key,value> pairs
- Periodically buffered output pairs written to local disk

## Reduce

- Reduce workers iterate over ordered intermediate data
  - Each unique key encountered – values are passed to user's reduce function
  - eg. <key, [value1, value2,..., valueN]>
- Output of user's *reduce* function is written to output file on global file system
- When all tasks have completed, master wakes up user program


Input 0
Input 1
Input 2
Map 0
Map 1
Map 2
Shuffle
Reduce 0
Reduce 1
Out 0
Out 1


Master Server distributes M map task to mappers and monitors their progress.
Map Worker reads the allocated data, saves the map results in local buffer.
Shuffle phase assigns reducers to these buffers, which are remotely read and processed by reducers.
Reducers o/p the result on stable storage.

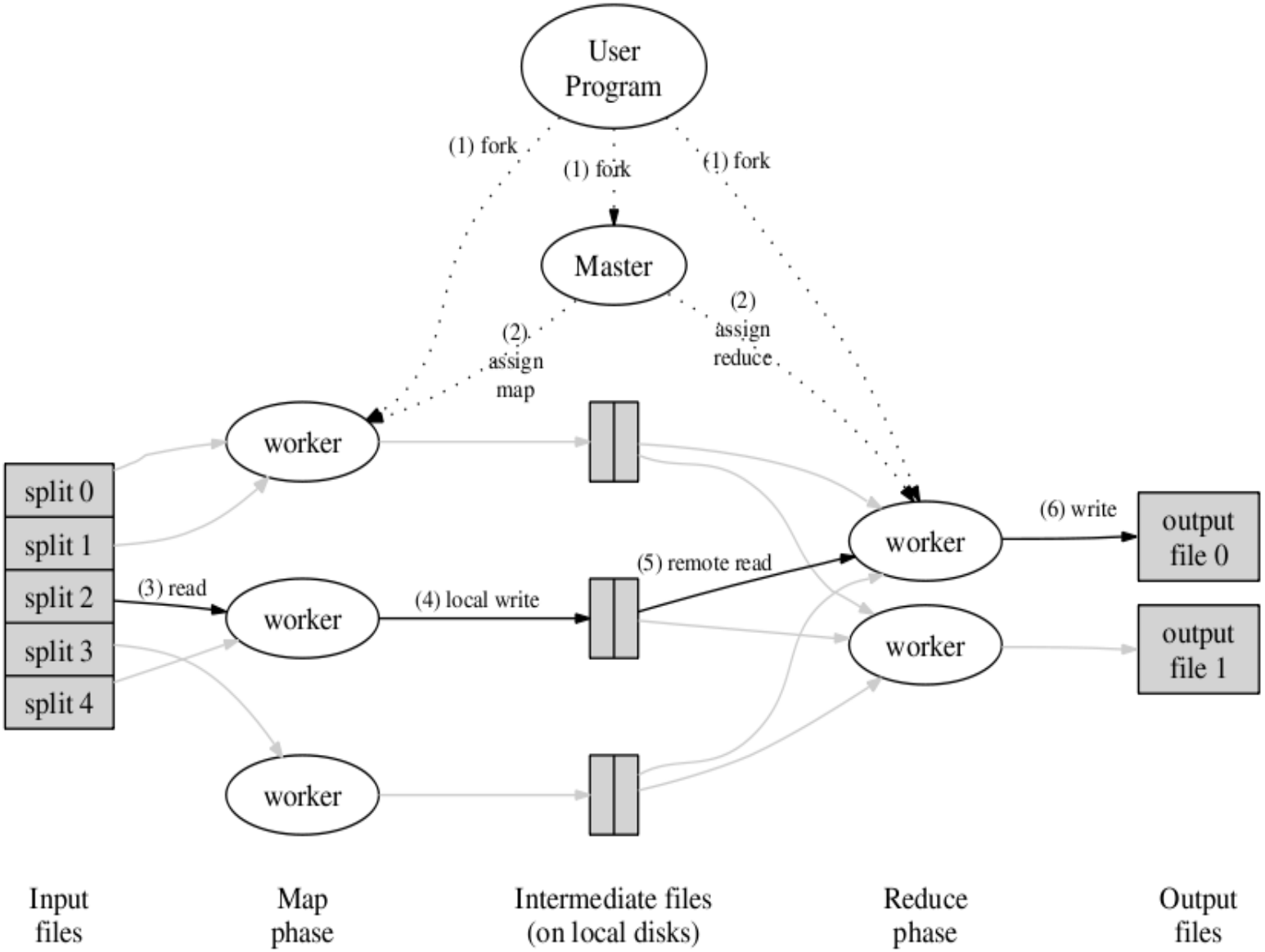



Figure 1: Execution overview

# DATA FLOW

- Input, final output are stored on a distributed file system
  - Scheduler tries to schedule map tasks “close” to physical storage location of input data
- Intermediate results are stored on local FS of map and reduce workers
- Output is often input to another map reduce task

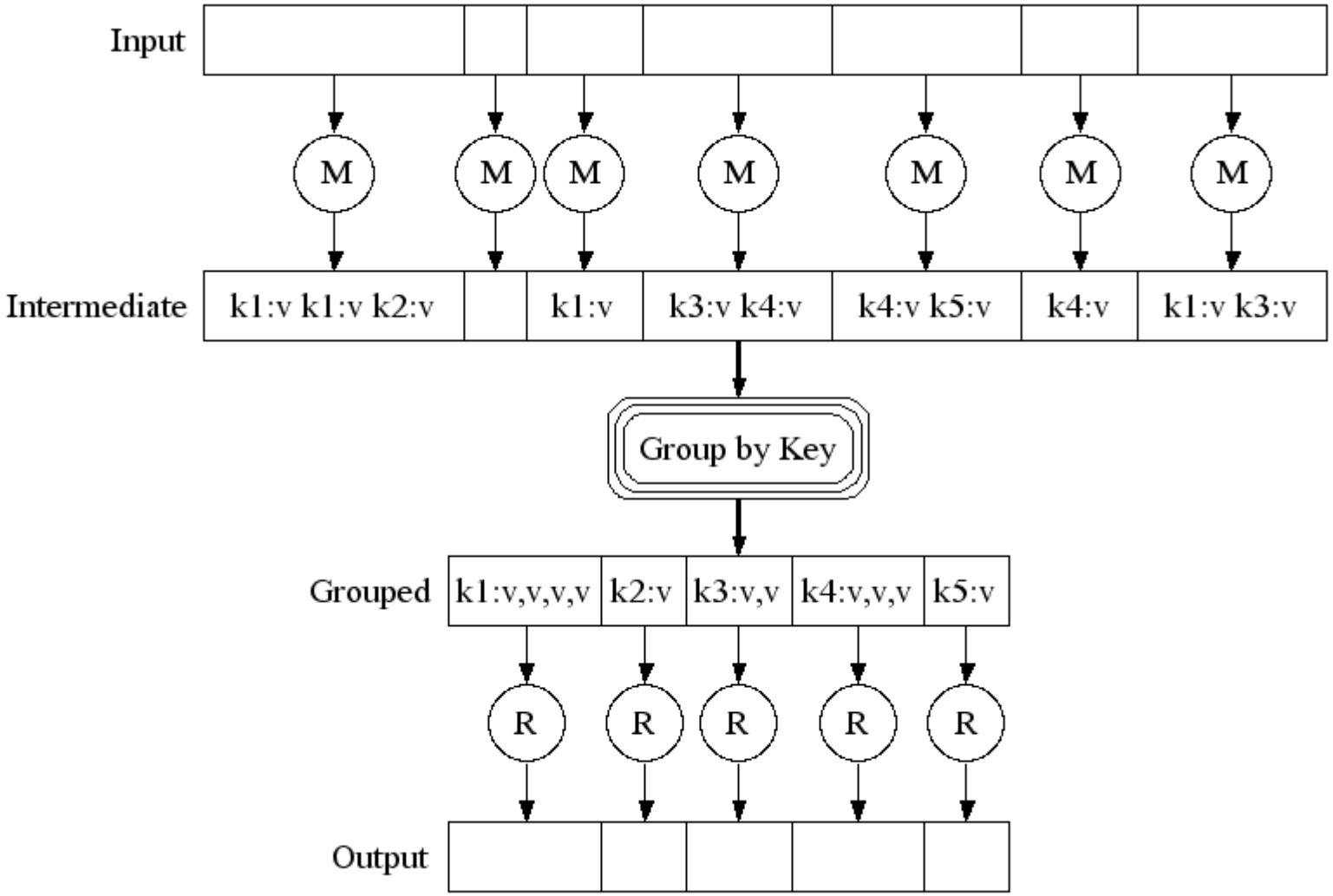
# COORDINATION

- Master data structures
  - Task status: (idle, in-progress, completed)
  - Idle tasks get scheduled as workers become available
  - When a map task completes, it sends the master the location and sizes of its R intermediate files, one for each reducer
  - Master pushes this info to reducers
- Master pings workers periodically to detect failures

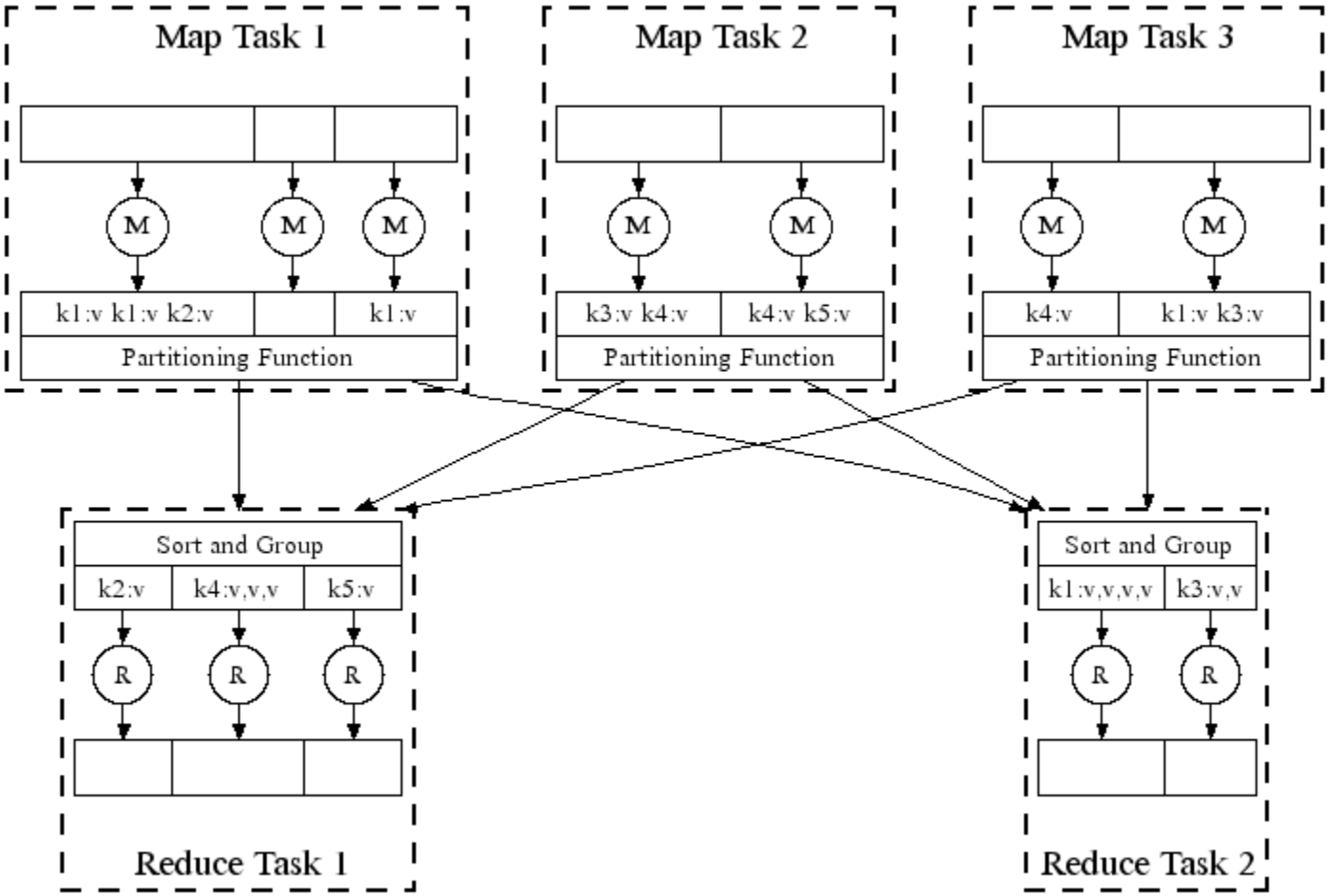
# FAILURES

- Map worker failure
  - Map tasks completed or in-progress at worker are reset to idle
  - Reduce workers are notified when task is rescheduled on another worker
- Reduce worker failure
  - Only in-progress tasks are reset to idle
- Master failure
  - MapReduce task is aborted and client is notified

# EXECUTION


Input
M
M
M
M
M
M
M
Intermediate
k1:v k1:v k2:v
k1:v
k3:v k4:v
k4:v k5:v
k4:v
k1:v k3:v
Group by Key
Grouped
k1:v,v,v,v
k2:v
k3:v,v
k4:v,v,v
k5:v
R
R
R
R
R
Output

# PARALLEL EXECUTION



# HOW MANY MAP AND REDUCE JOBS?

- M map tasks, R reduce tasks
- Rule of thumb:
  - Make M and R much larger than the number of nodes in cluster
  - One DFS chunk per map is common
  - Improves dynamic load balancing and speeds recovery from worker failure
- Usually R is smaller than M, because output is spread across R files

# COMBINERS

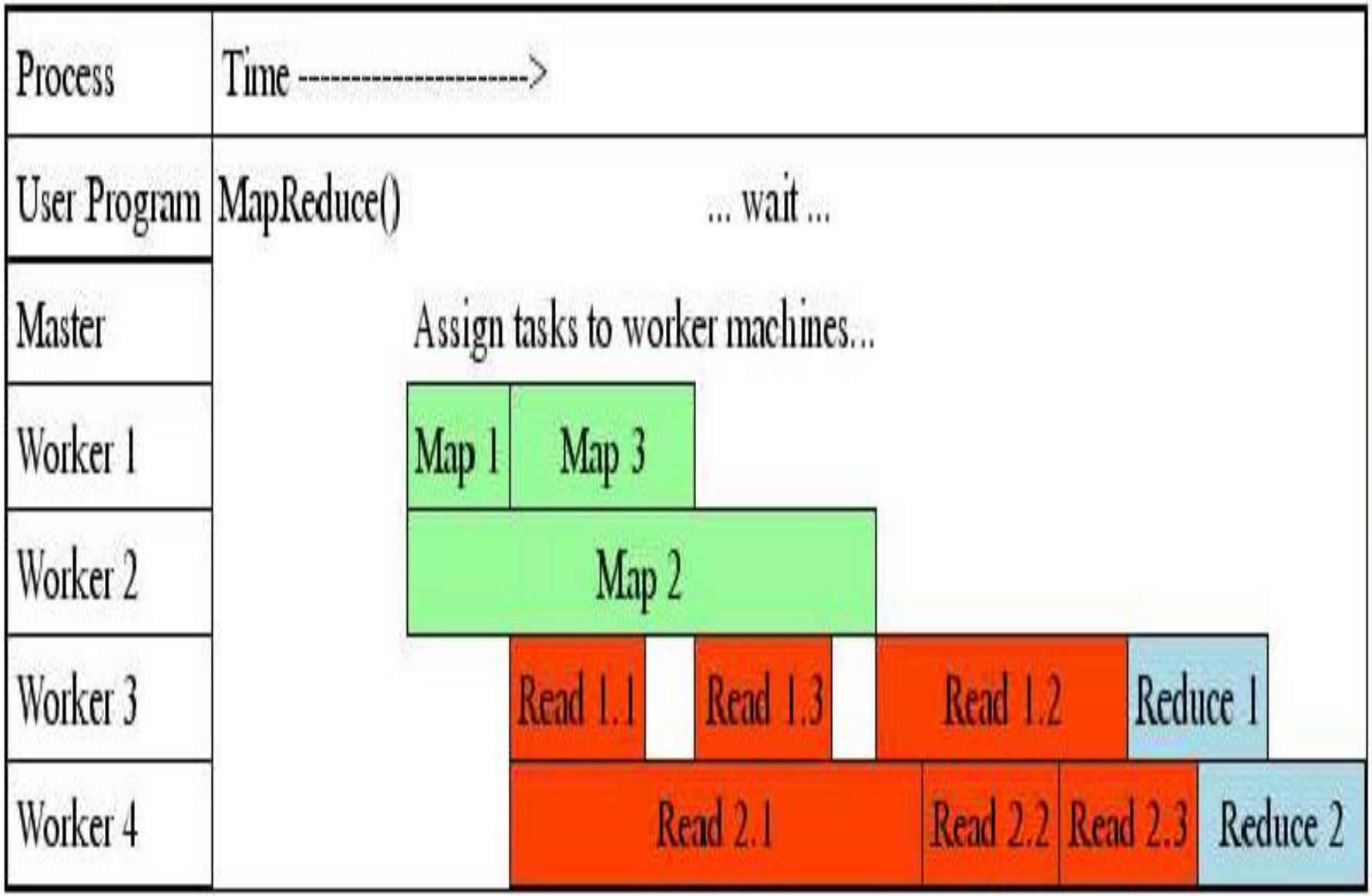
- Often a map task will produce many pairs of the form (k,v1), (k,v2), … for the same key k
  - E.g., popular words in Word Count
- Can save network time by pre-aggregating at mapper
  - combine(k1, list(v1)) → v2
  - Usually same as reduce function
- Works only if reduce function is commutative and associative

# PARTITION FUNCTION

- Inputs to map tasks are created by contiguous splits of input file
- For reduce, we need to ensure that records with the same intermediate key end up at the same worker
- System uses a default partition function e.g., hash(key) mod R
- Sometimes useful to override
  - E.g., hash(hostname(URL)) mod R ensures URLs from a host end up in the same output file

# EXECUTION SUMMARY

- How is this distributed?
  1. Partition input key/value pairs into chunks, run map() tasks in parallel
  2. After all map()s are complete, consolidate all emitted values for each unique emitted key
  3. Now partition space of output map keys, and run reduce() in parallel
- If map() or reduce() fails, reexecute!


Process
Time ---------->
User Program
MapReduce()
... wait ...
Master
Assign tasks to worker machines...
Worker 1
Map 1
Map 3
Worker 2
Map 2
Worker 3
Read 1.1
Read 1.3
Read 1.2
Reduce 1
Worker 4
Read 2.1
Read 2.2
Read 2.3
Reduce 2

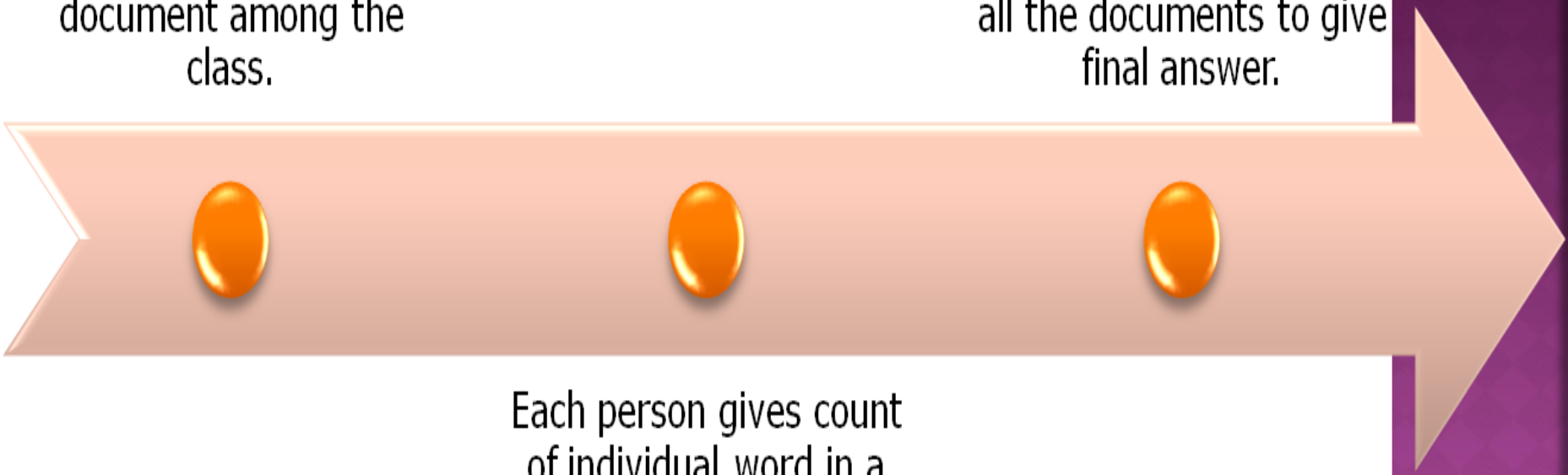
# WORD COUNT EXAMPLE

Divide collection of document among the class.

Each person gives count of individual word in a document. Repeats for assigned quota of documents.

(Done w/o communication )

Sum up the counts from all the documents to give final answer.



```
map(String input_key, String input_value):
// input_key: document name
// input_value: document contents
for each word w in input_value:
  EmitIntermediate(w, "1");
reduce(String output_key, Iterator
  intermediate_values):
// output_key: a word
// output_values: a list of counts
int result = 0;
for each v in intermediate_values:
  result += ParseInt(v);
Emit(AsString(result));
```

- Map()
  - Input <filename, file text>
  - Parses file and emits <word, count> pairs
    - eg. <”hello”, 1>
- Reduce()
  - Sums all values for the same key and emits <word, TotalCount>
    - eg. <”hello”, (3 5 2 7)>  => <”hello”, 17>

- File
  Hello World Bye World
  Hello Hadoop GoodBye Hadoop
- Map

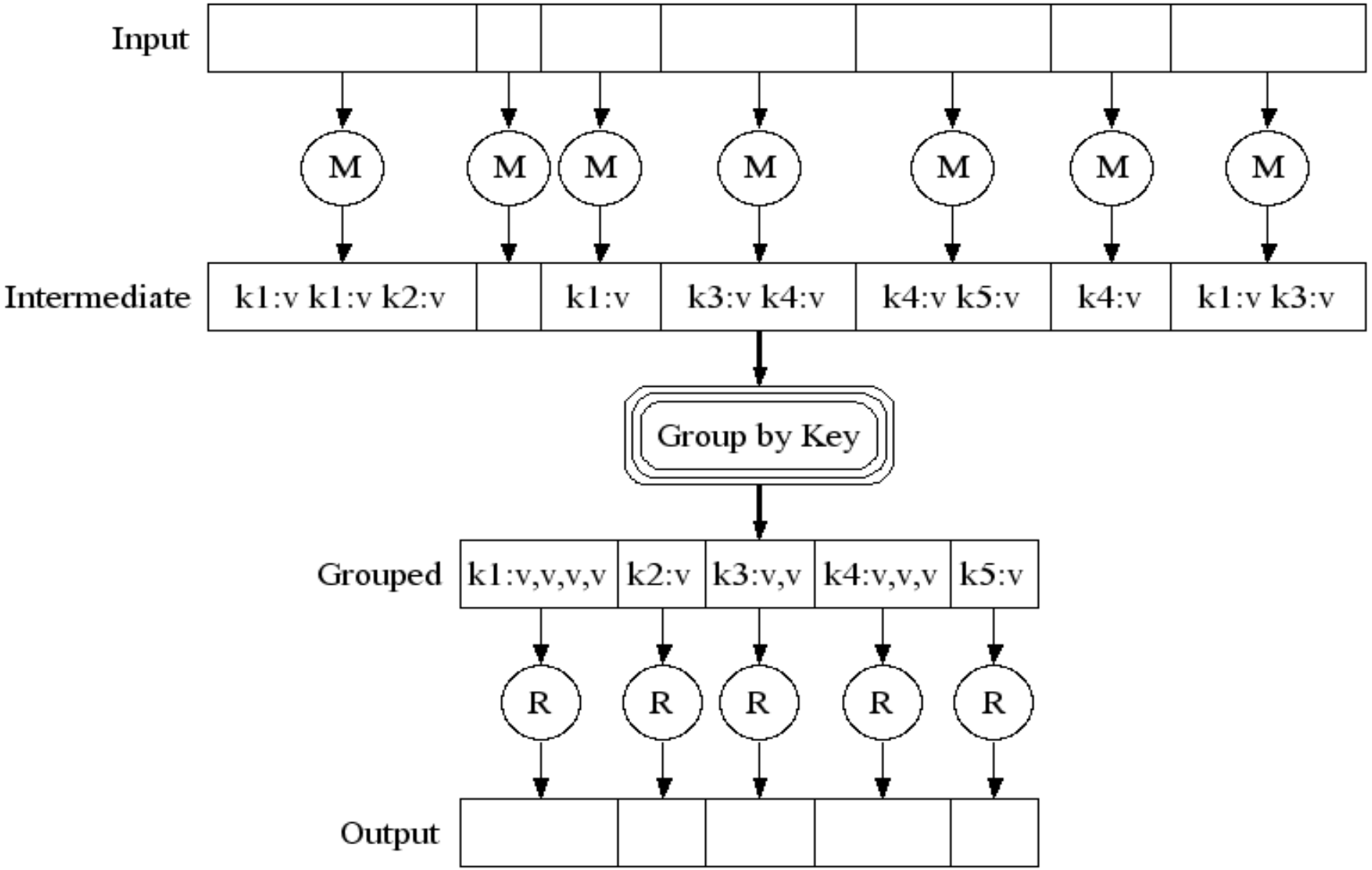
For the given sample input the first map emits:

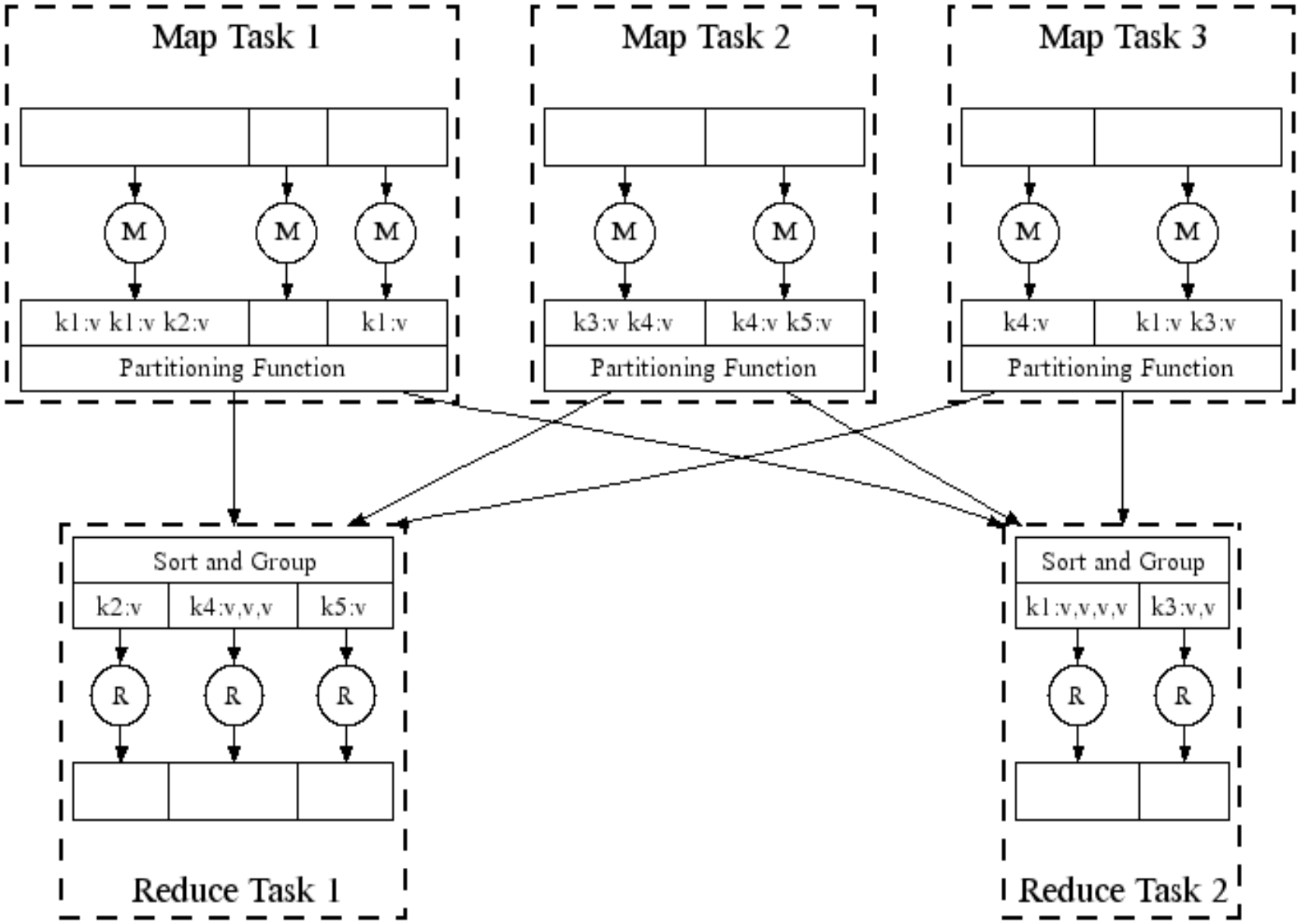
< Hello, 1>
< World, 1>
< Bye, 1>
< World, 1>

- The second map emits:

< Hello, 1>
< Hadoop, 1>
< Goodbye, 1>
< Hadoop, 1>

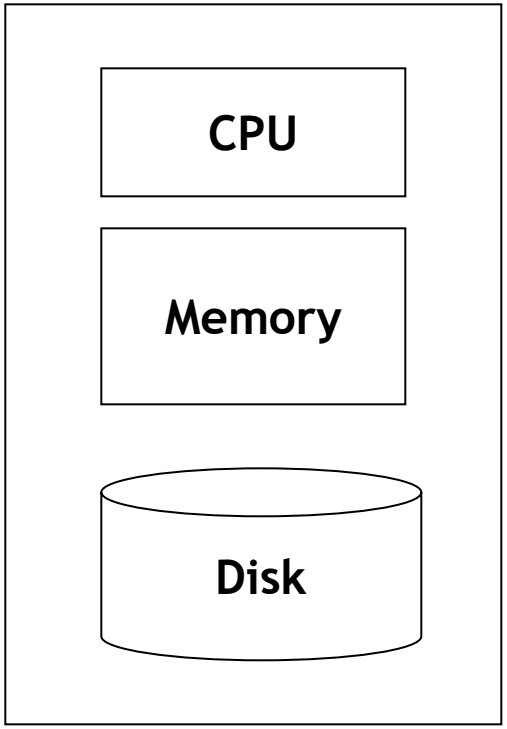
The output of the first combine:
< Bye, 1>
< Hello, 1>
< World, 2>
The output of the second combine:
< Goodbye, 1>
< Hadoop, 2>
< Hello, 1>
Thus the output of the job (reduce) is:
< Bye, 1>
< Goodbye, 1>
< Hadoop, 2>
< Hello, 2>
< World, 2>


Input
M
M
M
M
M
M
M
Intermediate
k1:v k1:v k2:v
k1:v
k3:v k4:v
k4:v k5:v
k4:v
k1:v k3:v
Group by Key
Grouped
k1:v,v,v,v
k2:v
k3:v,v
k4:v,v,v
k5:v
R
R
R
R
R
Output


Map Task 1
Map Task 2
Map Task 3
M
M
M
M
M
M
M
M
k1:v k1:v k2:v
k1:v
k3:v k4:v
k4:v k5:v
k4:v
k1:v k3:v
Partitioning Function
Partitioning Function
Partitioning Function
Sort and Group
k2:v
k4:v,v,v
k5:v
Sort and Group
k1:v,v,v,v
k3:v,v
R
R
R
R
R
Reduce Task 1
Reduce Task 2

# MAP REDUCE ARCHITECTURE

# SINGLE-NODE ARCHITECTURE



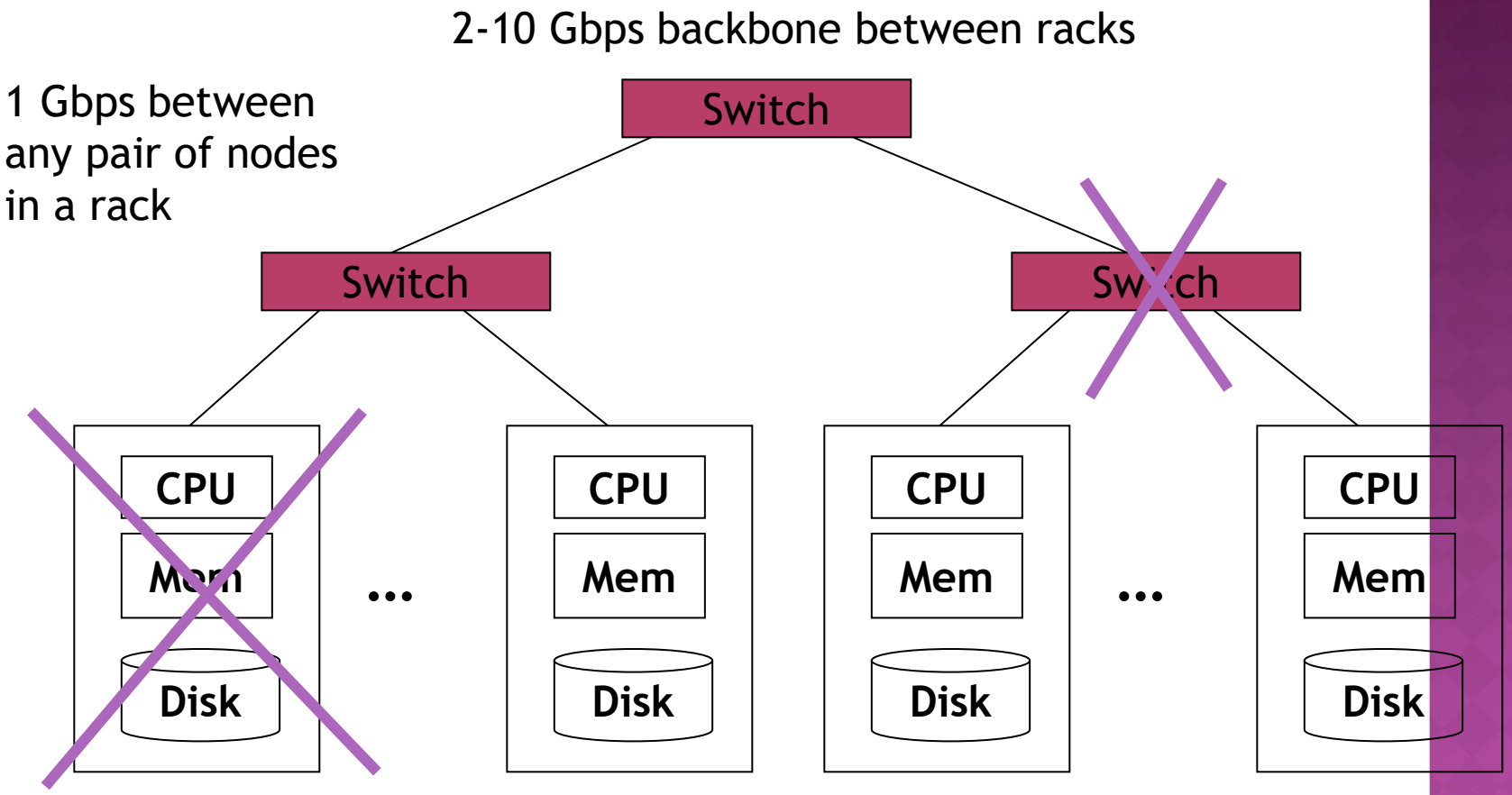

Machine Learning, Statistics

"Classical" Data Mining

# COMMODITY CLUSTERS

- Web data sets can be very large
  - Tens to hundreds of terabytes
- Cannot mine on a single server (why?)
- Standard architecture emerging:
  - Cluster of commodity Linux nodes
  - Gigabit ethernet interconnect
- How to organize computations on this architecture?
  - Mask issues such as hardware failure

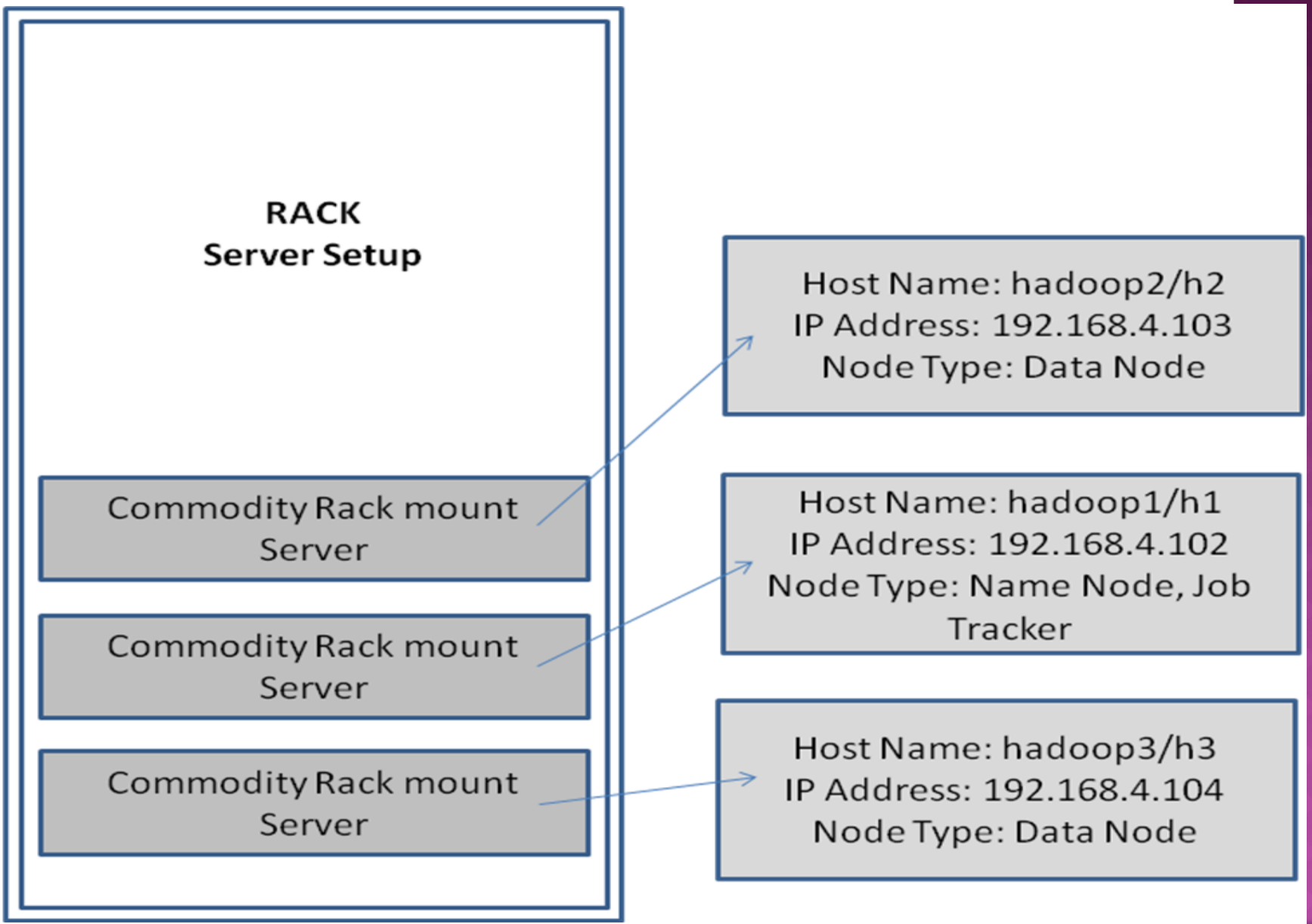
# CLUSTER ARCHITECTURE



# STABLE STORAGE

- First order problem: if nodes can fail, how can we store data persistently?
- Answer: Distributed File System
  - Provides global file namespace
  - Google GFS; Hadoop HDFS; Kosmix KFS
- Typical usage pattern
  - Huge files (100s of GB to TB)
  - Data is rarely updated in place
  - Reads and appends are common

# CONFIGURATION



# HADOOP : MAP REDUCE FRAMEWORK

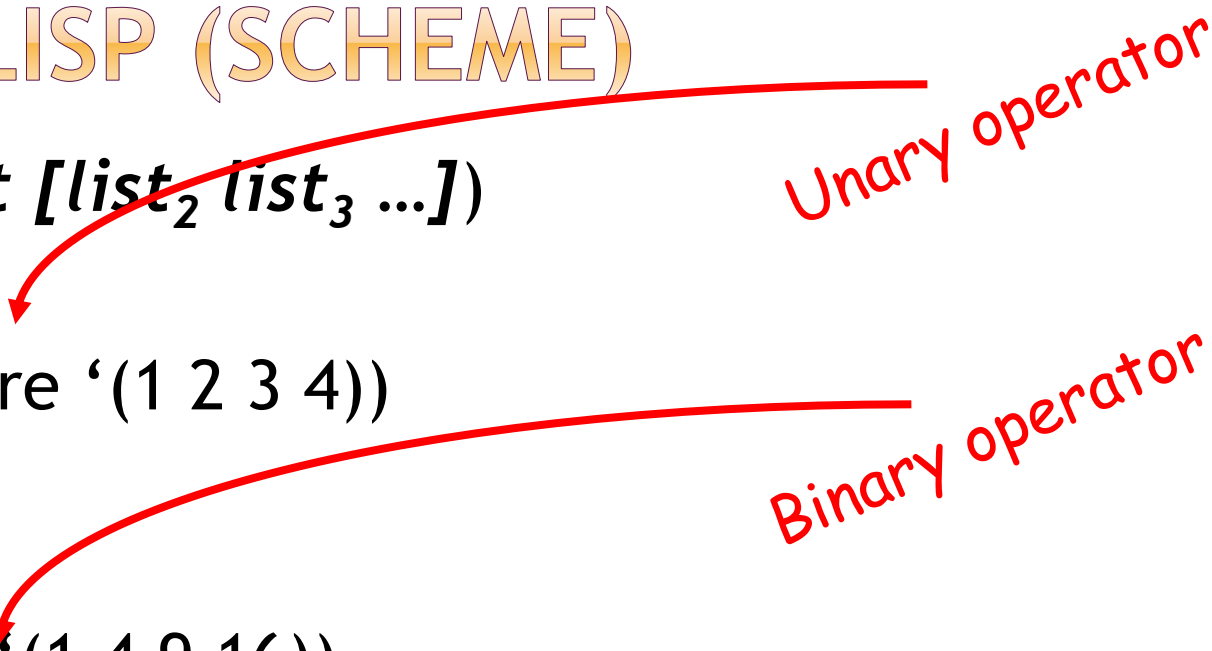
**Hadoop Map-Reduce is a software framework** for easily writing **applications** which **process vast amounts of data** (multi-terabyte data-sets) **in-parallel** on large clusters (thousands of nodes) of **commodity** hardware in a **reliable, fault-tolerant manner**.

# FUNCTIONAL PROGRAMMING

- functional languages :  Take example of Programming model from Lisp

# MAP IN LISP (SCHEME)

- (map ***f list [list$_2$ list$_3$ …]***)
- (map square ‘(1 2 3 4))
  - (1 4 9 16)
- (reduce + ‘(1 4 9 16))
  - (+ 16 (+ 9 (+ 4 1) ) )
- (reduce + (map square (map – $l_1$ $l_2$))))



# FAULT TOLERANCE

- Workers are periodically pinged by master
  - No response = failed worker
- Master writes periodic checkpoints
- On errors, workers send “last gasp” UDP packet to master
  - Detect records that cause deterministic crashes and skips them

# FAULT TOLERANCE

- Input file blocks stored on multiple machines
- When computation almost done, reschedule in-progress tasks

# FAULT TOLERANCE / WORKERS

Handled via re-execution

- Detect failure via periodic heartbeats
- Re-execute completed + in-progress *map* tasks
  - Why????
- Re-execute in progress *reduce* tasks
- Task completion committed through master

# Master Failure

- Could handle, … ?
- But don't yet
  - (master failure unlikely)

# Refinement: Redundant Execution

Slow workers significantly delay completion time

- Other jobs consuming resources on machine
- Bad disks w/ soft errors transfer data slowly
- Weird things: processor caches disabled (!!)

Solution: Near end of phase, spawn backup tasks

- Whichever one finishes first "wins"

Dramatically shortens job completion time

# Refinement: Locality Optimization

- Master scheduling policy:
  - Asks GFS for locations of replicas of input file blocks
  - Map tasks typically split into 64MB (GFS block size)
  - Map tasks scheduled so GFS input block replica are on same machine or same rack
- Effect
  - Thousands of machines read input at local disk speed
    - Without this, rack switches limit read rate

# Refinement
# Skipping Bad Records

- Map/Reduce functions sometimes fail for particular inputs
  - Best solution is to debug & fix
    - Not always possible ~ third-party source libraries
  - On segmentation fault:
    - Send UDP packet to master from signal handler
    - Include sequence number of record being processed
  - If master sees two failures for same record:
    - Next worker is told to skip the record

# Other Refinements

- Sorting guarantees
  - within each reduce partition
- Compression of intermediate data
- Combiner
  - Useful for saving network bandwidth
- Local execution for debugging/testing
- User-defined counters

# PERFORMANCE

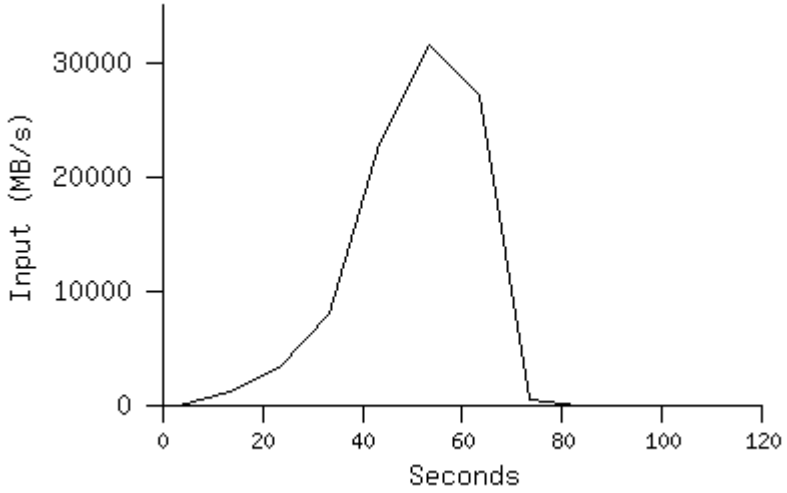
Tests run on cluster of 1800 machines:

- 4 GB of memory
- Dual-processor 2 GHz Xeons with Hyperthreading
- Dual 160 GB IDE disks
- Gigabit Ethernet per machine
- Bisection bandwidth approximately 100 Gbps

Two benchmarks:

| | |
|---|---|
| MR_GrepScan | 1010 100-byte records to extract records matching a rare pattern (92K matching records) |
| MR_SortSort | 1010 100-byte records (modeled after TeraSort benchmark) |

# MR_GREP



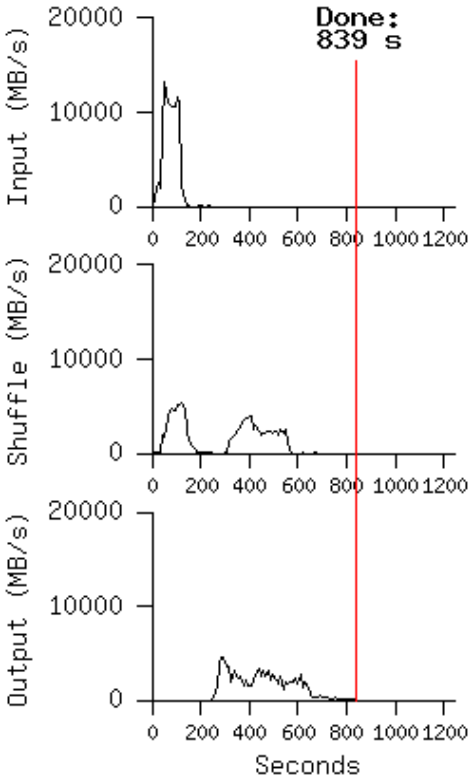

Locality optimization helps:

- 1800 machines read 1 TB at peak ~31 GB/s
- W/out this, rack switches would limit to 10 GB/s

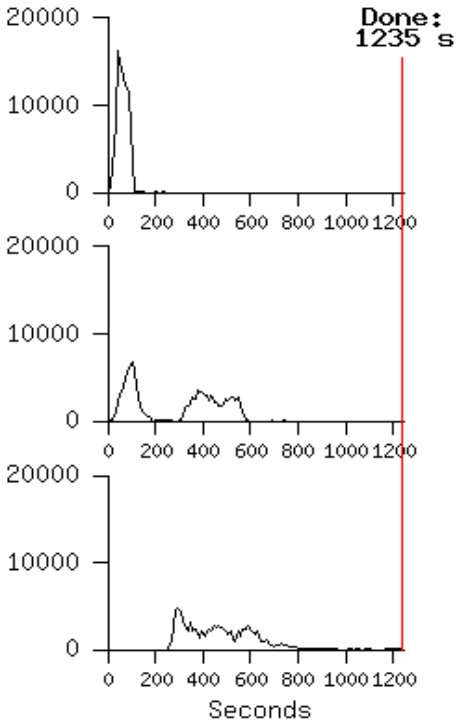
Startup overhead is significant for short jobs

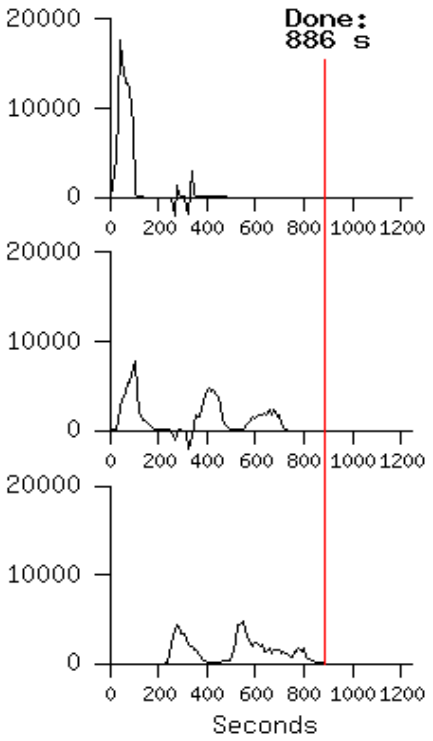
# MR_SORT

**Normal**




**No backup tasks 200 processes killed**




- Backup tasks reduce job completion time a lot!
- System deals well with failures

# USAGE IN AUG 2004

| | | |
|---|---|---|
| Number of jobs | 29,423 | |
| Average job completion time | 634 | secs |
| Machine days used | 79,186 | days |
| Input data read | 3,288 | TB |
| Intermediate data produced | 758 | TB |
| Output data written | 193 | TB |
| Average worker machines per job | 157 | |
| Average worker deaths per job | 1.2 | |
| Average map tasks per job | 3,351 | |
| Average reduce tasks per job | 55 | |
| Unique *map* implementations | 395 | |
| Unique *reduce* implementations | 269 | |
| Unique *map*/*reduce* combinations | 426 | |