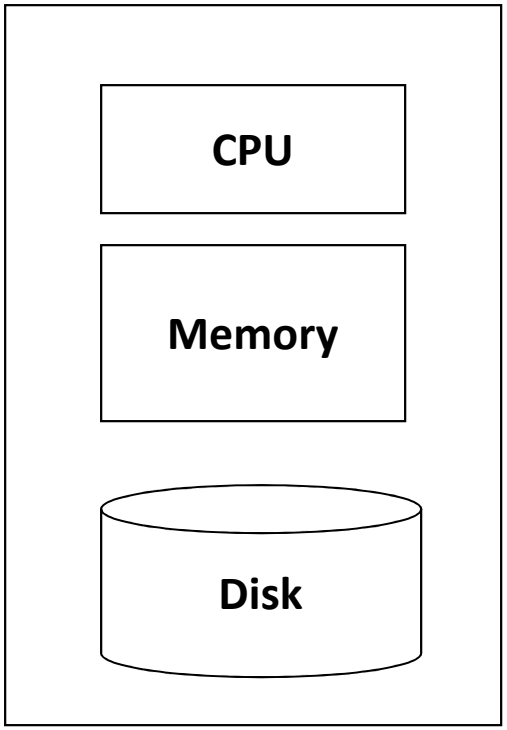
Map-Reduce

- Problem Characteristics:
  - Large scale – inherently parallel computations
- Network commodity PCs
- Employ data replication and redundant computations
- Use FP inspired programming model:
  - Map Reduce

Single-node architecture



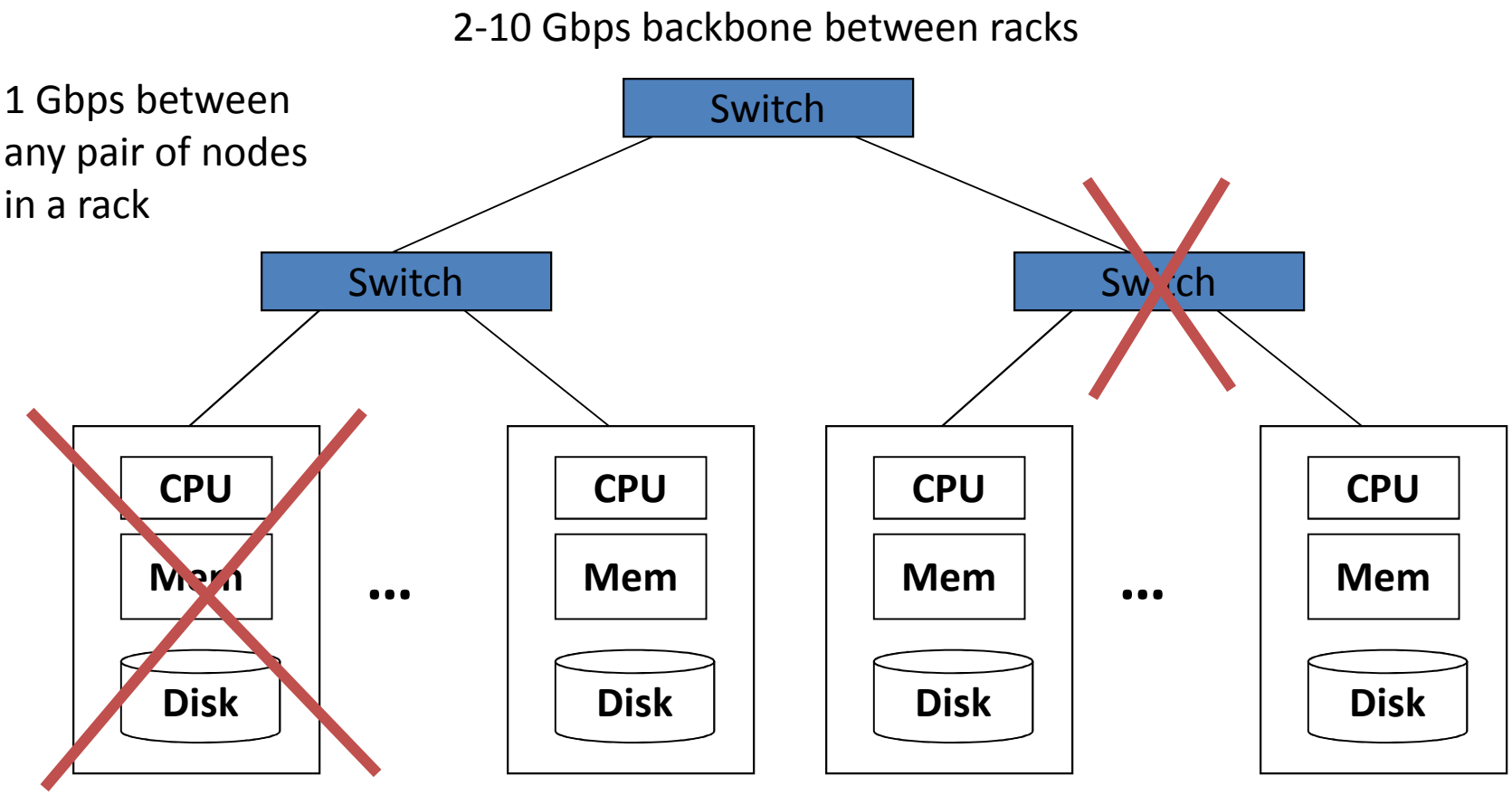
Machine Learning, Statistics

"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

Distributed File System

- Chunk Servers
  - File is split into contiguous chunks
  - Typically each chunk is 16-64MB
  - Each chunk replicated (usually 2x or 3x)
  - Try to keep replicas in different racks
- Master node
  - a.k.a. Name Nodes in HDFS
  - Stores metadata
  - Might be replicated
- Client library for file access
  - Talks to master to find chunk servers
  - Connects directly to chunkservers to access data

Motivation for MapReduce (why)

- Large-Scale Data Processing
  - Want to use 1000s of CPUs
    - But don't want hassle of ***managing*** things

- MapReduce Architecture provides
  - Automatic parallelization & distribution
  - Fault tolerance
  - I/O scheduling
  - Monitoring & status updates

What is Map/Reduce

- Map/Reduce
  - Programming model from LISP
  - (and other functional languages)

- Many problems can be phrased this way

- Easy to distribute across nodes
- Nice retry/failure semantics

Map in LISP (Scheme)

- (map ***f list*** $[$***list***$_2$ ***list***$_3$ ***…***$]$)

- (map square ‘(1 2 3 4))
  - (1 4 9 16)
- (reduce ***f id list***)

Warm up: Word Count

- We have a large file of words, one word to a line
- Count the number of times each distinct word appears in the file

- *Sample application*: analyze web server logs to find popular URLs

Word Count (2)

- Case 1: Entire file fits in memory
- Case 2: File too large for mem, but all <word, count> pairs fit in mem
- Case 3: File on disk, too many distinct words to fit in memory
  - **`sort datafile | uniq -c`**

Word Count (3)

- To make it slightly harder, suppose we have a large corpus of documents
- Count the number of times each distinct word occurs in the corpus

  `words(docs/*) | sort | uniq -c`

  where **`words`** takes a file and outputs the words in it, one to a line
- The above captures the essence of MapReduce
  - Great thing is it is naturally parallelizable

MapReduce

- Input: a set of key/value pairs
- User supplies two functions:
  - map(k,v) → list(k1,v1)
  - reduce(k1, list(v1)) → v2
- (k1,v1) is an intermediate key/value pair
- Output is the set of (k1,v2) pairs

Word Count using MapReduce

```
map(key, value):
// key: document name; value: text of document
    for each word w in value:
        emit(w, 1)


reduce(key, values):
// key: a word; values: an iterator over counts
        result = 0
        for each count v in values:
            result += v
        emit(key,result)
```

Count, Illustrated

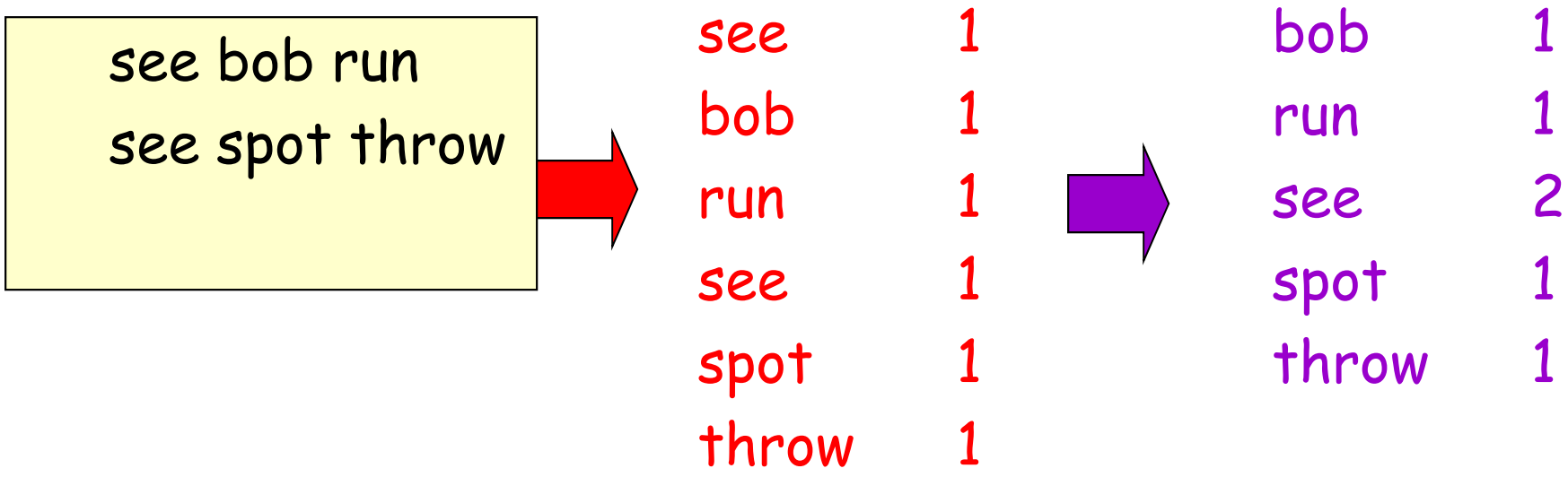
map(key=url, val=contents):

For each word *w* in contents, emit (w, "1")

reduce(key=word, values=uniq_counts):

Sum all "1"s in values list

Emit result "(word, sum)"



Implementation Overview

Typical cluster:

- 100s/1000s of 2-CPU x86 machines, 2-4 GB of memory
- Limited bisection bandwidth
- Storage is on local IDE disks
- GFS: distributed file system manages data (SOSP'03)
- Job scheduling system: jobs made up of tasks, scheduler assigns tasks to machines

Implementation is a C++ library linked into user programs

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

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!