

# Introduction

*CS6450: Distributed Systems*

Lecture 1

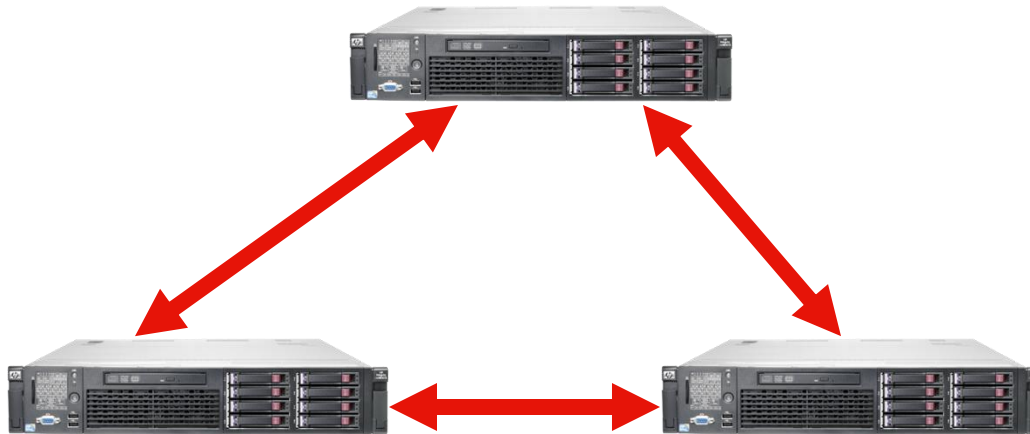
Ryan Stutsman

Some material taken/derived from Princeton COS-418 materials created by Michael Freedman, Kyle Jamieson, and Wyatt Lloyd at Princeton University.

Licensed for use under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License.

Some material taken/derived from MIT 6.824 by Robert Morris, Franz Kaashoek, and Nickolai Zeldovich.

# What is a Distributed System?



- Multiple computers
- Connected by a network
- Doing something together

# Why Distribute Systems?

## Why not just use one computer?

- Connect physically separate entities, Sharing
- Performance & Capacity: parallel CPU/mem/disk/net
- Fault-tolerance
- Security via physical isolation
- But, complex; hard to debug
  - New classes of problems, e.g. partial failure (did server accept my e-mail?)

# Historical Context

## 1980s: Local Area Networks

- DNS, email, NFS

## Late 1990s: Peer-to-Peer

- Napster, Gnutella, Bittorrent

## Late 1990s, 2000s: Data Centers

- Web search, web mail, online shopping
- Large-scale processing, storage, distributed databases

## 2010s: Cloud Computing

- Virtual machines, Cloud Databases/Storage, Serverless

# Today

- Web Search (e.g., Google, Bing)
- Shopping (e.g., Amazon, Walmart)
- File Sync (e.g., Dropbox, Google Drive, OneDrive)
- Social Networks (e.g., Facebook, Twitter, TikTok)
- Music (e.g., Spotify, Apple Music)
- Ride Sharing (e.g., Uber, Lyft)
- Video (e.g., Youtube, Netflix)
- Online gaming (e.g., Fortnite, DOTA2, Roblox)
- ...



**Backrub (Google) 1997**



# Google 2012

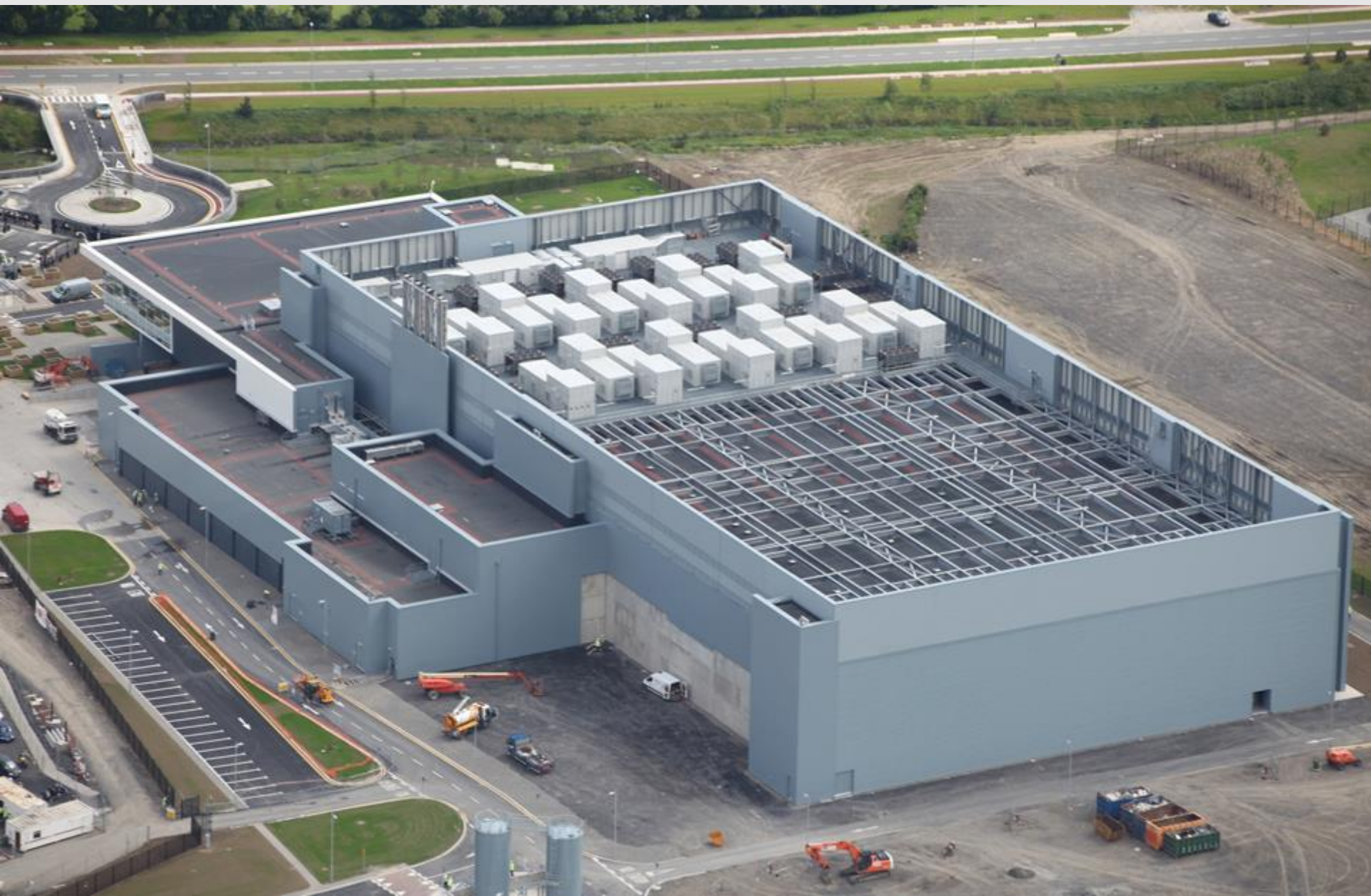




Google ~2012



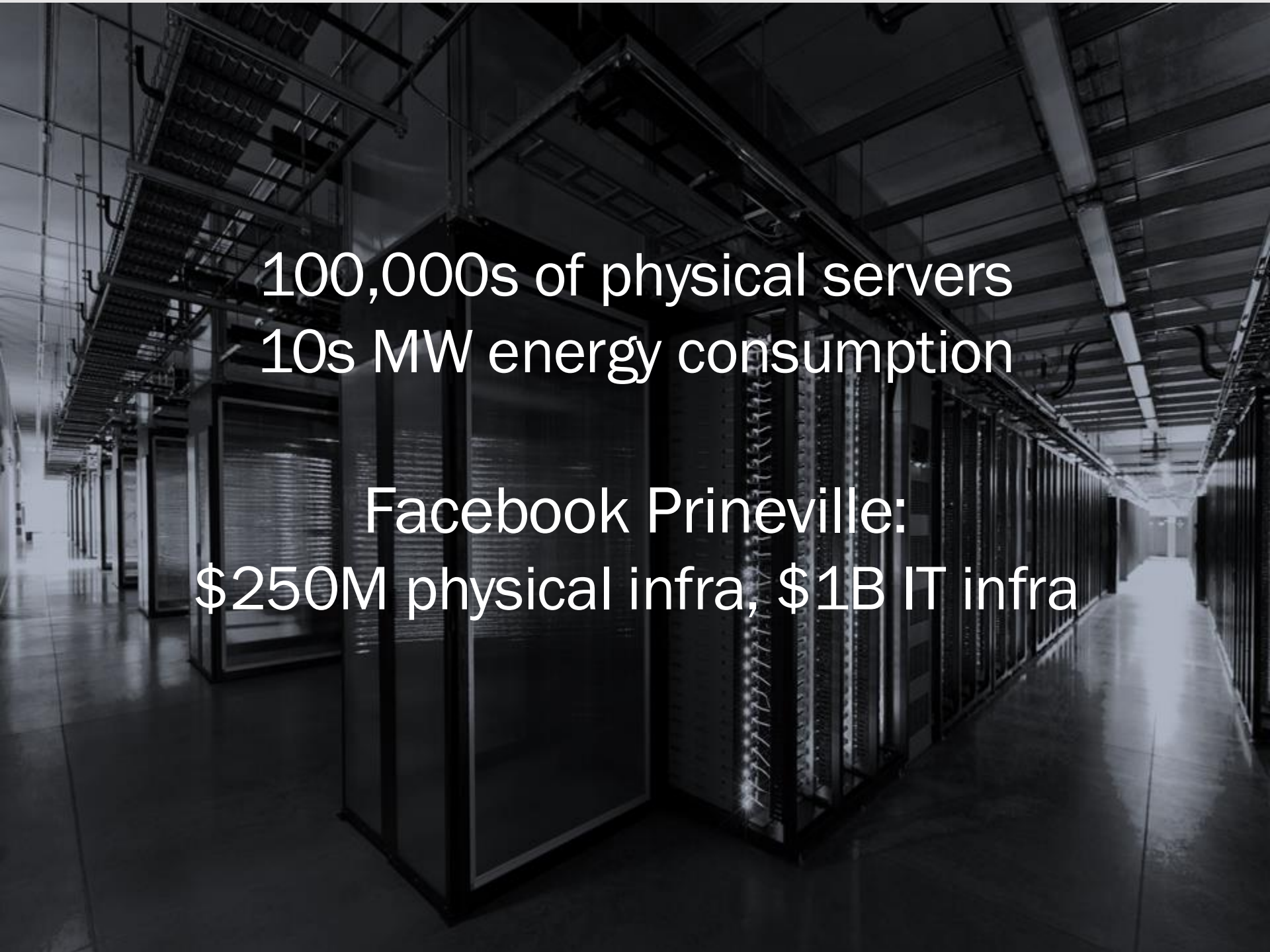




**Microsoft ~2013**







100,000s of physical servers  
10s MW energy consumption

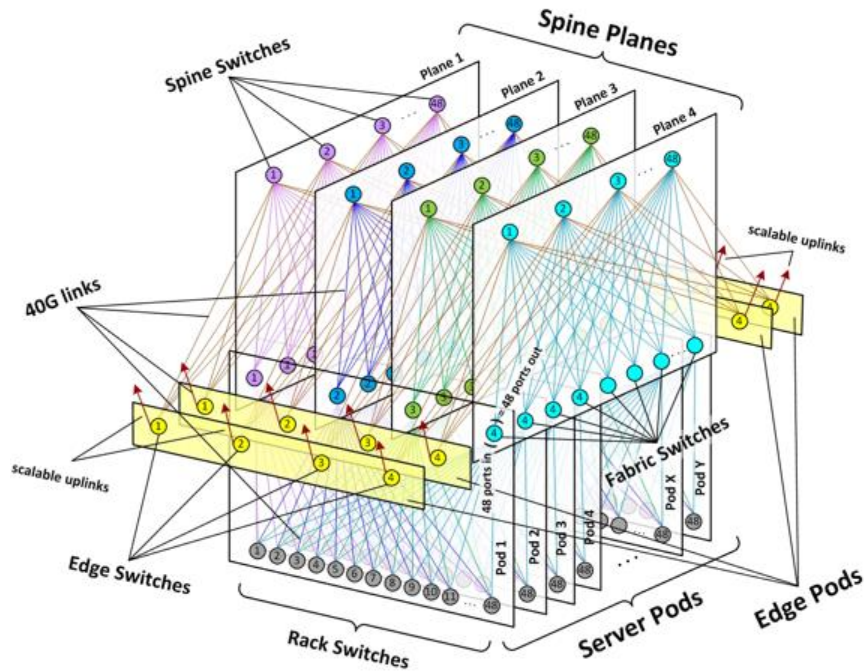
Facebook Prineville:  
\$250M physical infra, \$1B IT infra





Source:  
Deseret News,  
Data Center Frontier

# Everything changes at scale



Machine failures, switch failures (TOR, spine), partitions, flapping routes...

<https://engineering.fb.com/production-engineering/introducing-data-center-fabric-the-next-generation-facebook-data-center-network/>

# Challenges, Why is it Hard?

- Many concurrent parts
- Must deal with (continuous?) partial failure
- Cannot distinguish network failures from crashes



Complexity

- Hard to scale performance
  - Unless embarrassingly parallel



# Why Take this Class?

- Interesting
  - hard problems, (simple but) non-obvious solutions
- Ideas behind real systems
  - driven by the rise of web; you'll work on these large-scale systems in the field
  - Don't just know to use Zookeeper or etcd, understand how it works
- Active research area
  - lots of progress + big unsolved problems
- Hands-on
  - you'll build interesting systems in the labs

# Research results matter: NoSQL

## Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels

Amazon.com

### ABSTRACT

Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in the world; even the slightest outage has significant financial consequences and impacts customer trust. The Amazon.com platform, which provides services for many web sites worldwide, is implemented on top of an infrastructure of tens of thousands of servers and network components located in many datacenters around the world. At this scale, small and large components fail continuously and the way persistent state is managed in the face of these failures drives the reliability and scalability of the software systems.

This paper presents the design and implementation of Dynamo, a highly available key-value storage system that some of Amazon's core services use to provide an "always-on" experience. To achieve this level of availability, Dynamo sacrifices consistency under certain failure scenarios. It makes extensive use of object versioning and application-assisted conflict resolution in a manner that provides a novel interface for developers to use.

### Categories and Subject Descriptors

D.4.2 [Operating Systems]: Storage Management; D.4.5

[Operating Systems]: Reliability; D.4.2 [Operating Systems]: Performance;

### General Terms

Algorithms, Management, Measurement, Performance, Design, Reliability.

One of the lessons our organization has learned from operating Amazon's platform is that the reliability and scalability of a system is dependent on how its application state is managed. Amazon uses a highly decentralized, loosely coupled, service oriented architecture consisting of hundreds of services. In this environment there is a particular need for storage technologies that are always available. For example, customers should be able to view and add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornados. Therefore, the service responsible for managing shopping carts requires that it can always write to and read from its data store, and that its data needs to be available across multiple data centers.

Dealing with failures in an infrastructure comprised of millions of components is our standard mode of operation; there are always a small but significant number of server and network components that are failing at any given time. As such Amazon's software systems need to be constructed in a manner that treats failure handling as the normal case without impacting availability or performance.

To meet the reliability and scaling needs, Amazon has developed a number of storage technologies, of which the Amazon Simple Storage Service (also available outside of Amazon and known as Amazon S3), is probably the best known. This paper presents the design and implementation of Dynamo, another highly available and scalable distributed data store built for Amazon's platform. Dynamo is used to manage the state of services that have very high reliability requirements and need tight control over the

ers may be parti-  
tain connected to  
is a central goal  
e communication  
esign copes with

re accommodated  
. Replication is  
e reachable from  
ps. Weak consist-  
-viding one copy  
less a quorum of  
hat they wish to  
ity in partitioned  
model in which  
out the need for  
computer eventu-  
ctly or indirectly,

goal in designing  
st replicated data  
applications. We  
may read weakly  
ons may conflict  
over, applications  
of conflicts since  
plication.

t for application-  
ous systems, such  
value of semantic  
ries, and several  
and database cons-  
s work by letting

Distributed

David Kar

### Abstract

We describe a family  
that can be used to de-  
in the network. Our p  
very large networks  
hot spots can be sever  
to have complete inf  
network. The proto  
work protocols such  
The protocols work v  
ing resources, and sc

Our caching prot  
that we call consiste  
hash function is one  
function changes. I  
hash functions, we a  
not require users to h  
network. We believe  
prove to be useful in  
servers and/or quoru

### 1 Introduction

In this paper, we de  
works that can be us  
of "hot spots". Hot  
wish to simultaneous  
is not provisioned to  
service may be degr

# Research results matter: Paxos

The Part-Ti

Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems

Brian M. Oki  
Robert M. Lister

## The Chubby lock service for loosely-coupled distributed systems

Mike Burrows, *Google Inc.*

### Abstract

We describe our experiences with the Chubby lock service, which is intended to provide coarse-grained locking as well as reliable (though low-volume) storage for a loosely-coupled distributed system. Chubby provides an interface much like a distributed file system with advisory locks, but the design emphasis is on availability and reliability, as opposed to high performance. Many instances of the service have been used for over a year, with several of them each handling a few tens of thousands of clients concurrently. The paper describes the initial design and expected use, compares it with actual use, and explains how the design had to be modified to

example, the Google File System [7] uses a Chubby lock to appoint a GFS master server, and Bigtable [3] uses Chubby in several ways: to elect a master, to allow the master to discover the servers it controls, and to permit clients to find the master. In addition, both GFS and Bigtable use Chubby as a well-known and available location to store a small amount of meta-data; in effect they use Chubby as the root of their distributed data structures. Some services use locks to partition work (at a coarse grain) between several servers.

Before Chubby was deployed, most distributed systems at Google used *ad hoc* methods for primary election (when work could be duplicated without harm), or

ing of nodes connected by  
dependent computers that  
nding messages over the  
work may fail, we assume  
odes can crash, but we  
The network may lose,  
r messages out of order.  
tion into subnetworks that  
r. We assume that nodes  
partitions are eventually

el of computation in which  
each of which resides at  
le contains within it both  
the objects; modules can  
r state intact. No other  
another module directly.  
es that can be used to  
te by means of *remote*  
ils are called *clients*; the

n our method. Ideally,  
t concern for availability in  
that supports our model of  
n then uses our technique



# Research results matter: MapReduce

## MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

### Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers

given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a *map* operation to each logical "record" in our input in order to compute a set of intermediate key/value pairs, and then applying a *reduce* operation to all the values that shared the same key, in order to combine the derived data appropriately. Our use of a functional model with user-specified map and reduce operations allows us to paral-



# Course Content

# Focus: Main Concepts

- Fault Tolerance
  - Availability → Replication
  - Recoverability → Logging, Transactions
- Consistency
  - Does `get(k)` return value of last `put(k, v)`?
  - What is the contract with crashes, message loss, concurrency?
  - Different answers with different tradeoffs
- Performance
  - Throughput
  - Latency, Tail latency
- Scaling & Sharding



# Rough Progression, More Detail

- Communication & Clocks
  - Networking, Concurrency, and RPC
  - Time & Logical Clocks
  - Vector Clocks
- Scaling/Sharding
  - Decentralized Systems
  - Distributed Hash Tables
- Replicated State Machines
  - Primary/Backup
  - View Change & Consensus
  - Paxos & Raft
- Consistency
  - Linearizability & Sequential Consistency
  - Causal & Eventual consistency
- Transactions
  - Serializability
  - Atomic Commitment and 2PC
  - Concurrency Control: 2PL, OCC, MVCC
- Performance
  - Modeling & Measurement
- Byzantine Fault Tolerance
  - PBFT
  - Bitcoin & Nakamoto Consensus

# Big Picture Course Goals

- Understand mechanisms that underlie modern distributed systems (RPC, consensus, leases, concurrency control, etc.)
- Learn about some of the most influential works in distributed systems
- Learn how to read distributed systems papers
- Learn how to manage writing highly concurrent and nondeterministic code
  - In my opinion, much harder than "just" parallel programming.
- Get a sense of how massive scale systems "fit" together


# Course Organization



# Calendar

- Jumping off point for everything in class
- Readings, assignments, due dates
  - Lecture
    - WEB 1250 (Tue & Thu 15:40 to 17:00)
  - Office Hours
    - Ryan: Thu 12:00 to 14:00 (In Person in MEB 3436; I will monitor [MS Teams Office Hours Channel](#) ➡ during office hours as well; just @ message me there and I will reply)
    - Chloe: Mon 16:40 to 17:40 and Tue 17:30 to 18:30 (In Person in TBD; I will monitor [MS Teams Office Hours Channel](#) ➡ during office hours as well; just @ message me there and I will reply)

## Schedule

Date	Lecture	
Tue 8/20	Introduction & MapReduce  <a href="#">Slides</a> ↓	<a href="#">Lab Setup</a> , <a href="#">Homework 1</a> , and <a href="#">Lab 2</a> Released (we won't do Lab 1)  Optional Reading: <a href="#">MapReduce paper</a> ➡ Coulouris §21.6.1
		<b>REQUIRED:</b> Do the <a href="#">Online Go Tutorial</a> ➡ before class

# Getting Help

- Office Hours
  - Chloe: Mon 16:40 – 17:40 and Tue 17:30 – 18:30
  - Ryan: Thu 12:00-14:00
  - Both via MS Teams and in-person, see syllabus for locations
- MS Teams
  - Good place to ask and answer questions
    - About labs; #lab2, #lab3a, #lab3b, ...
    - About material from lecture: #general
  - Can send private messages to instructors
    - Some delay in responses except during office hours

# Course Format: Lectures

- In-Person Lecture: Tue/Thu 15:40 to 17:00
  - Slides available on course website
  - Will **try** to have them up the night before
  - Not recorded
- Occasionally required reading before class (listed in schedule/table at top of syllabus)
  - Some pre-lecture quizzes or in-class quizzes over required readings
  - Expect more such quizzes if attendance drops



# Final Course Grade

- Labs/Programming Assignments (34% total)
  - **THREE** free late days, TA automatically applies them
  - Otherwise, late turnins are graded at 90, 80, 70, 60% multiplier for each day late after that
- Homeworks (33% total)
- Reading Comprehension/Quizzes (33% total)
  - No single quiz worth more than 3% of final grade

# Course Format: “Labs” (34%)

- Three labs (one of which is broken into 3 parts)
- Linearizable Key-Value Store
- Consensus State Machine Replication Library
- Fault-tolerant Consensus-Replicated Key-Value Store
- Get started early on labs, **especially Lab 3**
- Require comfort with concurrency that takes awhile to acquire
- All test cases that we use are given to you

# Labs are a LOT of work

- Lab 2 is easier, but still hard
- You'll need the full time for some parts
- Think about how to structure your code/threading
- Debugging can dominate (5x dev time or more)
- **Read** and understand given test cases for clues
  - Test-driven development works well here
- Assignment 4 uses your solution for Assignment 3



# Course Format: Homework (33%)

- Homeworks/problem sets via online Canvas "quizzes"
  - About 4 to 8
  - Standard Collaboration policy: you can discuss together but don't copy off each other

# Course Format: Quizzes (33%)

- Cover prior lecture or assigned reading
  - Assigned reading will sometimes be research papers; in those cases often the “quiz” will be provided offline before class
  - If offline, standard collaboration policy applies
  - Else, if live in class, then no collaboration

# Books?

- Slides/lecture notes
- Papers (required or optional) also serve as reference for many topics that aren't covered directly by a text
- “Distributed Systems”, Coulouris 5e book is good but expensive
- “Distributed Systems”, v3.01 by van Steen (free online)
- Will try to keep some alternate readings up-to-date from both books

# Policies: Write Your Own Code & Solutions

Everyone needs to read the [SoC Policy on Academic Misconduct](#).

You must not make your code public (on github or by any other means). The labs we use are the same ones used in MIT's 6.5840. These labs were **painstakingly** developed by the MIT faculty, and are key in their course. **IT IS CRITICAL THAT SOLUTIONS TO THEIR LABS ARE NOT MADE PUBLIC IN ANY WAY.**

Working with others on assignments is a good way to learn the material and we encourage it. However, there are limits to the degree of cooperation that we will permit.

When working on programming assignments, you must work only with others whose understanding of the material is approximately equal to yours. In this situation, working together to find a good approach for solving a programming problem is cooperation; listening while someone dictates a solution is cheating. You must limit collaboration to a high-level discussion of solution strategies, and stop short of actually writing down a group answer. Anything that you hand in, whether it is a written problem or a computer program, must be written in your own words. If you base your solution on any other written solution, you are cheating.

Never look at another student's code or share your code with any other student. *We do not distinguish between cheaters who copy other's work and cheaters who allow their work to be copied.* If you cheat, you will be given an E in the course and referred to the University Student Behavior Committee. Clearly, any attempt to subvert the ordinary grading process constitutes cheating.

**If you have any questions about what constitutes cheating, please ask first.**



# Gen AI Tools

- Use of Gen AI tools use is allowed, but with specific and important restrictions.
- Restriction #1: Indicate Your Use of the Models
  - Must indicate use of models in code or in assignment response.
  - With Co-pilot, indicate its use at the top of every affected file.
- Restriction #2: Do Not Provide Solutions to the Models for Training
  - IT IS CRITICAL THAT SOLUTIONS TO THEIR LABS ARE NOT MADE PUBLIC IN ANY WAY.
  - You MUST NOT use any (partial) solution as input to a Gen AI tool that might use that input for training.
  - Github Copilot. May use Copilot PROVIDED that training on your code is disabled, otherwise Copilot will learn your solutions.
  - ChatGPT, Gemini, etc. Do not paste any partial solution into these systems, since they train on input by default.
  - Please realize: these models already have been trained on others' past solutions to these assignments. You are responsible for what you learn.
- Restriction #3: No Wholesale Copying of Assignment-specific Solutions
  - You must not ask for specific or complete solutions and you must not copy them.
  - Asking ChatGPT to even just show you an implementation of "func (rf \*Raft) AppendEntriesRPC" is Academic Misconduct. Copying from such a solution is Academic Misconduct as well.
  - Asking ChatGPT to write a method "func partition(key string, n\_partitions int) int" is not Academic Misconduct as long as you indicate that you used code or ideas you gained from ChatGPT in the code.
- In Classwork Other Than Code
  - Similar restrictions apply to homeworks and quizzes. Indicate when you've used Gen AI in your work, don't put any part of a solution into them as input, and don't get wholesale answers or copy them.
- Don't paste AI generated text as solutions. It's Academic Misconduct.

# TODOs to Start on Now

- Online Go Tutorial (by class Thu 8/22)
- Lab Setup
- Homework 1 (by Tue 8/27)
- Lab 2, once done with other stuff, go ahead and get started on this

# Case Study: MapReduce

# Google Circa 2000

- Addicted to cheap, unreliable hardware
  - Young company, expensive hardware not practical
- Massive and growing datasets; 100s of terabytes
  - Crawl web, build massive reverse index for web search
- Few expert programmers can write distributed programs to process them
  - Scale so large jobs can't complete before failures
- Let every Google engineer with the ability to write parallel, scalable, distributed, fault-tolerant code?

# MapReduce

Fault-tolerant scaling of general computation is hard

**Solution:** restrict programming interface and shape of computations supported

User provides two functions:

$\text{map}(\text{key}, \text{value}) \rightarrow \text{list}(\langle k', v' \rangle)$

- Applies function to (key, value) pair and produces set of intermediate pairs

$\text{reduce}(\text{key}, \text{list}(\text{value})) \rightarrow \langle k', v' \rangle$

- Applies aggregation function to values
- Outputs result

MapReduce automatically handles scaling and fault tolerance



# MapReduce: Word Count

map(key, value):

for each word w in value:

EmitIntermediate(w, "1");

reduce(key, values):

int result = 0;

for each v in values:

result += ParseInt(v);

Emit(AsString(result));

# Use Cases

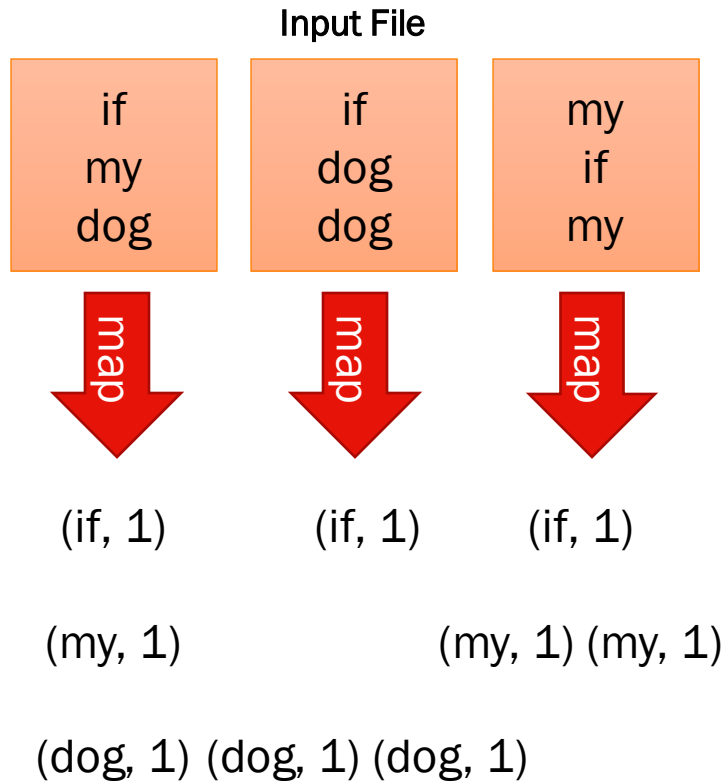
- Distributed Grep
  - map: emit line if pattern matches, else don't
  - reduce: identity function
- URL Access Frequency
  - map: emit <URL, 1>
  - reduce: <URL, count>
- Reverse Web-Link Graph
  - map: <target, source> for each link to target in source
  - reduce: <target, list(source)>
- Inverted Index
  - map: emit <hostname, list<word>> for each input document
  - reduce: throw away infrequent words, emit <hostname, list<word>>

# MapReduce: Abstract View

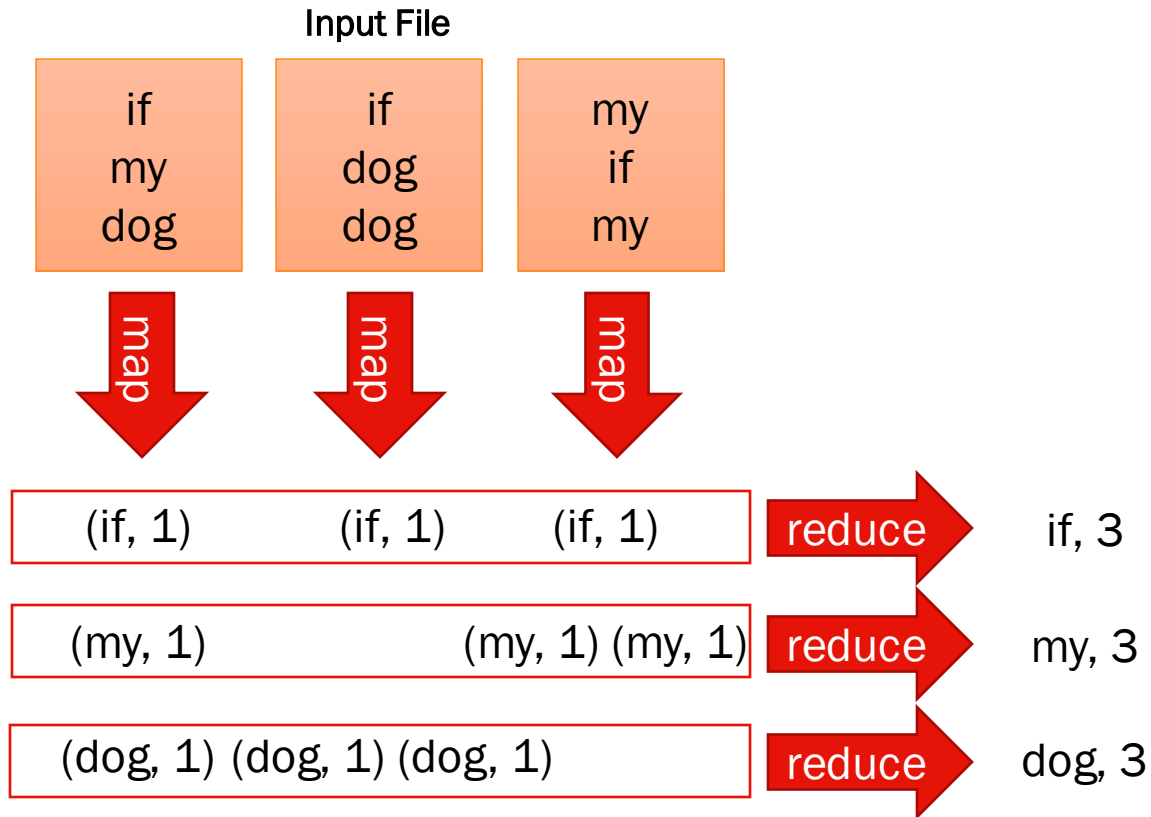
Input File

if my dog	if dog dog	my if my
-----------------	------------------	----------------

# MapReduce: Abstract View



# MapReduce: Abstract View

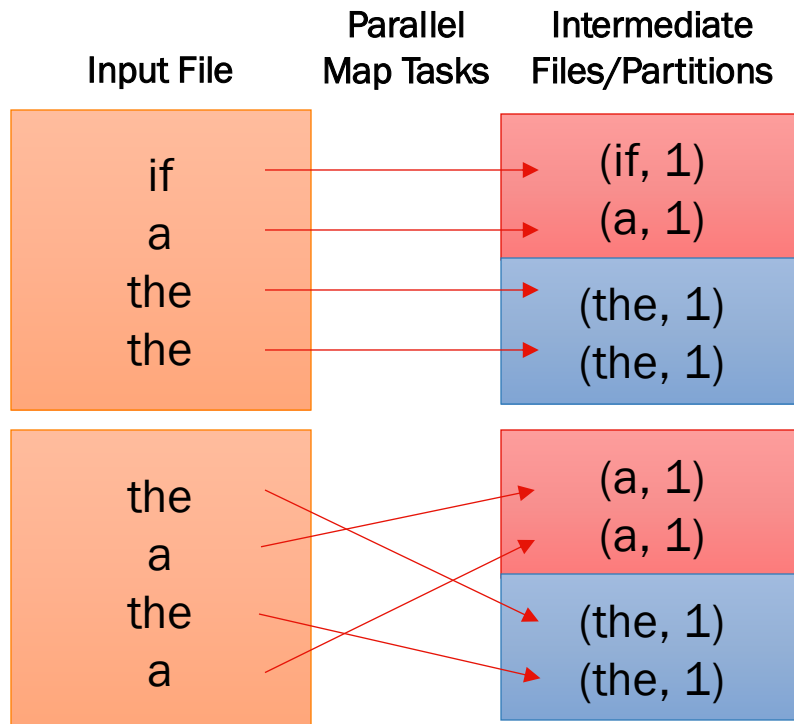


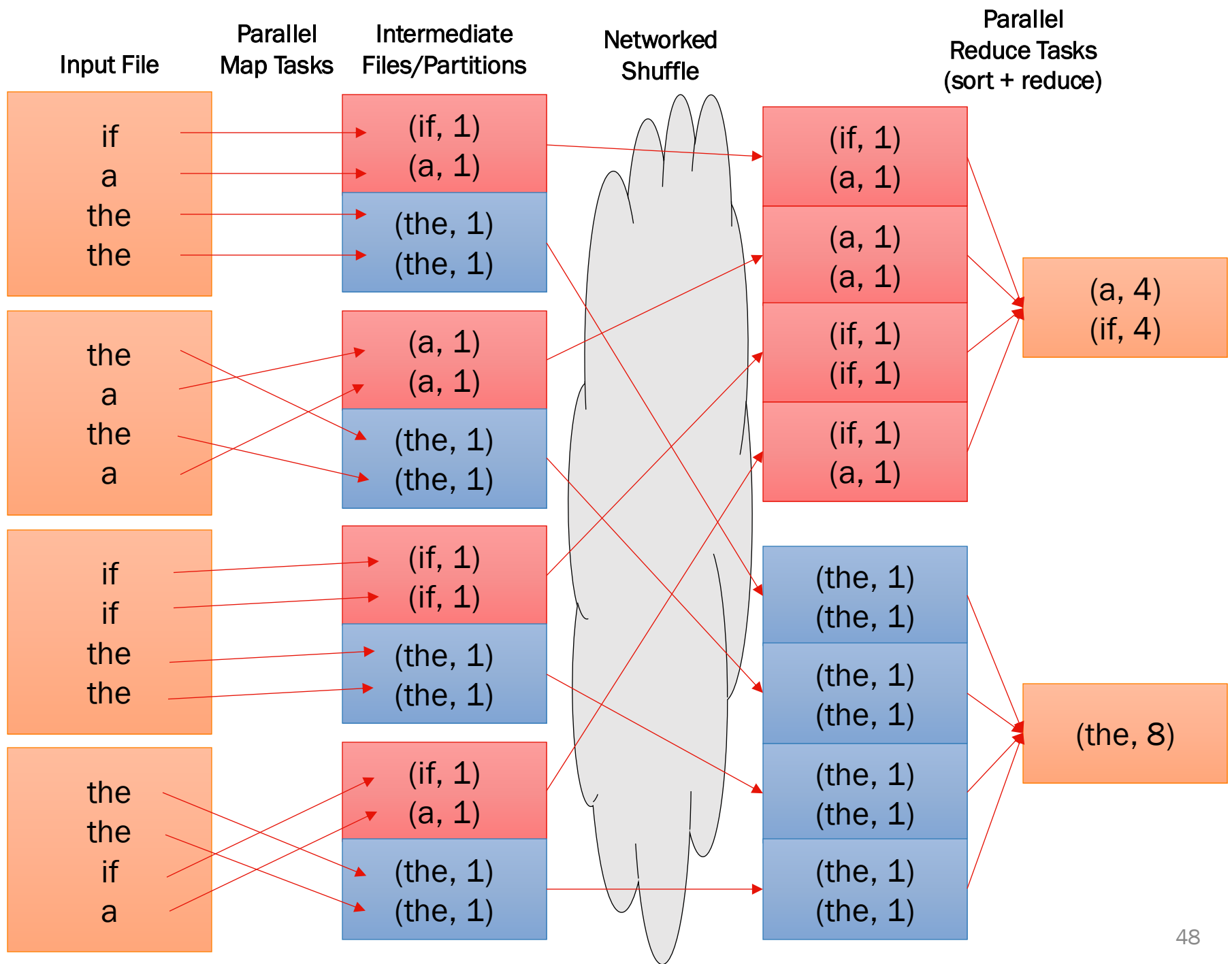


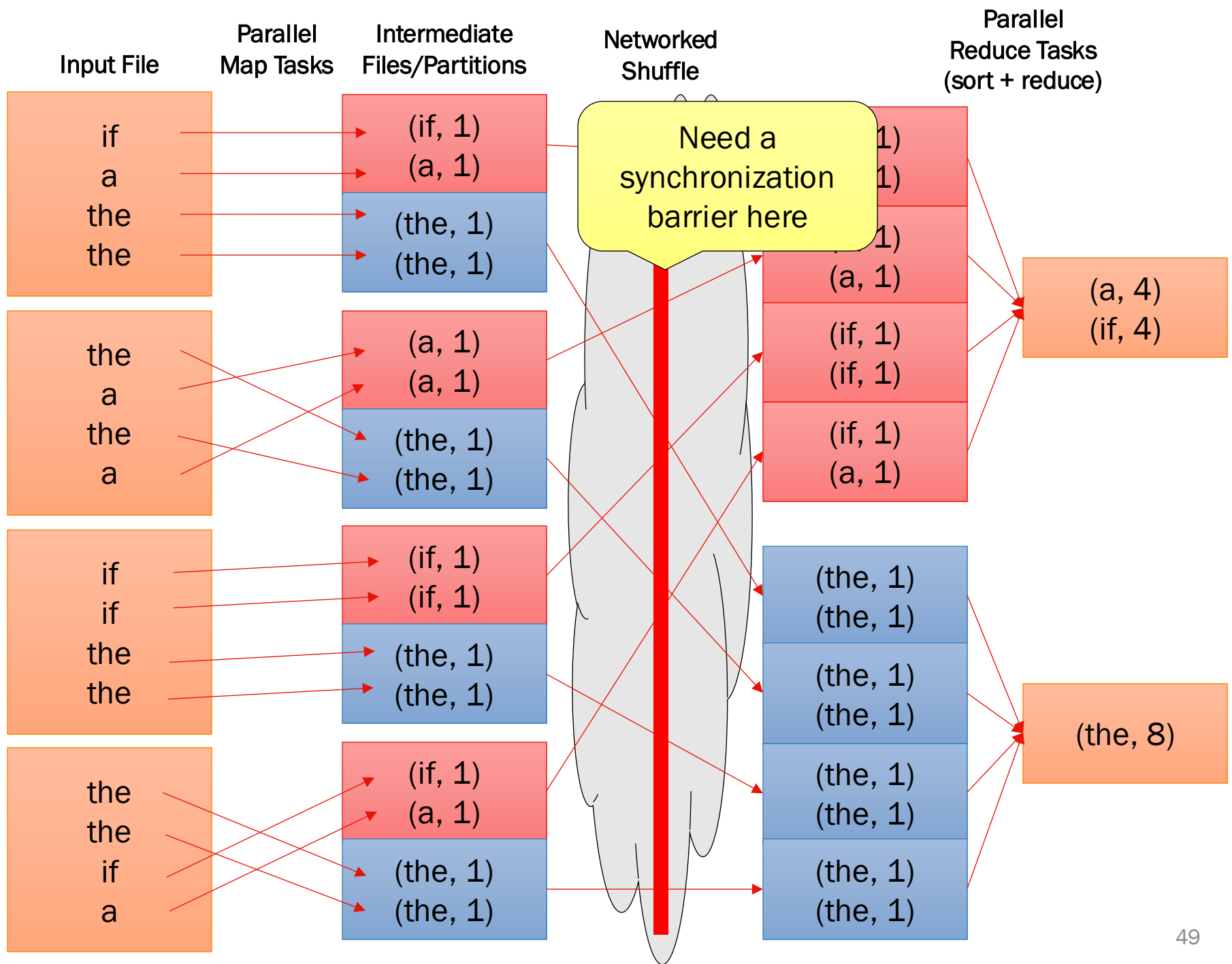
# MapReduce: Partitioning

`partition(key, n) → int`

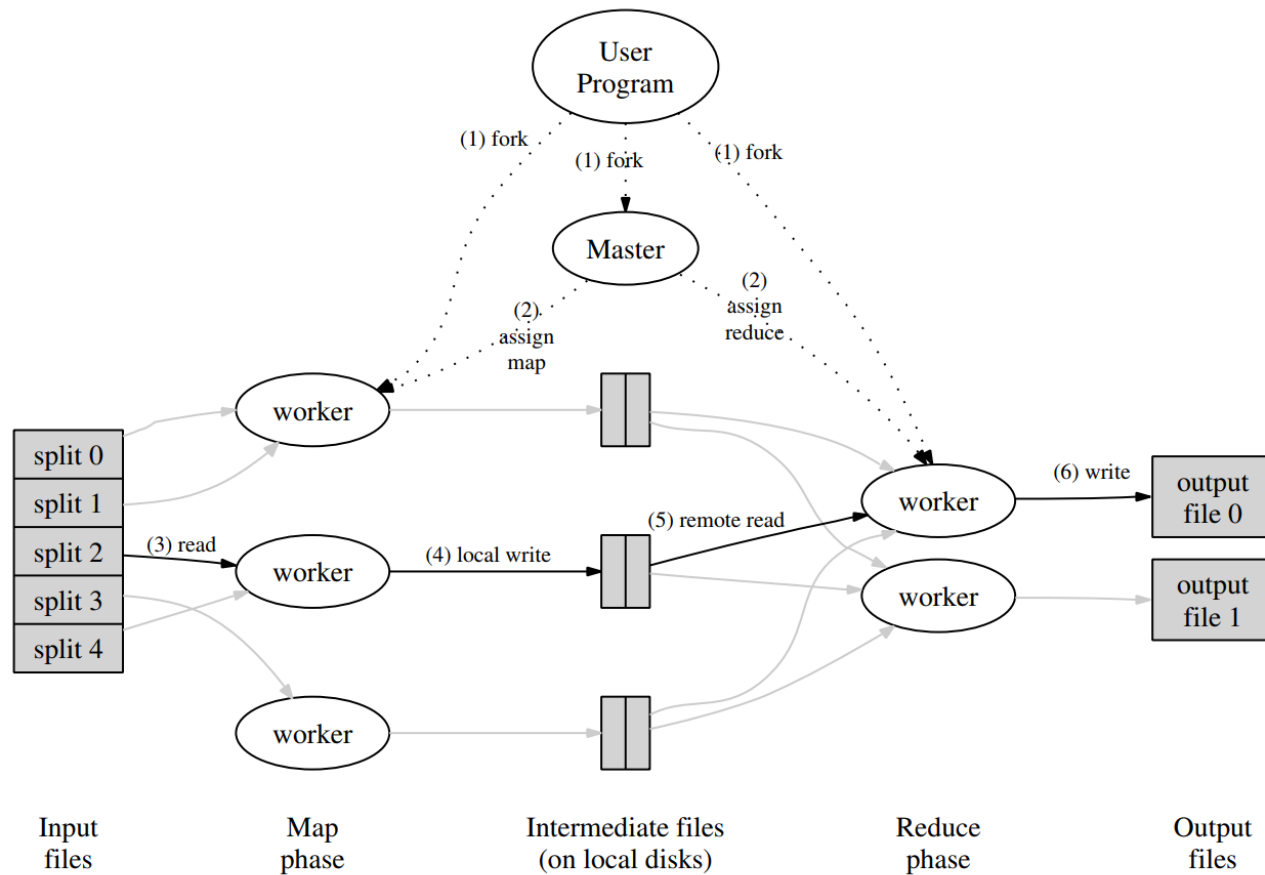
- Need to aggregate intermediate vals with same key
- Given  $n$  partitions, map key to partition  $0 \leq i < n$
- Typically, via `hash(key) mod n`







# MapReduce Processes



# MapReduce Flow

- Map task writes intermediate output to local disk, separated by partitioning; once completed, tells coordinator/master node
- Reduce task told of location of map task outputs, pulls their partition's data from each mapper, execute function across data



# Fault Tolerance in MapReduce

- Coordinator/master monitors state of system
  - What should we do if coordinator fails?
- Map worker failure
  - What is lost?
  - What should we do?
- Reducer worker failure
  - What is lost?
  - What should we do?

# Fault Tolerance in MapReduce

- Coordinator/master monitors state of system
  - If master fails, job aborts and client notified
- Map worker failure
  - Both in-progress/completed tasks marked as idle
  - Reduce workers notified when map task is re-executed on another map worker
- Reducer worker failure
  - In-progress tasks reset to idle (and re-executed)
  - Completed tasks written to global file system

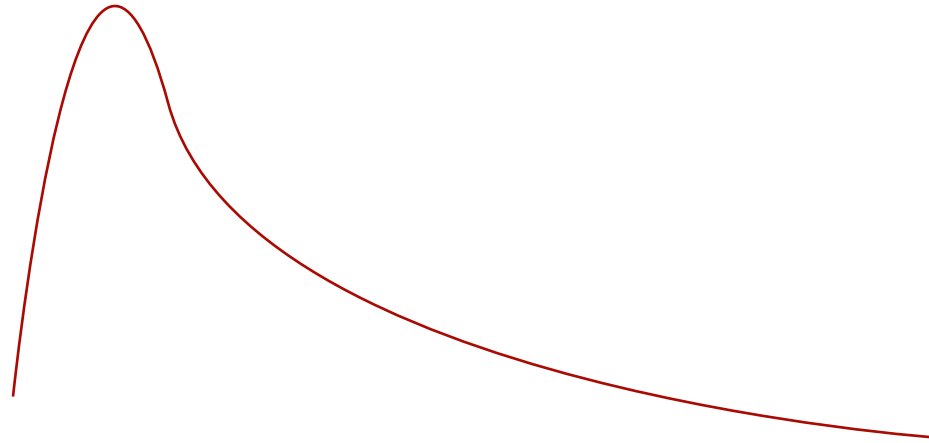
# Safety Under Failure

- Is it possible for a map task be executed twice?
- Is it possible for a reduce task be executed twice?

# Safety Under Failure

- Is it possible for a map task be executed twice?
  - Yes, e.g. lost messages could lead to same input partition being processed by two different workers
  - What makes this safe?
  - map is deterministic and side-effect free
- Reduce has the same restrictions
  - Tasks must write to temp files and then atomically rename to correct intermediate file name

# Stragglers



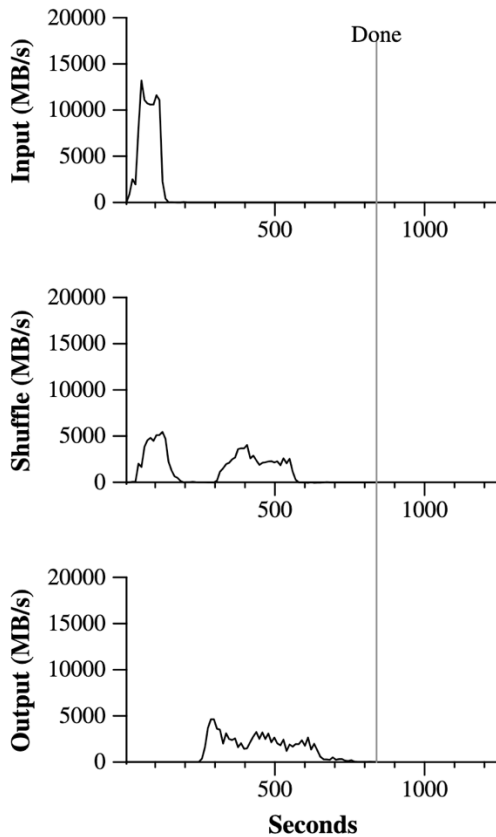
Map Task Completion Time Distribution

- "Tail latency" means some workers (always) finish late
- How can MR work around this?

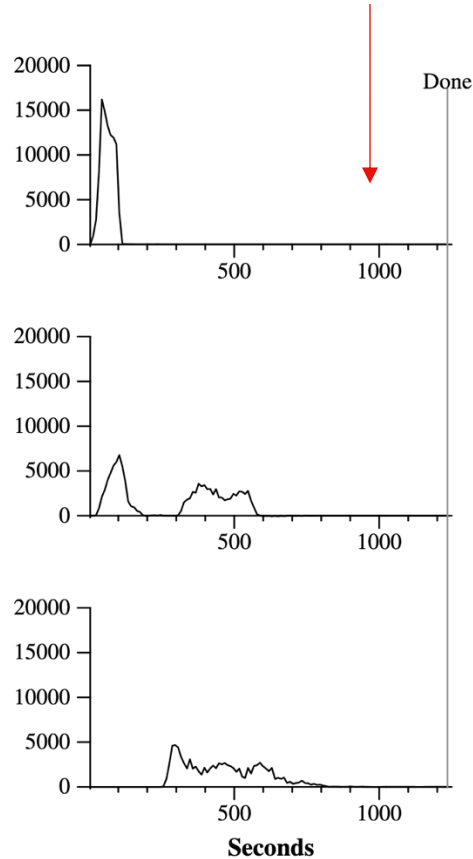


A few slow tasks  
prevent completion

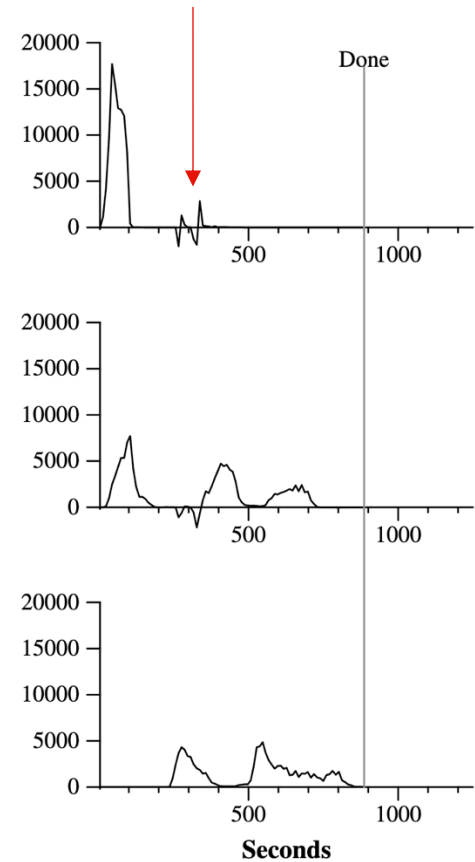
Lost intermediate files;  
recreated



(a) Normal execution



(b) No backup tasks



(c) 200 tasks killed

1800 machines, 15000 map tasks, 4000 reduce tasks, ~30 GB/s disk bw (in 2004!!)

# Takeaways

- Big picture of map-reduce model
  - Break input into chunks, process chunks in parallel
  - Each reduce task pulls output from each map task, input partitioned so all common keys are in one partition; sort, reduce input values for key, output
  - Failures: redo task
- Determinism eases fault-tolerance (+ helps perf)
  - Map-reduce repeats work under partial failure
  - Results are the same, since repeating work produces the same results

# For Thursday

- Do Lab Setup and Online Go Tour,
  - **Pay special attention to Concurrency in Go Tour**
  - goroutines, buffered/unbuffered channels, mutexes
- Go helpful to avoid ratholeing on mundane details
  - Labs try to ensure that hard problems are w/ distributed systems
  - not e.g. fighting against language, libraries, etc.
  - Go is type-safe, garbage collected, slick RPC library
  - makes building services/systems simpler
- Start on Homework 1 (Due Tue)