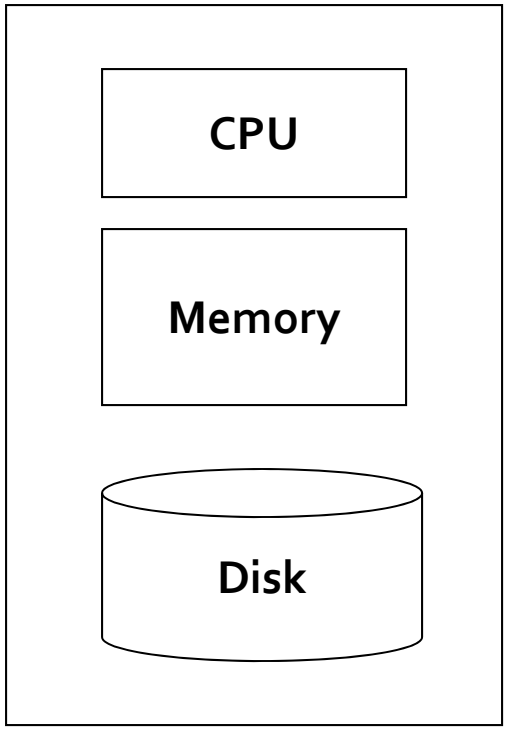
Map Reduce Framework

MapReduce

- Much of the course will be devoted to **large scale computing** for **data mining**
- **Challenges:**
  - How to distribute computation?
  - Distributed/parallel programming is hard
- **Map-reduce** addresses all of the above
  - Google's computational/data manipulation model
  - Elegant way to work with big data

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
- Takes even more to **do** something useful with the data!
- **Today, a standard architecture for such problems is emerging:**
  - Cluster of commodity Linux nodes
  - Commodity network (ethernet) to connect them

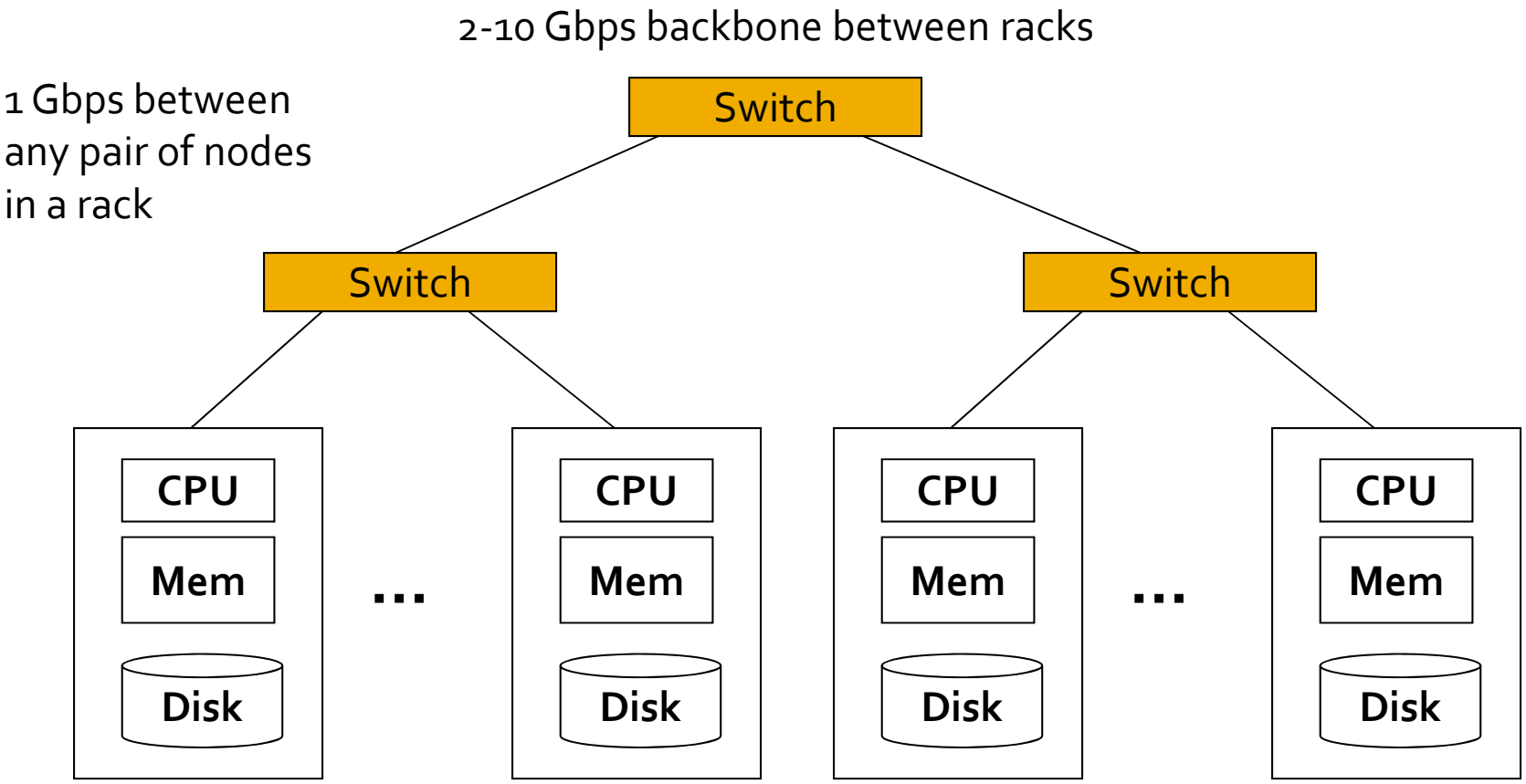
Commodity Clusters

- Web data sets are massive
  - 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



Each rack contains 16-64 nodes

In 2011 it was guestimated that Google had 1M machines, http://bit.ly/Shh0RO

M45: Open Academic Cluster

- Yahoo M45 cluster:
  - Datacenter in a Box (DiB)
  - 1000 nodes, 4000 cores, 3TB RAM, 1.5PB disk
  - High bandwidth connection to Internet
  - Located on Yahoo! campus
  - World's top 50 supercomputer

Large scale computing

- **Large scale computing** for **data mining** problems on **commodity hardware:**
  - PCs connected in a network
  - Process huge datasets on many computers
- Challenges:
  - How do you distribute computation?
  - Distributed/parallel 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

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 1,000 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

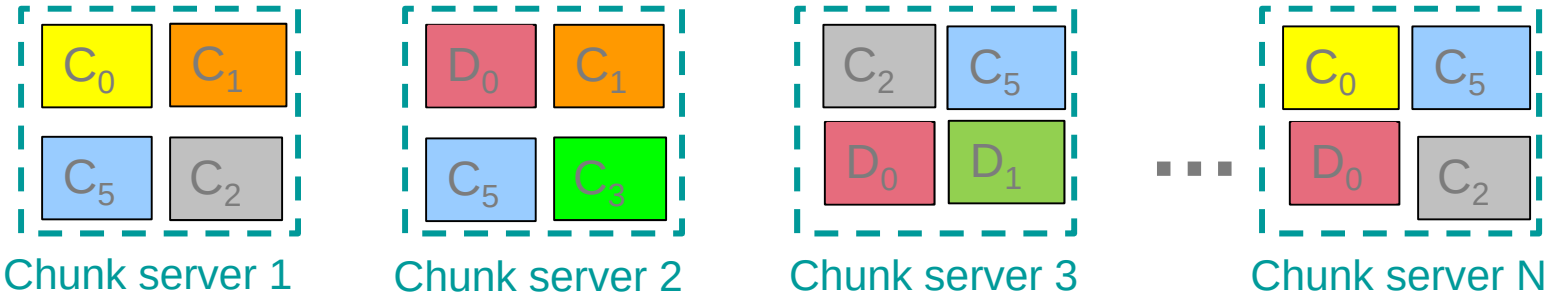
- Problem:
  - If nodes fail, how to store data persistently?
- Answer:
  - Distributed File System:
    - Provides global file namespace
    - Google GFS; Hadoop HDFS; Kosmix KFS
- Typical usage pattern
  - Huge files (100s of GB to TB)
  - Data is rarely updated in place
  - Reads and appends are common

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 Hadoop's HDFS
  - Stores metadata
  - Might be replicated
- Client library for file access:
  - Talks to master to find chunk servers
  - Connects directly to chunk servers to access data

Distributed File System

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



Bring computation directly to the 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

Warm up: Word Count (2)

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

Warm up: Word Count (3)

- Suppose we have a large corpus of documents
- Count occurrences of words:
  - **`words(docs/*) | sort | uniq -c`**
    - where **`words`** takes a file and outputs the words in it, one per a line
- 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 result

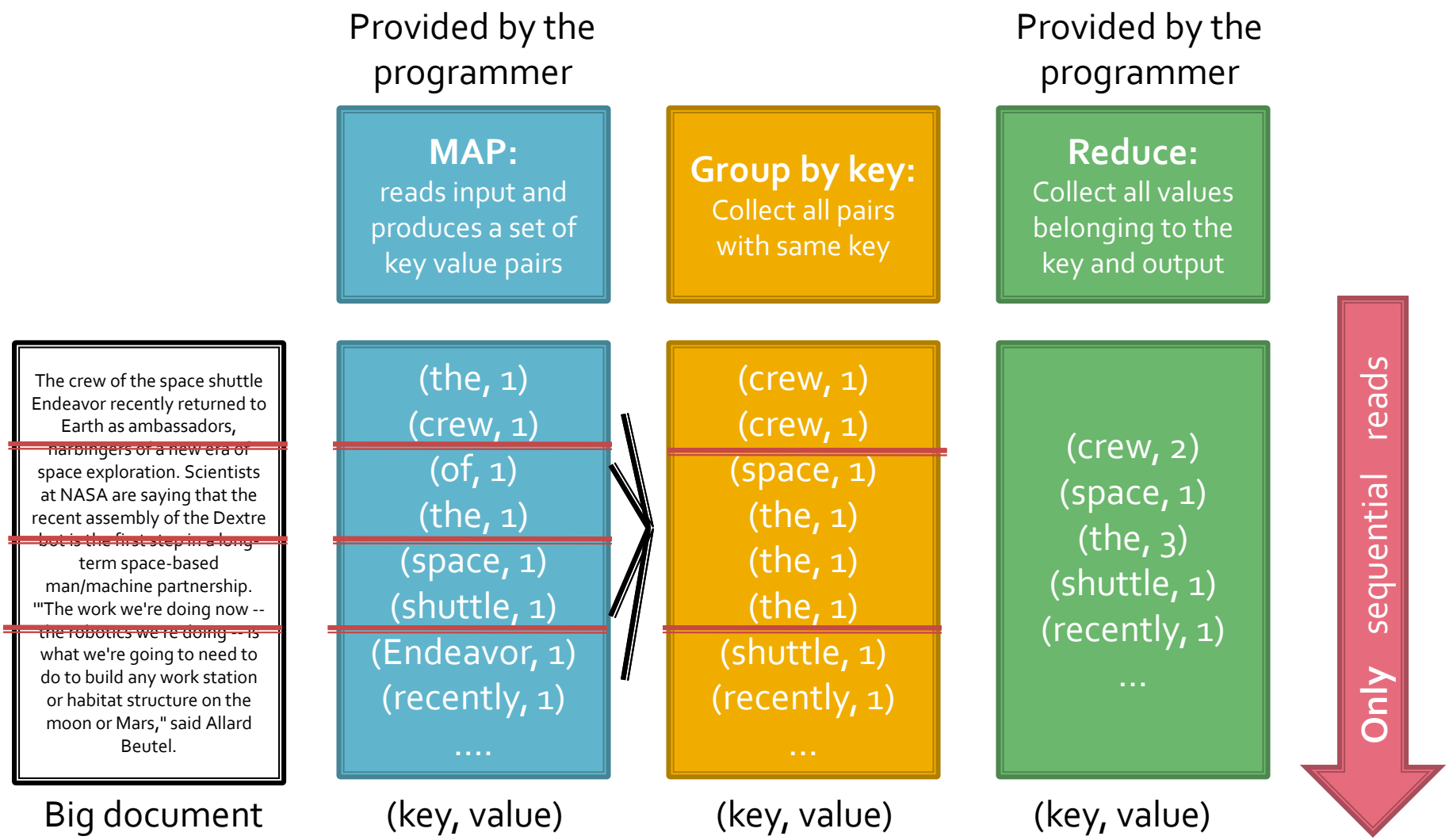
Outline stays the same, **map** and **reduce** change to fit the problem

More specifically

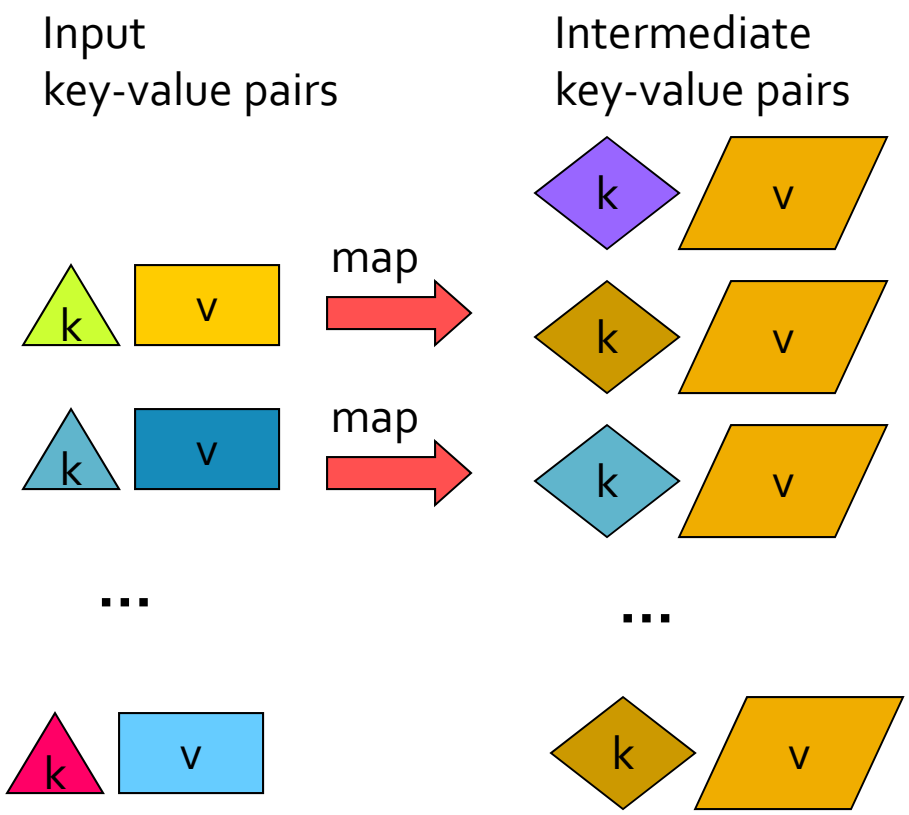
- Program specifies two primary methods:
  - Map(k,v) → <k', v'>*
  - Reduce(k', <v'>*) → <k', v''>*

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

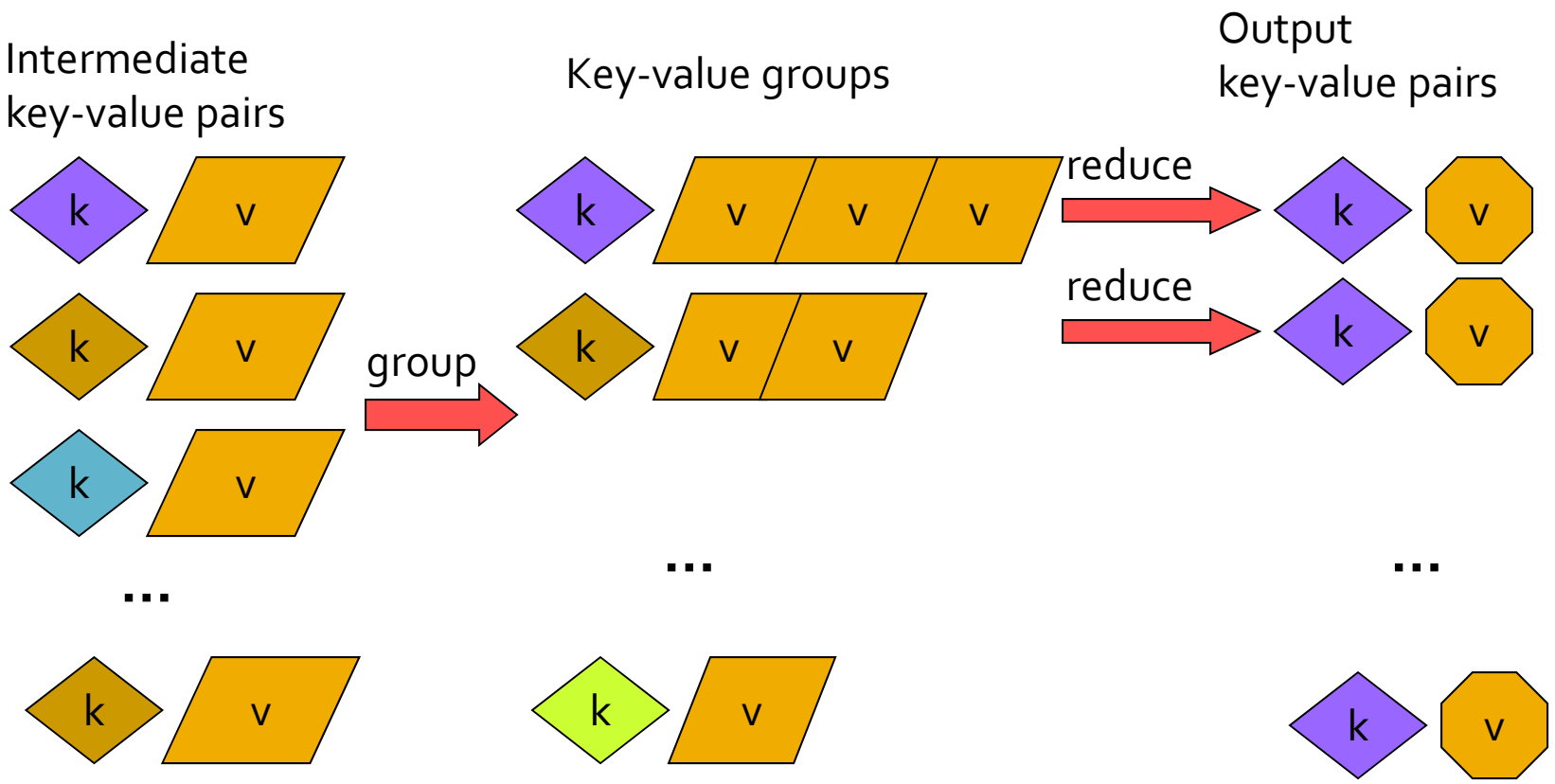
Map-Reduce: Word counting



MapReduce: The Map Step



MapReduce: The Reduce Step



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)
```

Map-Reduce: Environment

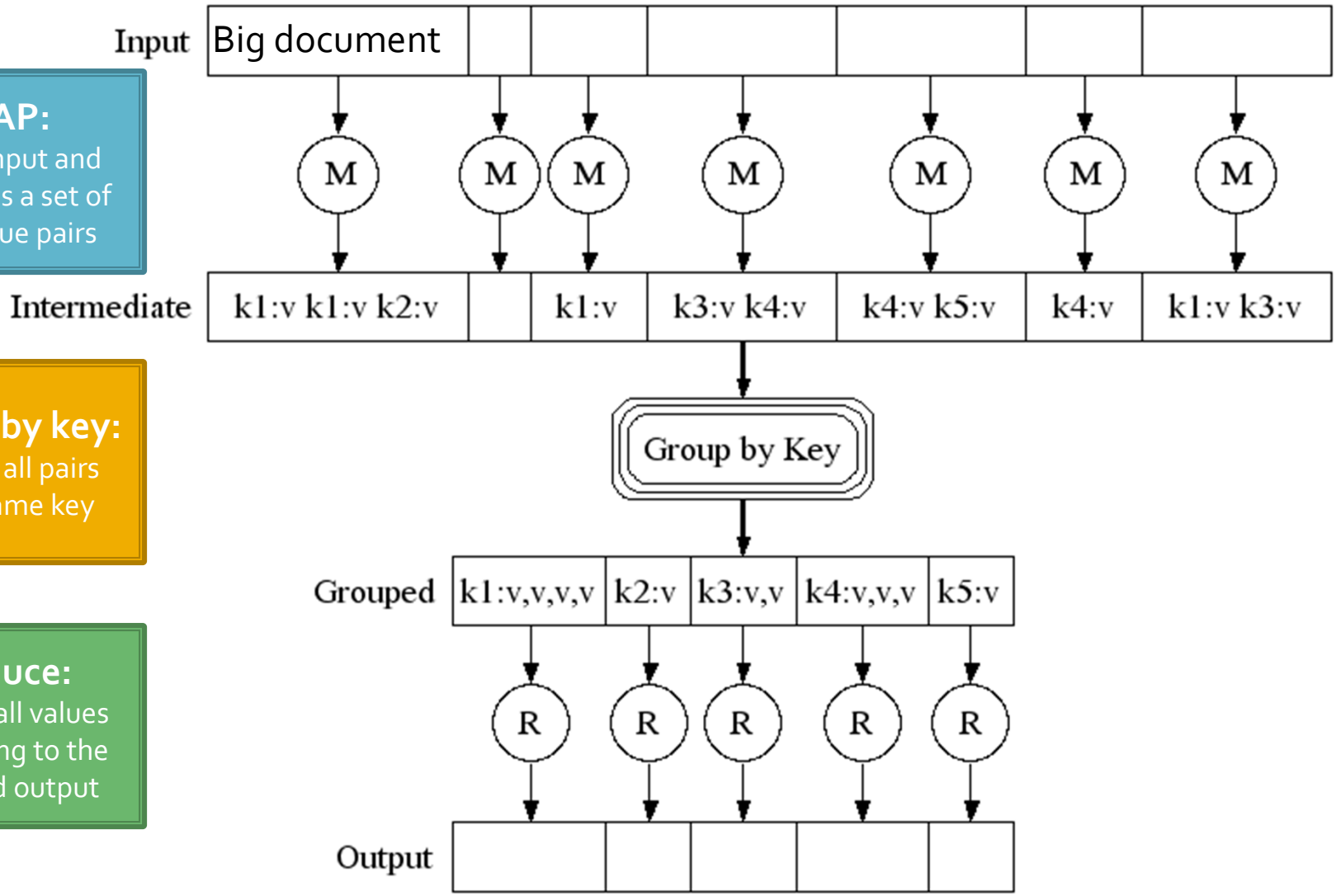
- Map-Reduce environment takes care of:
  - Partitioning the input data
  - Scheduling the program's execution across a set of machines
  - Handling machine failures
  - Managing required inter-machine communication
- Allows programmers without a PhD in parallel and distributed systems to use large distributed clusters

Map-Reduce: A diagram

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

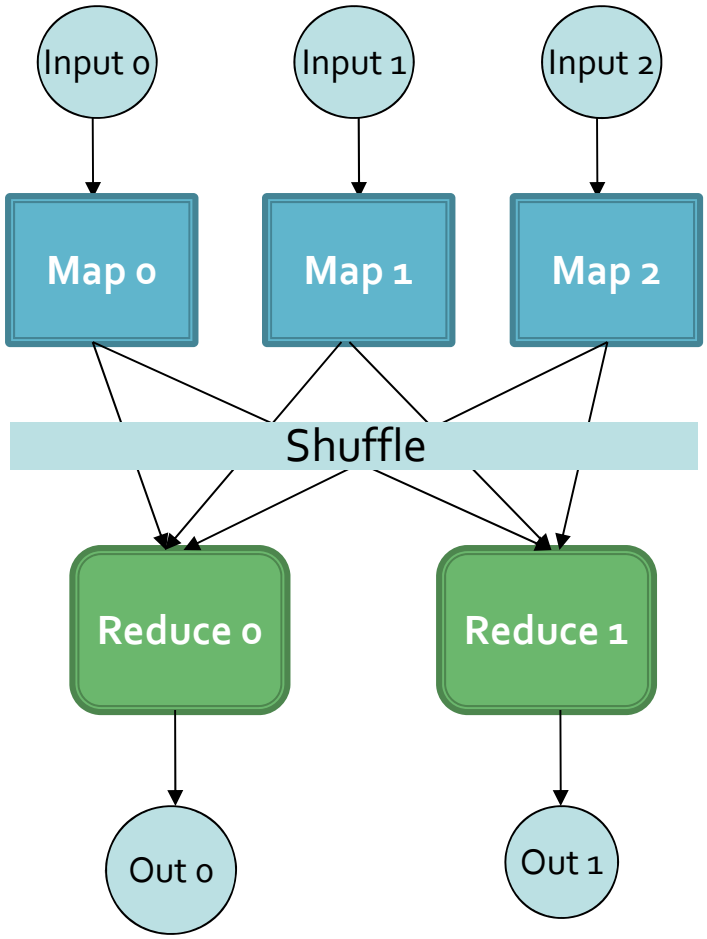
Group by key:
Collect all pairs with same key

Reduce:
Collect all values belonging to the key and output

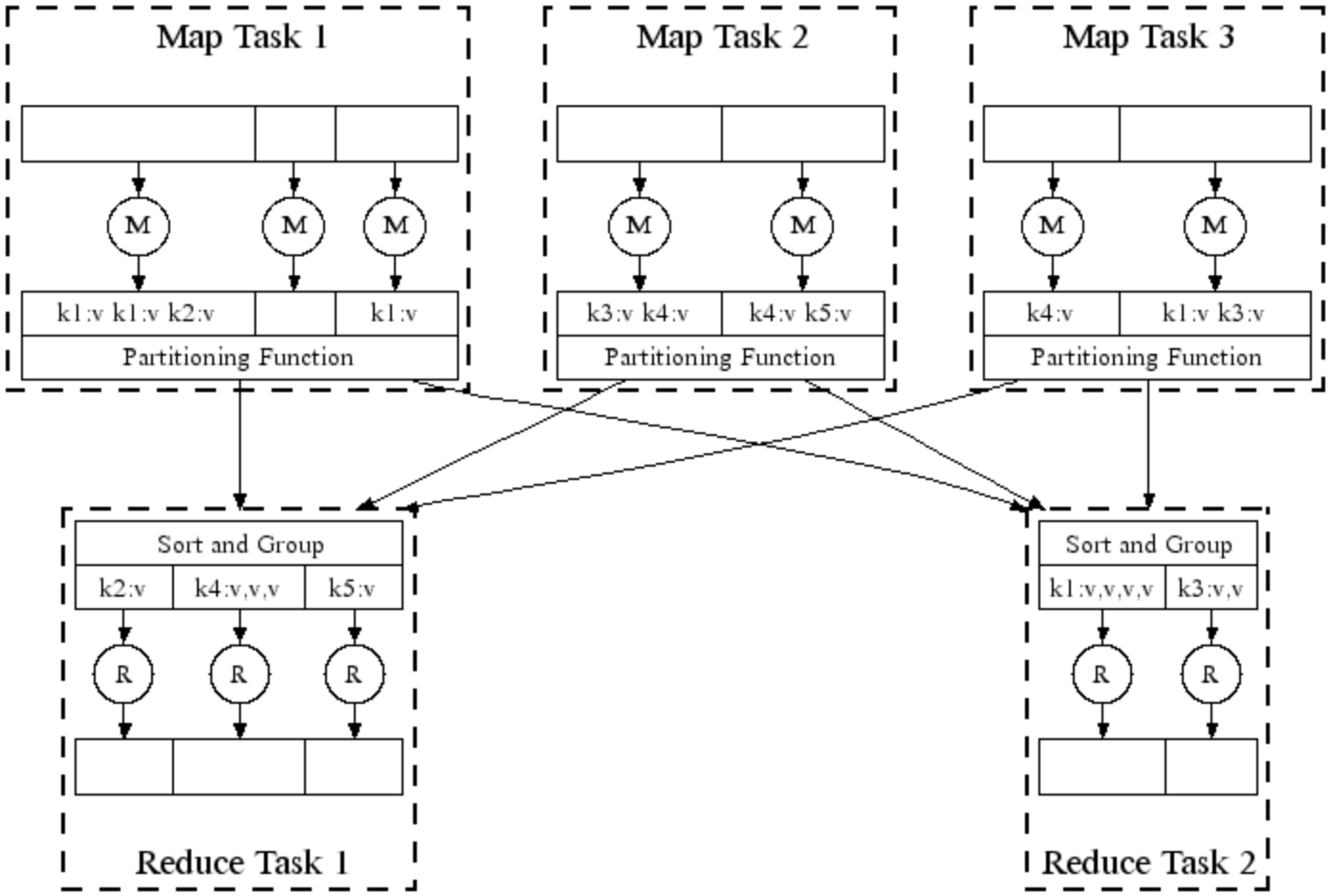


Map-Reduce

- 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



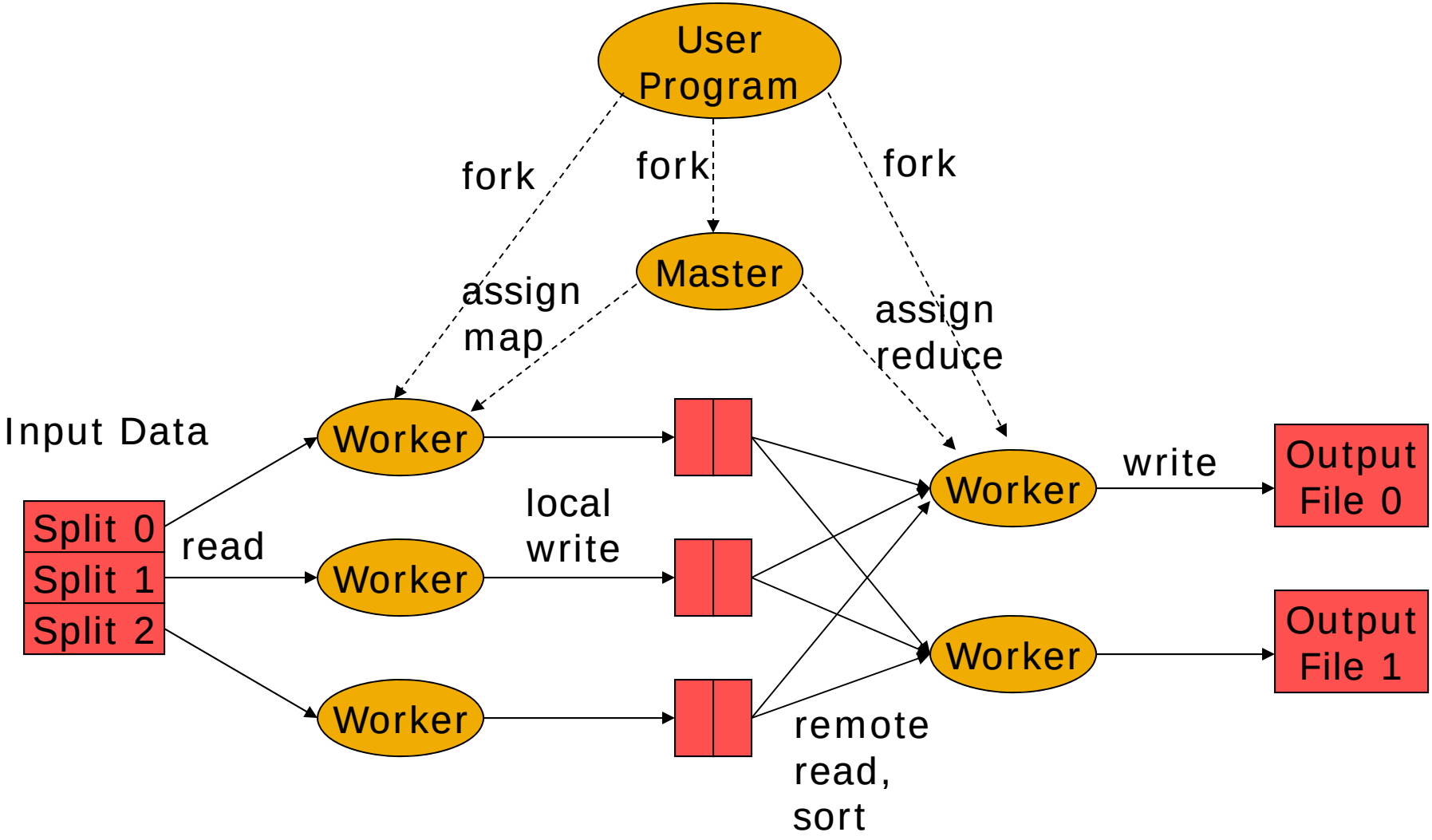
Implementation

- A program forks a *master* process and many *worker* processes.
- Input is partitioned into some number of *splits*.
- Worker processes are assigned either to perform Map on a split or Reduce for some set of intermediate keys.

Data flow

- Input and 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

Distributed Execution Overview



Responsibility of the Master

1. Assign Map and Reduce tasks to Workers.
2. Check that no Worker has died (because its processor failed).
3. Communicate results of Map to the Reduce tasks.

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

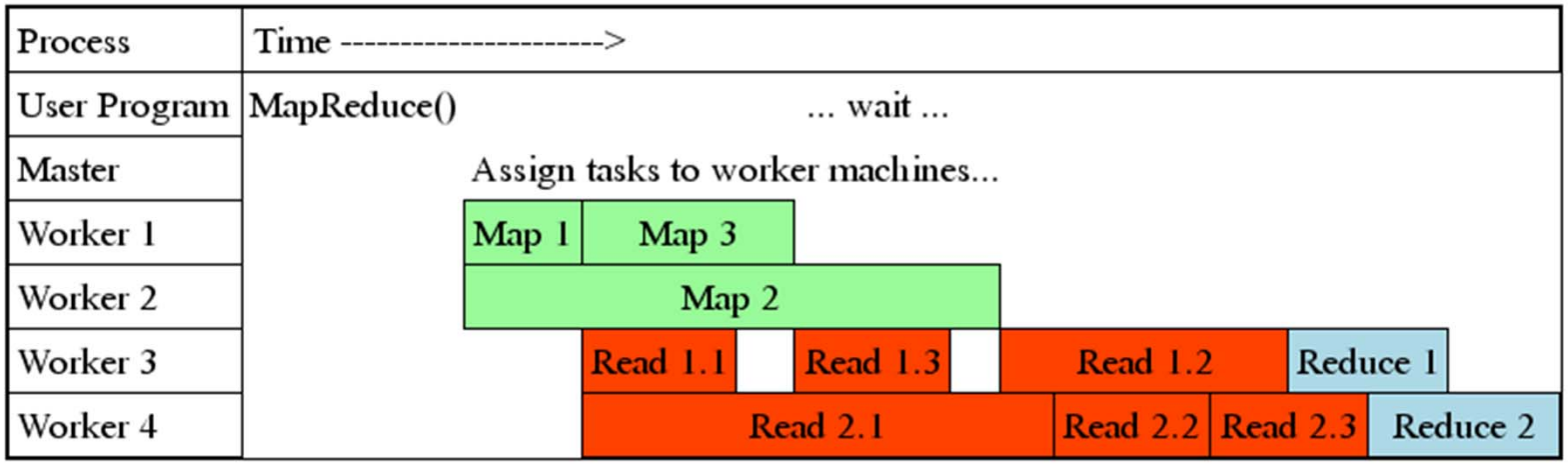
- Map worker failure
  - Map tasks completed or in-progress at worker are reset to idle
  - Reduce workers are notified when task is rescheduled on another worker
- Reduce worker failure
  - Only in-progress tasks are reset to idle
- Master failure
  - MapReduce task is aborted and client is notified

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

Task Granularity & Pipelining

- Fine granularity tasks: map tasks >> machines
  - Minimizes time for fault recovery
  - Can pipeline shuffling with map execution
  - Better dynamic load balancing



Problems suited for Map-Reduce

- Want to simulate disease spreading in a network
- **Input**
  - Each line: node id, virus parameters
- **Map**
  - Reads a line of input and simulate the virus
  - Output: triplets (node id, virus id, hit time)
- **Reduce**
  - Collect the node IDs and see which nodes are most vulnerable

Example 2: Language model

- Statistical machine translation:
  - Need to count number of times every 5-word sequence occurs in a large corpus of documents
- Easy with MapReduce:
  - Map:
    - Extract (5-word sequence, count) from document
  - Reduce:
    - Combine counts

Example 3: Host size

- Suppose we have a large web corpus
- 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
- Other examples:
  - Link analysis and graph processing
  - Machine Learning algorithms

Example: Join By Map-Reduce

- **Compute the natural join *R(A,B)* ⋈ *S(B,C)***
- *R* and *S* are each stored in files
- Tuples are pairs *(a,b)* or *(b,c)*

A	B
a_1	b_1
a_2	b_1
a_3	b_2
a_4	b_3

R

⋈

B	C
b_2	c_1
b_2	c_2
b_3	c_3

S

=

A	C
a_3	c_1
a_3	c_2
a_4	c_3

Map-Reduce Join

- **Use a hash function *h* from B-values to *1...k***
- **A Map process turns:**
  - Each input tuple *R(a,b)* into key-value pair *(b,(a,R))*
  - Each input tuple *S(b,c)* into *(b,(c,S))*
- **Map processes** send each key-value pair with key *b* to Reduce process *h(b)*
  - Hadoop does this automatically; just tell it what *k* is.
- Each **Reduce process** matches all the pairs *(b,(a,R))* with all *(b,(c,S))* and outputs *(a,b,c)*.

Cost Measures for Algorithms

- **In MapReduce we quantify the cost of an algorithm using**

1. *Communication cost* = total I/O of all processes
2. *Elapsed communication cost* = max of I/O along any path
3. (*Elapsed*) *computation cost* analogous, but count only running time of processes

Note that here the big-O notation is not the most useful (adding more machines is always an option)

Example: Cost Measures

- **For a map-reduce algorithm:**
  - **Communication cost =** input file size + 2 × (sum of the sizes of all files passed from Map processes to Reduce processes) + the sum of the output sizes of the Reduce processes.
  - **Elapsed communication cost** is the sum of the largest input + output for any map process, plus the same for any reduce process

What Cost Measures Mean

- Either the I/O (communication) or processing (computation) cost dominates
  - Ignore one or the other
- Total cost tells what you pay in rent from your friendly neighborhood cloud
- Elapsed cost is wall-clock time using parallelism

Cost of Map-Reduce Join

- **Total communication cost**
  = O(|R|+|S|+|R ⋈ S|)
- **Elapsed communication cost** = O(s)
  - We're going to pick ***k*** and the number of Map processes so that the I/O limit ***s*** is respected
  - We put a limit ***s*** on the amount of input or output that any one process can have. ***s* could be:**
    - What fits in main memory
    - What fits on local disk
- With proper indexes, computation cost is linear in the input + output size
  - So computation cost is like comm. cost

Implementations

- Google
  - Not available outside Google
- Hadoop
  - An open-source implementation in Java
  - Uses HDFS for stable storage
  - Download: http://lucene.apache.org/hadoop/
- Aster Data
  - Cluster-optimized SQL Database that also implements MapReduce

Cloud Computing

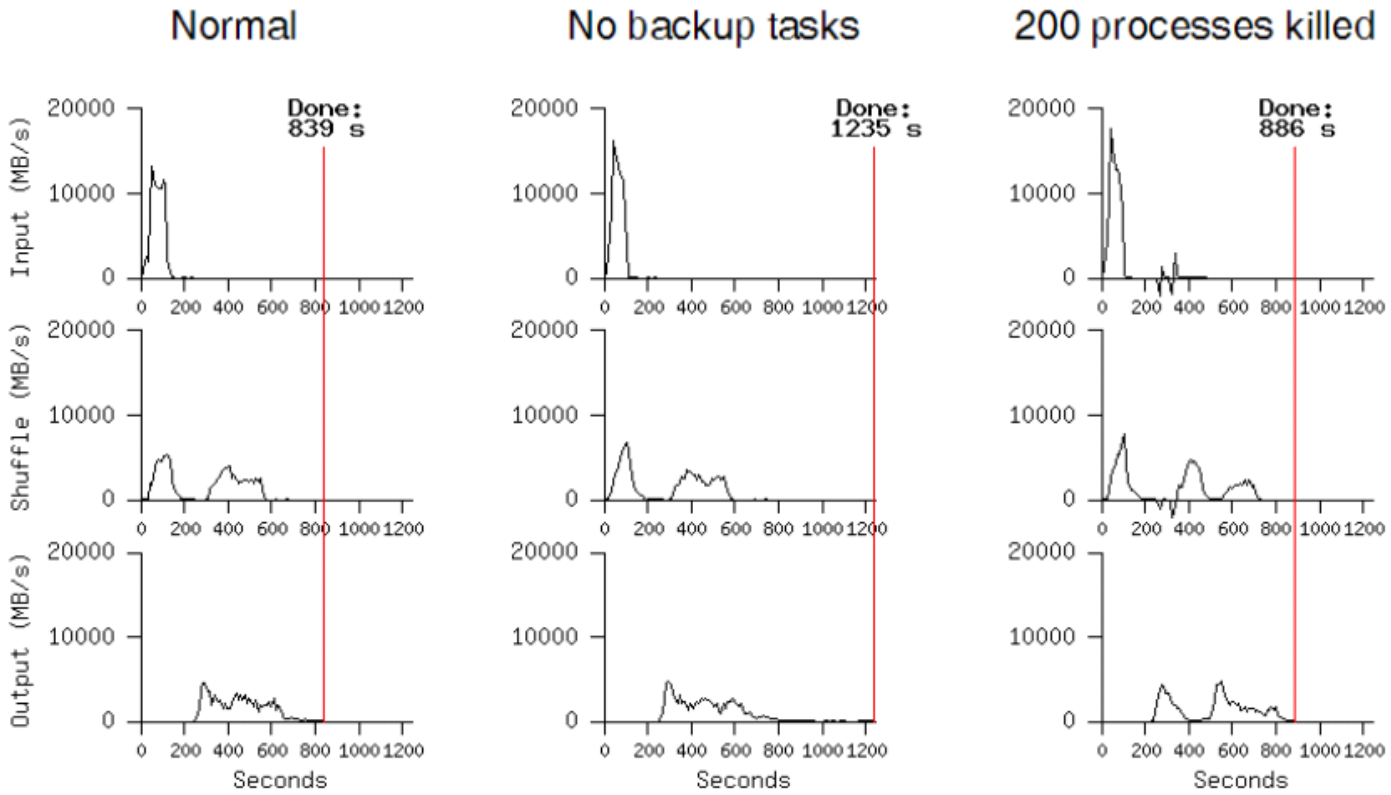
- Ability to rent computing by the hour
  - Additional services e.g., persistent storage
- Amazon's "Elastic Compute Cloud" (EC2)
- Aster Data and Hadoop can both be run on EC2

Refinement: Backup tasks

- Problem:
  - Slow workers significantly lengthen the job completion time:
    - Other jobs on the machine
    - Bad disks
    - Weird things
- Solution:
  - Near end of phase, spawn backup copies of tasks
    - Whichever one finishes first "wins"
- Effect:
  - Dramatically shortens job completion time

Refinements: Backup tasks

- Backup tasks reduce job time
- System deals with failures



Refinements: 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 the reduce function
- Works only if reduce function is commutative and associative

Refinements: Partition Function

- Inputs to map tasks are created by contiguous splits of input file
- Reduce needs to ensure that records with the same intermediate key end up at the same worker
- System uses a default partition function:
  - 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

Google File System (GFS)

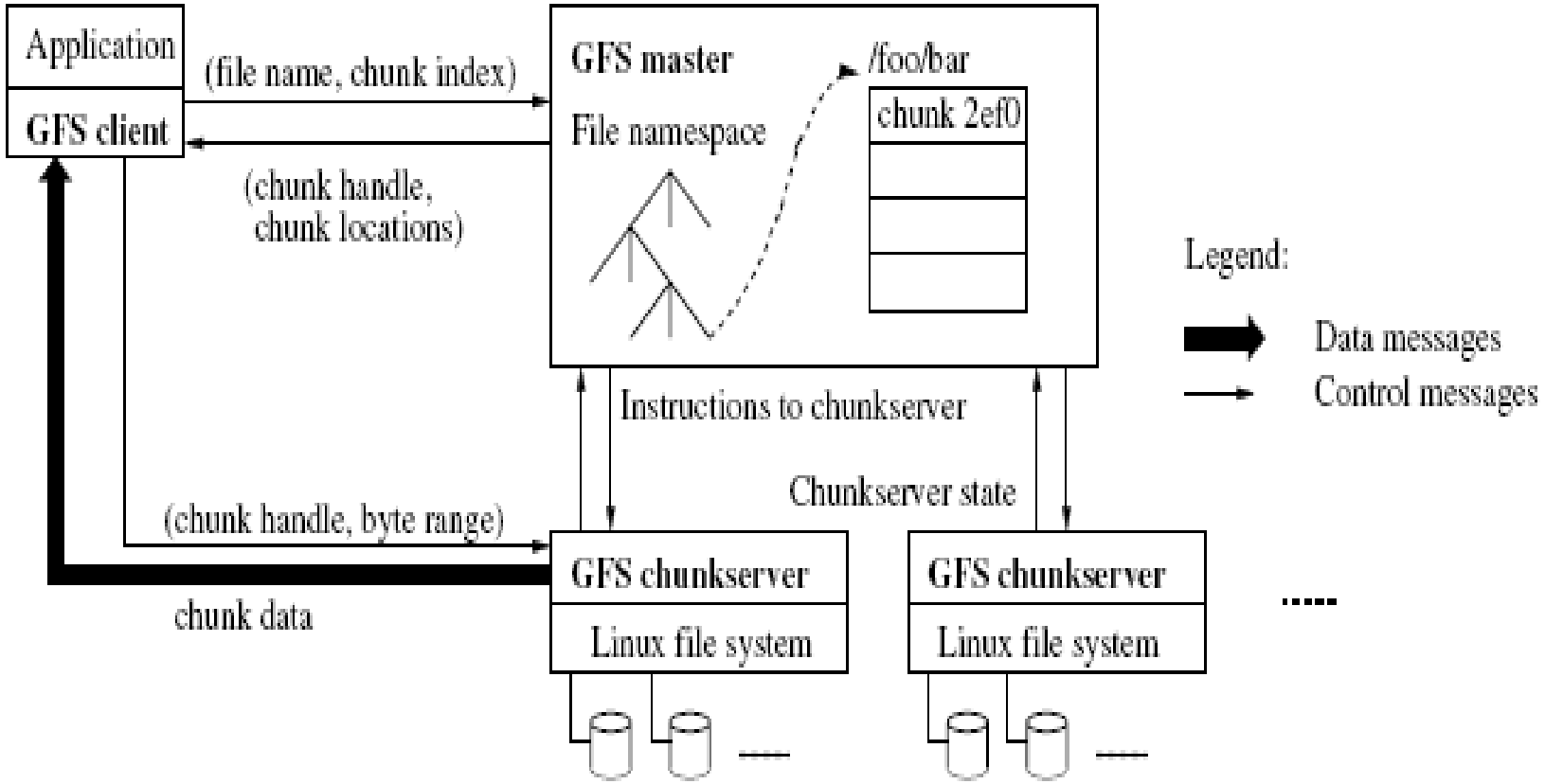
- Design Assumptions
  - Component failures are norm rather than the exception
    - Inexpensive commodity components
  - Files are huge by traditional standards
    - Multi-GB files are common
  - High sustained bandwidth is more important than low latency

"The Google File System", SOSP 2003

GFS (Design Assumptions)

- Most files are mutated by appending new data rather than overwriting existing data
  - Once written, the files are only read and often only sequentially
  - Two types of reads – large streaming reads and small random reads
- Efficient implementation
  - Allow multiple clients read the same file
  - Atomicity with minimal synchronization overhead is essential

GPS Architecture



GFS Design Parameters

- Client and Chunk server can run on the same machine
- Files divided into fixed-sized chunks (64 MB)
  - Chunk handle
  - What are the tradeoffs of the 64 MB chunk size?
- Chunkserver stores chunks on local disks as Linux files
  - Chunks are replicated on multiple chunk servers
- Master maintains all file system metadata – stateful (namespace, access control, mapping file to chunks, current location)

GFS Design Parameters

- Client's code implements GFS API and communicates with master and chunk servers to read and write
- Client communicates with master for metadata information
- Client communicates with chunk servers for data over TCP/IP
- No data caching!!!

Performance Results & Experience

Using 1,800 machines:

- MR_Grep scanned 1 terabyte in 100 seconds
- MR_Sort sorted 1 terabyte of 100 byte records in 14 minutes

Rewrote Google's production indexing system

- a sequence of ~~7~~, ~~10~~, ~~14~~, ~~17~~, ~~21~~, 24 MapReductions
- simpler
- more robust
- faster
- more scalable

Google

Reading

- Jeffrey Dean and Sanjay Ghemawat,

  MapReduce: Simplified Data Processing on Large Clusters
  http://labs.google.com/papers/mapreduce.html

- Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, **The Google File System**
  http://labs.google.com/papers/gfs.html

Resources

- Hadoop Wiki
  - Introduction
    - http://wiki.apache.org/lucene-hadoop/
  - Getting Started
    - http://wiki.apache.org/lucene-hadoop/GettingStartedWithHadoop
  - Map/Reduce Overview
    - http://wiki.apache.org/lucene-hadoop/HadoopMapReduce
    - http://wiki.apache.org/lucene-hadoop/HadoopMapRedClasses
  - Eclipse Environment
    - http://wiki.apache.org/lucene-hadoop/EclipseEnvironment
- Javadoc
  - http://lucene.apache.org/hadoop/docs/api/

Resources

- **Releases from Apache download mirrors**
  - http://www.apache.org/dyn/closer.cgi/lucene/hadoop/
- **Nightly builds of source**
  - http://people.apache.org/dist/lucene/hadoop/nightly/
- **Source code from subversion**
  - http://lucene.apache.org/hadoop/version_control.html

Further reading

- Programming model inspired by functional language primitives
- Partitioning/shuffling similar to many large-scale sorting systems
  - NOW-Sort ['97]
- Re-execution for fault tolerance
  - BAD-FS ['04] and TACC ['97]
- Locality optimization has parallels with Active Disks/Diamond work
  - Active Disks ['01], Diamond ['04]
- Backup tasks similar to Eager Scheduling in Charlotte system
  - Charlotte ['96]
- Dynamic load balancing solves similar problem as River's distributed queues
  - River ['99]