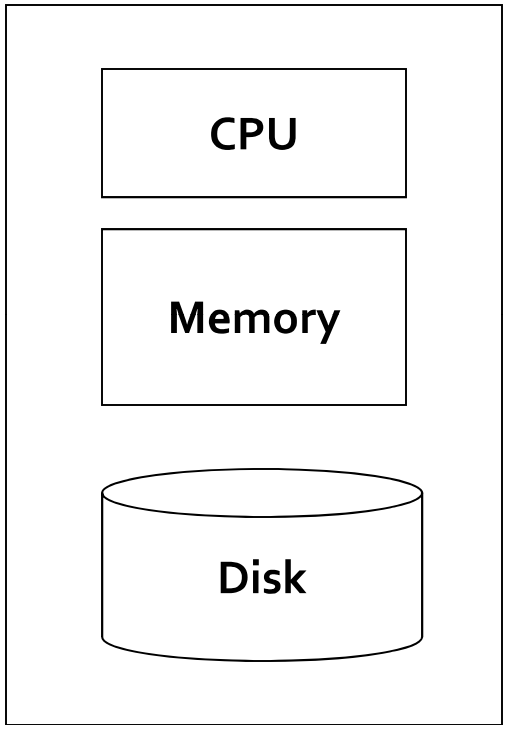
MapReduce

CS345a: Data Mining
Jure Leskovec
Stanford University

Single-node architecture



Machine Learning, Statistics

"Classical" Data Mining

Motivation (Google example)

- 20+ billion web pages x 20KB = 400+ TB
- 1 computer reads 30-35 MB/sec from disk
  - ~4 months to read the web
- ~1,000 hard drives to store the web

- Even more to do something with the data

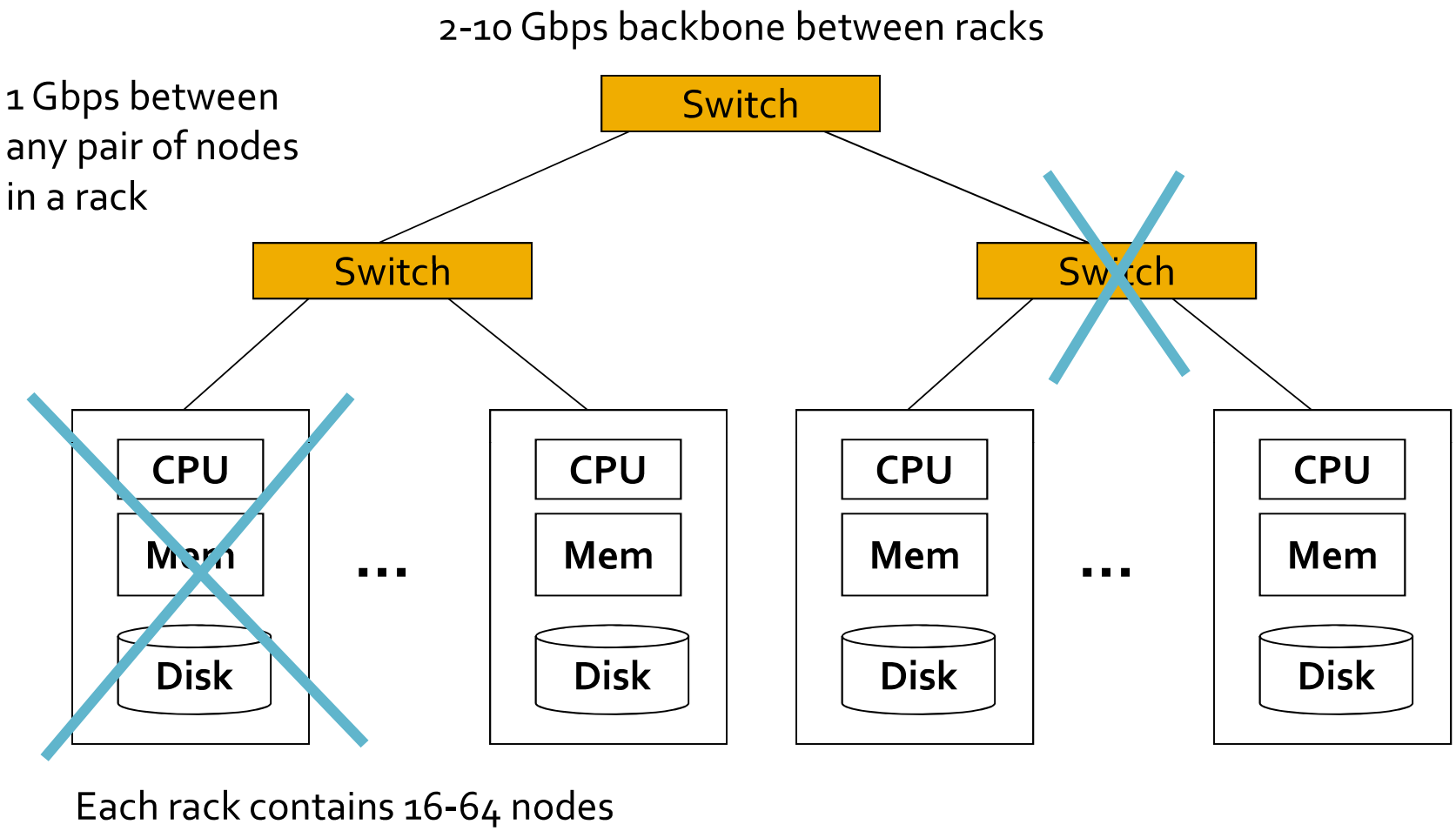
Commodity Clusters

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

Big computation – Big machines

- Traditional big-iron box (circa 2003)
  - 8 2GHz Xeons
  - 64GB RAM
  - 8TB disk
  - 758,000 USD
- Prototypical Google rack (circa 2003)
  - 176 2GHz Xeons
  - 176GB RAM
  - ~7TB disk
  - 278,000 USD
- In Aug 2006 Google had ~450,000 machines

Cluster Architecture



Large scale computing

- **Large scale computing** for **data mining** problems on **commodity hardware**
  - PCs connected in a network
  - Need to process huge datasets on large clusters of computers
- Challenges:
  - How do you distribute computation?
  - Distributed programming is hard
  - Machines fail
- **Map-reduce** addresses all of the above
  - Google's computational/data manipulation model
  - Elegant way to work with big data

M45: Open Academic Cluster

- Yahoo's collaboration with academia
  - Foster open research
  - Focus on large-scale, highly parallel computing
- Seed Facility: **M45**
  - Datacenter in a Box (DiB)
  - 1000 nodes, 4000 cores, 3TB RAM, 1.5PB disk
  - High bandwidth connection to Internet
  - Located on Yahoo! corporate campus
  - World's top 50 supercomputer

Implications

- Implications of such computing environment
  - Single machine performance does not matter
    - Add more machines
  - Machines break
    - One server may stay up 3 years (1,000 days)
    - If you have 1,0000 servers, expect to loose 1/day
  - How can we make it easy to write distributed programs?

Idea and solution

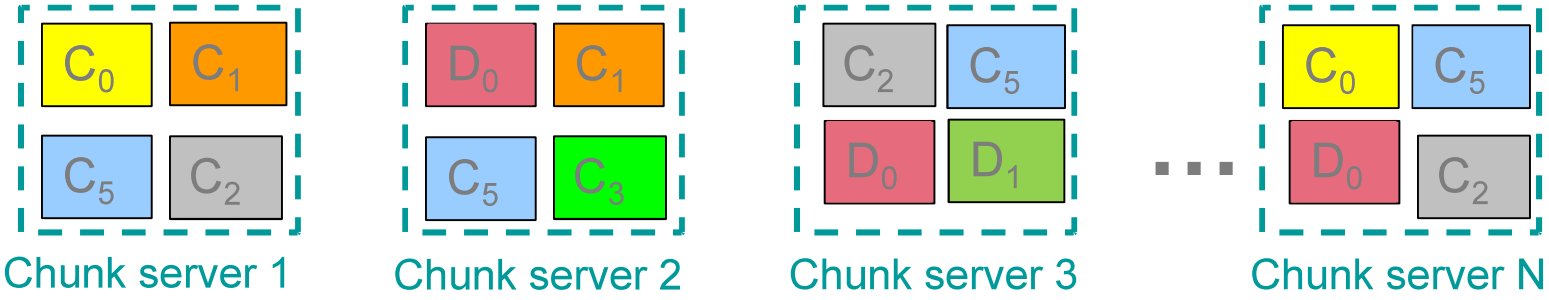
- **Idea**
  - Bring computation close to the data
  - Store files multiple times for reliability
- **Need**
  - Programming model
    - Map-Reduce
  - Infrastructure – File system
    - Google: GFS
    - Hadoop: HDFS

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

- Reliable distributed file system for petabyte scale
- Data kept in 64-megabyte "chunks" spread across thousands of machines
- Each chunk replicated, usually 3 times, on different machines
  - Seamless recovery from disk or machine failure



Bring computation directly to the data!

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

Warm up: Word Count

- We have a large file of words:
  - one word per 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

Map-Reduce: Overview

- Read a lot of data
- **Map**
  - Extract something you care about
- Shuffle and Sort
- **Reduce**
  - Aggregate, summarize, filter or transform
- Write the data

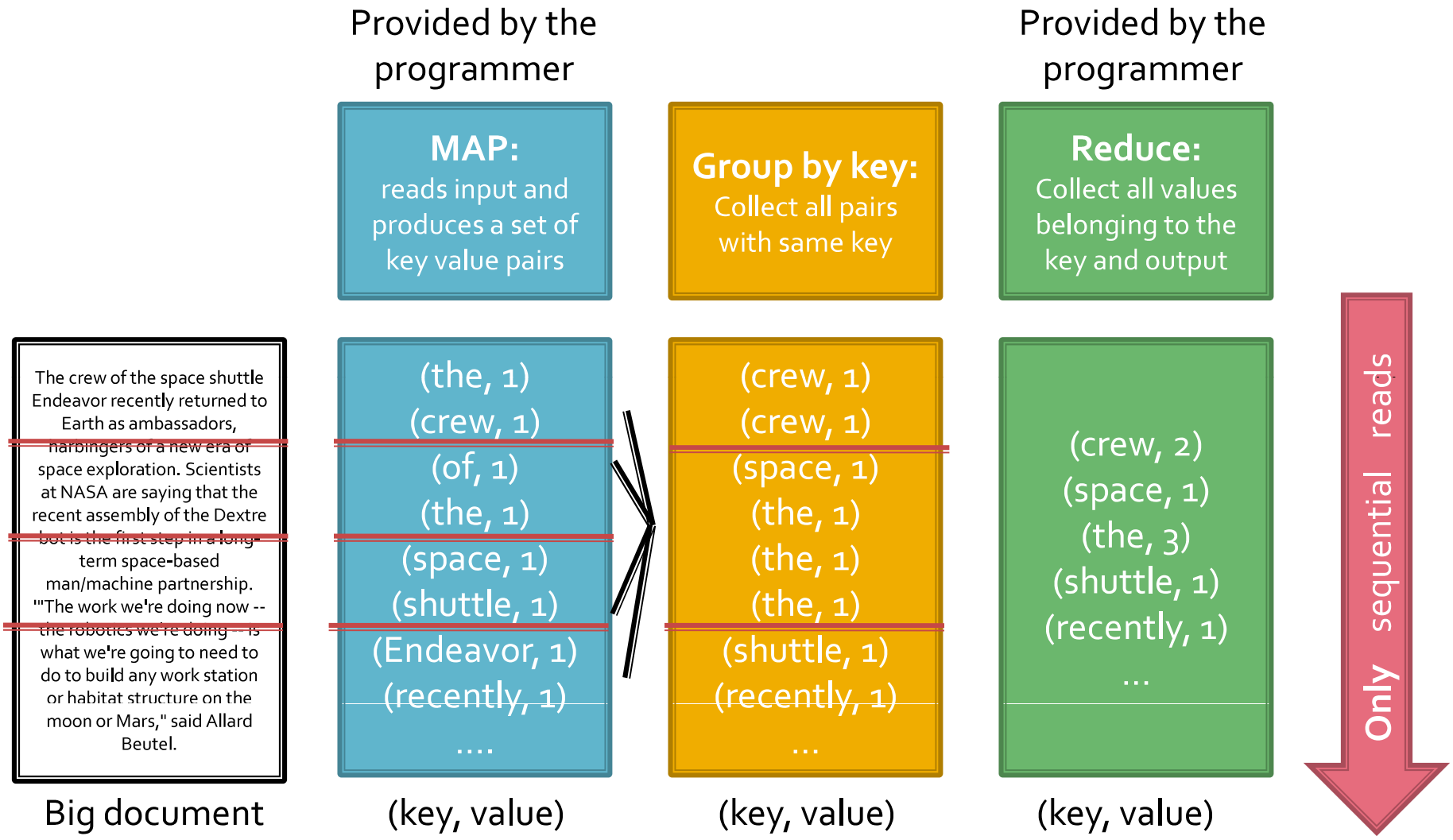
Outline stays the same, **map** and **reduce** change to fit the problem

More specifically

- Program specifies two primary methods:
  - Map(k,v) → <k', v'>*
  - Reduce(k', <v'>*) → <k', v''>*

- All v' with same k' are reduced together and processed in v' order

Map-Reduce: Word counting



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; value: an iterator over counts
        result = 0
        for each count v in values:
            result += v
        emit(result)
```

Example 1: Host size

- Suppose we have a large web corpus
- Let's look at the metadata file
  - Lines of the form (URL, size, date, …)
- For each host, find the total number of bytes
  - i.e., the sum of the page sizes for all URLs from that host

Example 2: Language model

- Statistical machine translation:
  - Need to count number of times every 5-word sequence occurs in a large corpuse of duments
- Easy with MapReduce:
  - Map: extract (5-word sequence, count) from document
  - Reduce: combine counts

Map-Reduce: Environment

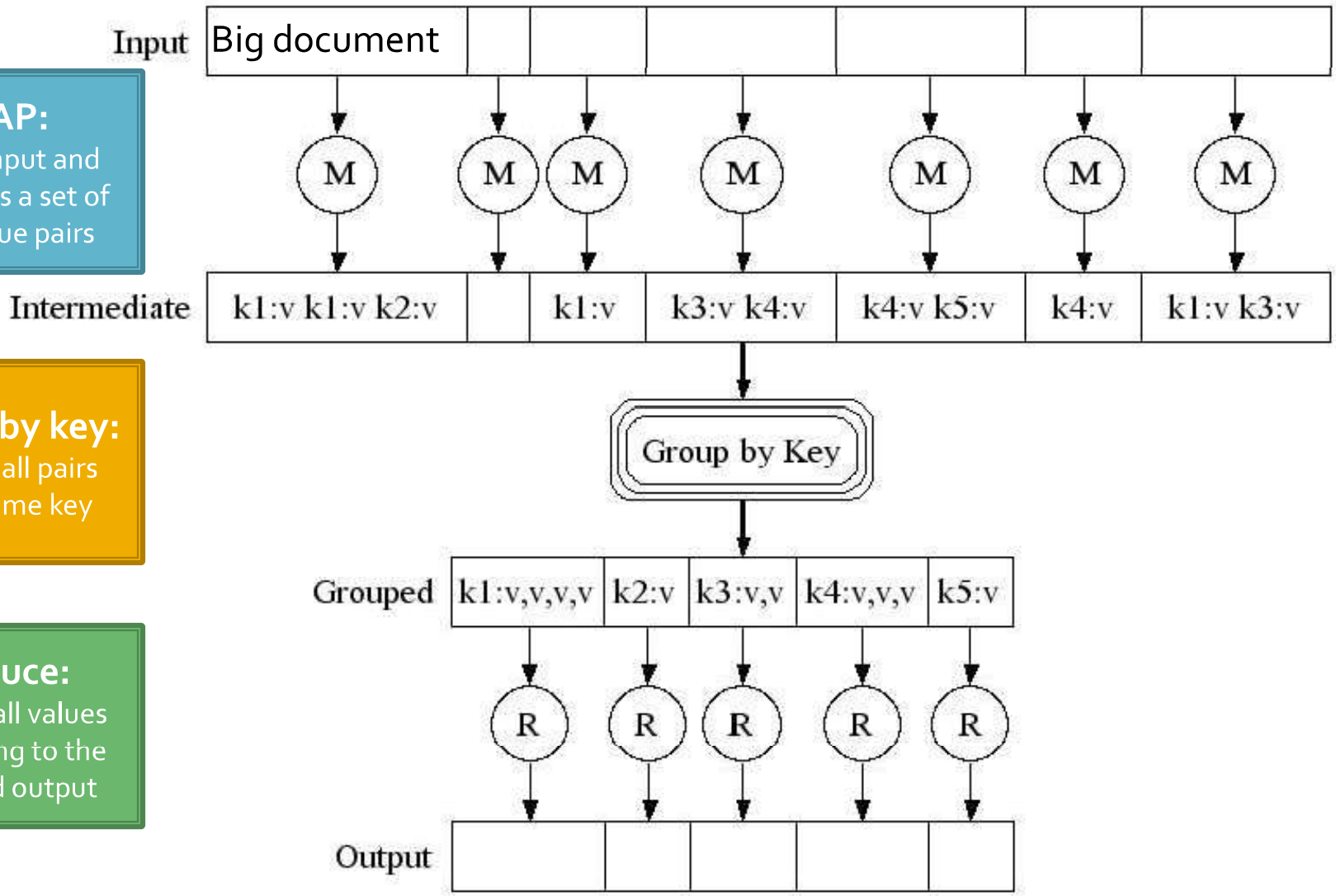
- Map-Reduce environment takes care of:
  - Partitioning the input data
  - Scheduling the program's execution across a set of machines
  - Handling machine failures
  - Managing required inter-machine communication
- Allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed cluster

Map-Reduce: A diagram

MAP:
reads input and produces a set of key value pairs

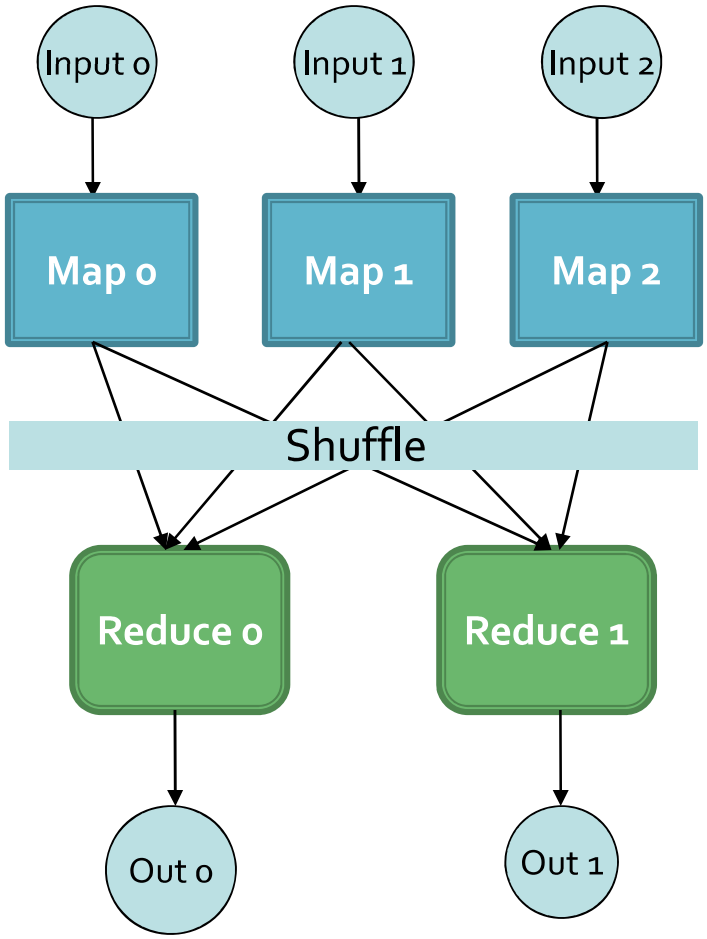
Group by key:
Collect all pairs with same key

Reduce:
Collect all values belonging to the key and output

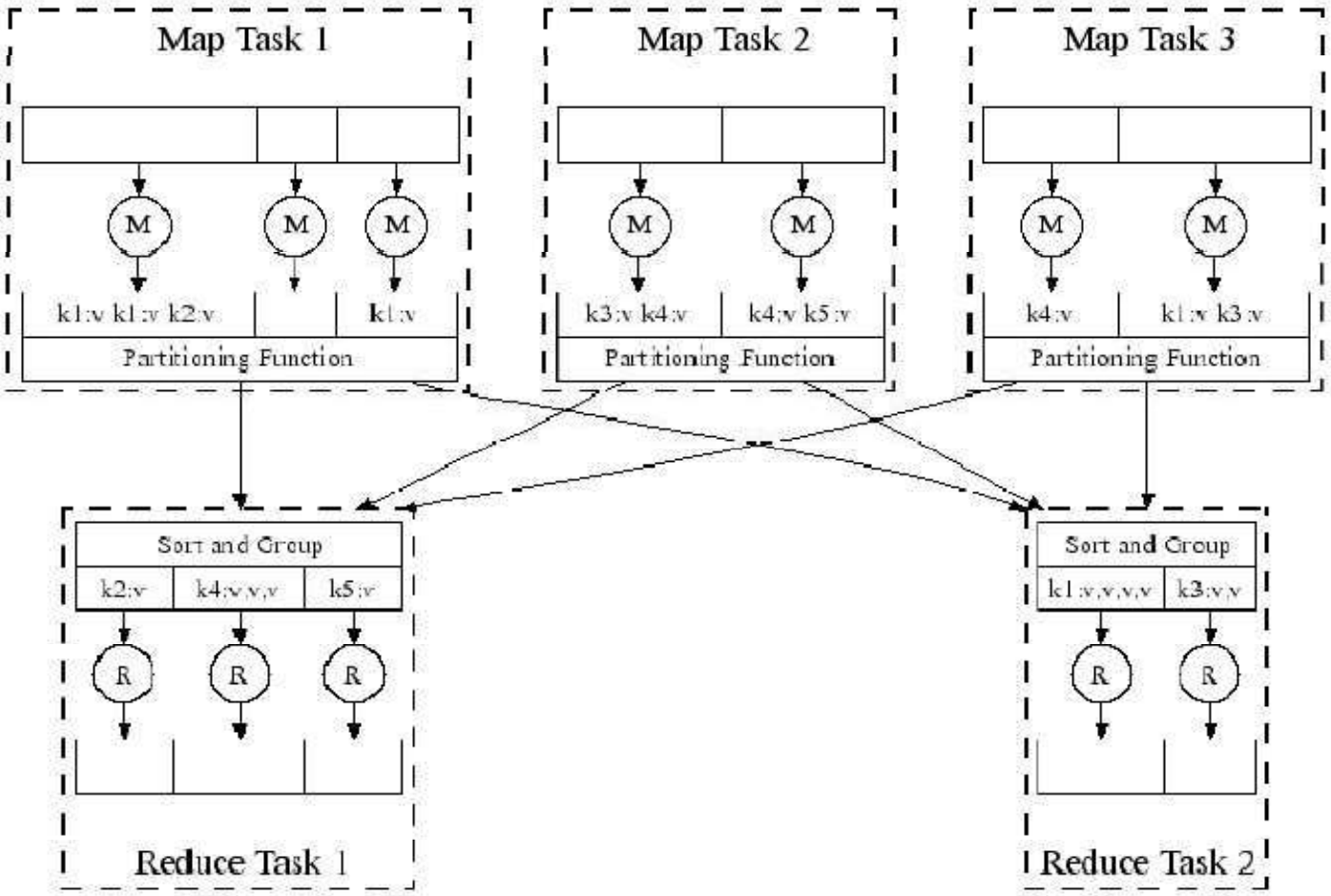


Map-Reduce

- Programmer specifies
  - Map and Reduce and input files
- **Workflow**
  - Read inputs as a set of key-value-pairs
  - **Map** transforms input kv-pairs into a new set of k'v'-pairs
  - Sorts & Shuffles the k'v'-pairs to output nodes
  - All k'v'-pairs with a given k' are sent to the same **reduce**
  - **Reduce** processes all k'v'-pairs grouped by key into new k''v''-pairs
  - Write the resulting pairs to files
- All phases are distributed with many tasks doing the work



Map-Reduce: in Parallel



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

Task Granularity & Pipelining

- Fine granularity tasks: map tasks >> machines
  - Minimizes time for fault recovery
  - Can pipeline shuffling with map execution
  - Better dynamic load balancing
- Often use 200,000 map & 5,000 reduce tasks
- Running on 2,000 machines