Course overview, principles, MapReduce

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

CS 240: *Computing Systems and Concurrency*
Lecture 1

Marco Canini

Credits: Michael Freedman and Kyle Jamieson developed much of the original material.
Parts adapted from CMU 15-440.

Backrub (Google) 1997

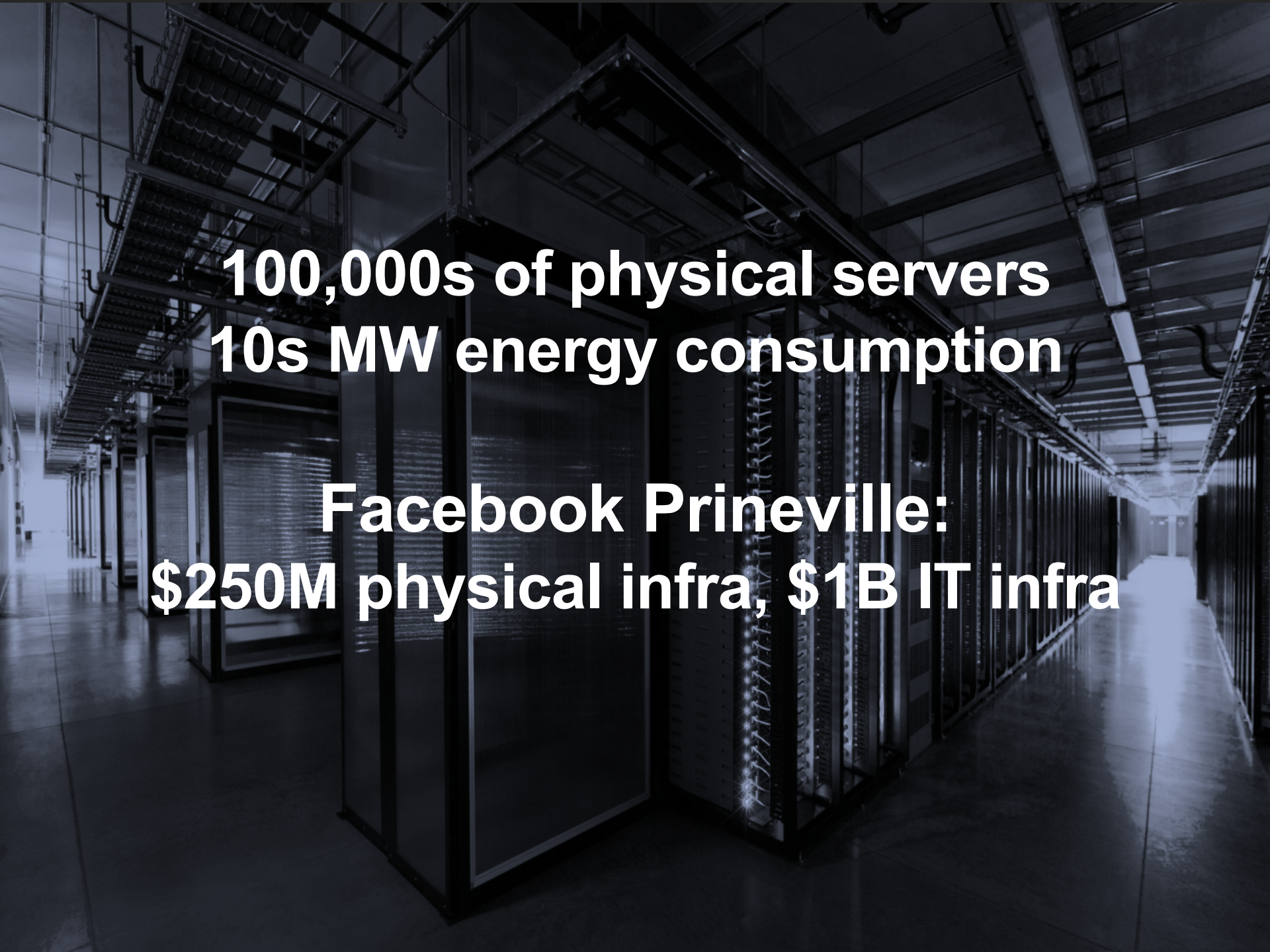
Google 2012

“The Cloud” is not amorphous

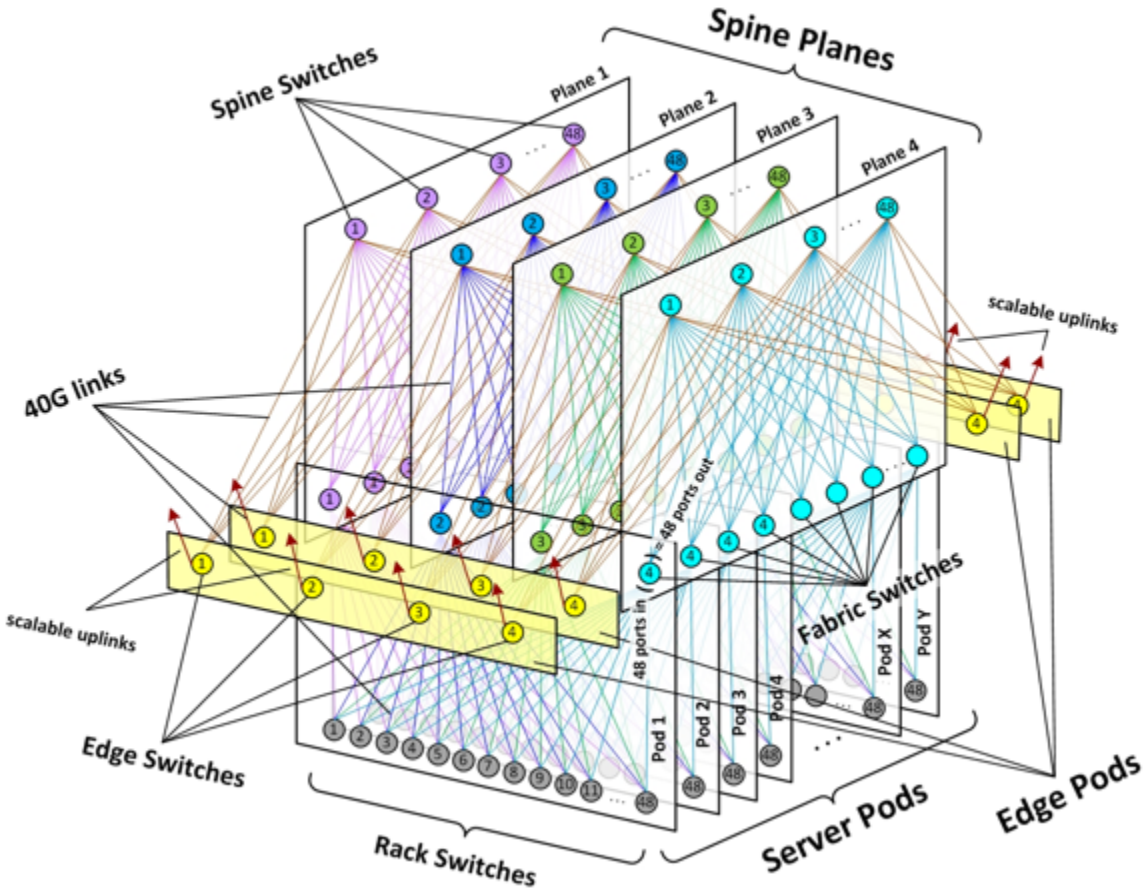
Microsoft


Google


Facebook


100,000s of physical servers
10s MW energy consumption
Facebook Prineville:
$250M physical infra, $1B IT infra

Everything changes at scale



“Pods provide 7.68Tbps to backplane”

The goal of “distributed systems”

- Service with higher-level abstractions/interface
  - e.g., file system, database, key-value store, programming model, RESTful web service, …
- Hide complexity
  - Scalable (scale-out)
  - Reliable (fault-tolerant)
  - Well-defined semantics (consistent)
  - Security
- Do “heavy lifting” so app developer doesn’t need to

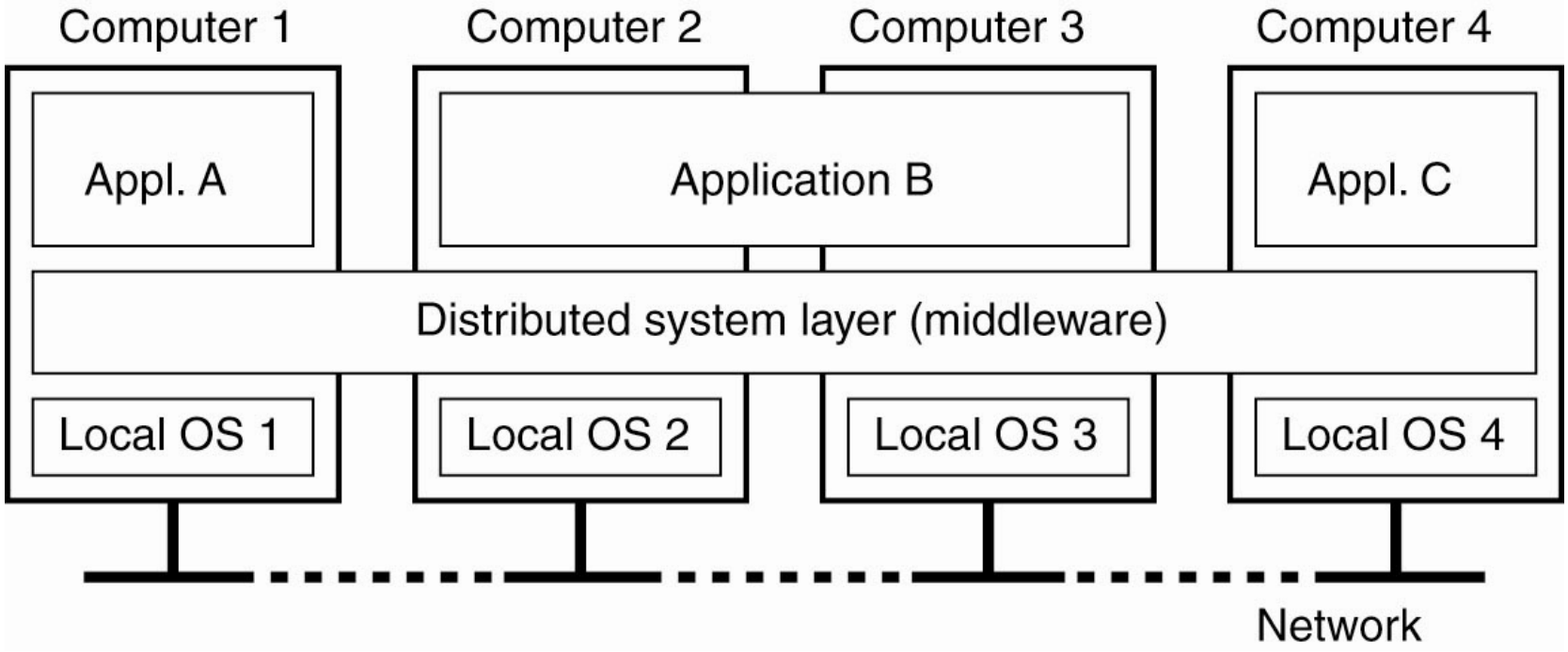
What is a distributed system?

- “A collection of independent computers that appears to its users as a single coherent system”
- Features:
  - No shared memory
  - Message-based communication
  - Each runs its own local OS
  - Heterogeneity
- Ideal: to present a single-system image:
  - The distributed system “looks like” a single computer rather than a collection of separate computers

Distributed system characteristics

- To present a single-system image:
  - Hide internal organization, communication details
  - Provide uniform interface
- Easily expandable
  - Adding new computers is hidden from users
- Continuous availability
  - Failures in one component can be covered by other components
- Supported by middleware

Distributed system as middleware



- A distributed system organized as middleware
- The middleware layer runs on all machines, and offers a uniform interface to the system

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

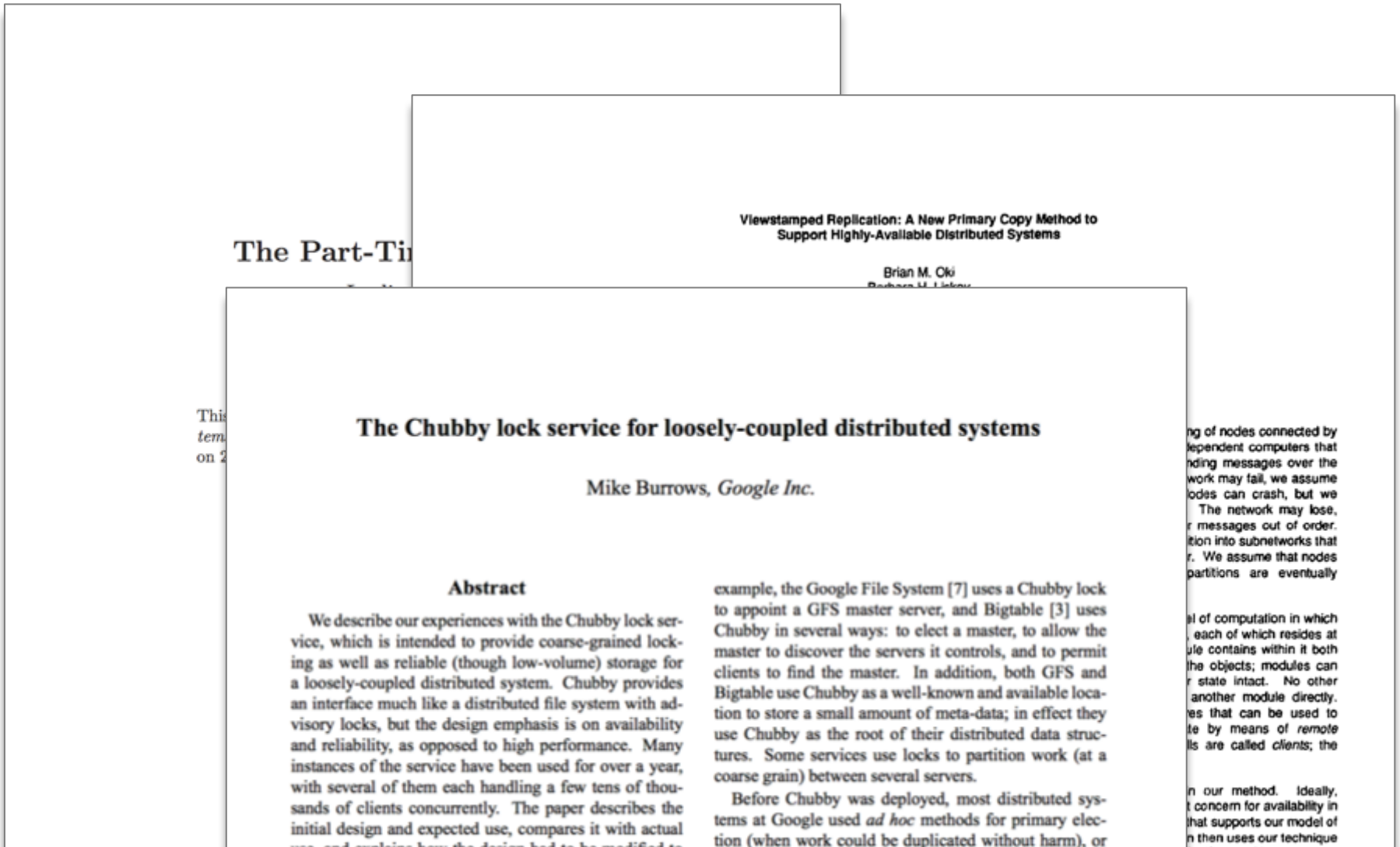
Algorithms, Management, Measurement, Performance, Design, Reliability.

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

Research results matter: Paxos



The Part-Ti

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

Brian M. Oki

The Chubby lock service for loosely-coupled distributed systems

Mike Burrows, *Google Inc.*

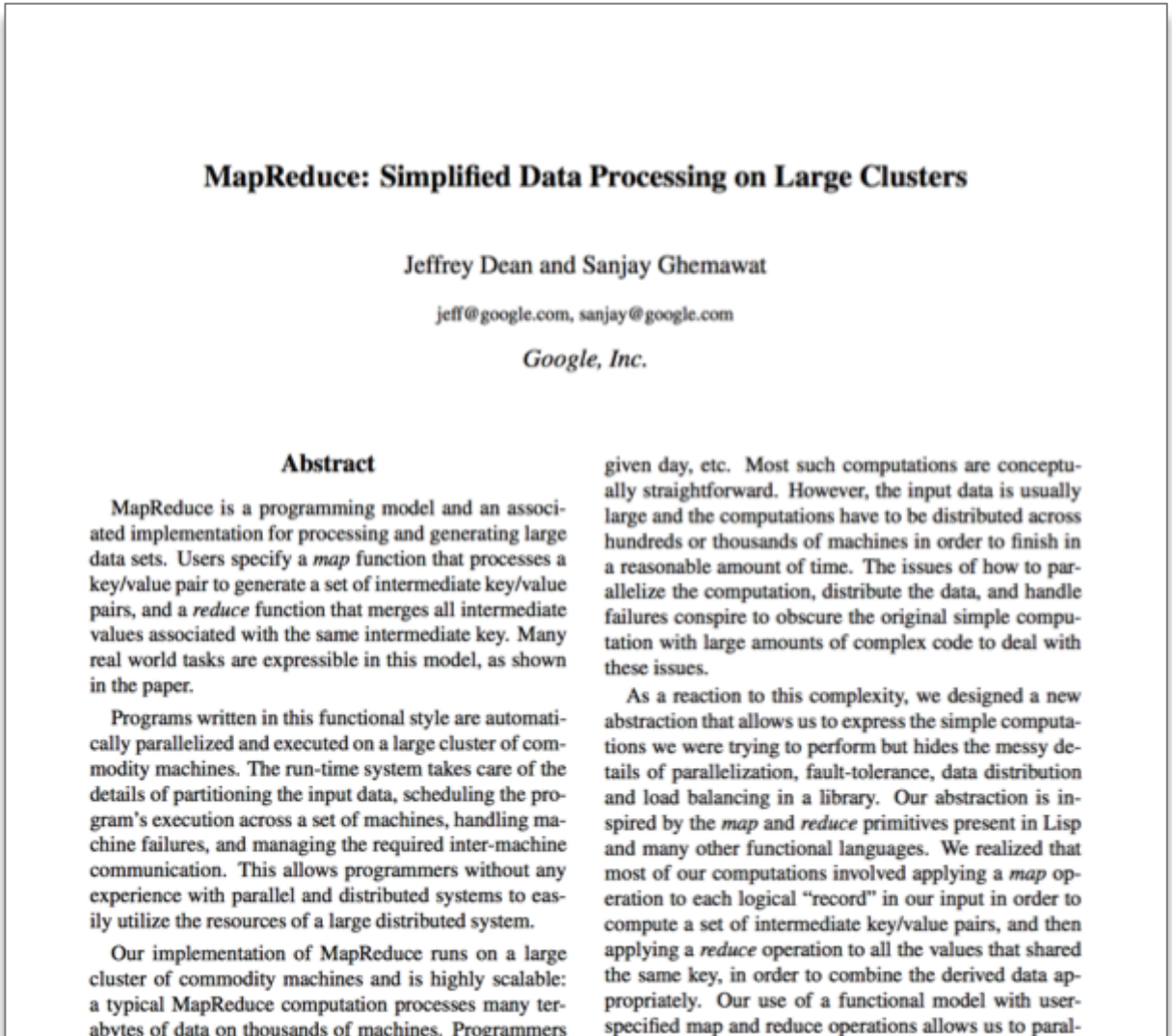
Abstract

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

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

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-

hadoop

Spark

kafka

Flink

S4 distributed stream computing platform

STORM

Course Organization

Course Goals

- Gain an understanding of the principles and techniques behind the design of modern, reliable, and high-performance systems
- In particular learn about distributed systems
  - Learn general systems principles (modularity, layering, naming, security, ...)
  - Practice implementing real, larger systems that must run in nasty environment
- One consequence: Must pass exams and projects independently as well as in total
  - Note, if you fail either you will not pass the class

Learning the material: People

- Lecture
  - Professor Marco Canini
  - Slides available on course website
  - Office hours immediately after lecture
- TAs
  - Hassan Alsibyani
  - Humam Alwassel
- Main Q&A forum: www.piazza.com
  - No anonymous posts or questions
  - Can send private messages to instructors

Learning the Material: Books

- Lecture notes!
- No required textbooks
- References available in the Library:
  - Programming reference:
    - *The Go Programming Language.* Alan Donovan and Brian Kernighan
  - Topic reference:
    - *Distributed Systems: Principles and Paradigms.* Andrew S. Tanenbaum and Maaten Van Steen
    - *Guide to Reliable Distributed Systems.* Kenneth Birman

Grading

- Four assignments (50% total)
  - 10% for 1 & 2
  - 15% for 3 & 4

- Two exams (50% total)
  - Midterm exam on October 22 (15%)
  - Final exam during exam period (35%)

About Projects

- Systems programming somewhat different from what you might have done before
  - Low-level (C / Go)
  - Often designed to run indefinitely (error handling must be rock solid)
  - Must be secure - horrible environment
  - Concurrency
  - Interfaces specified by documented protocols
- TAs' Office Hours
- Dave Andersen's "Software Engineering for System Hackers"
  - Practical techniques designed to save you time & pain

Where is Go used?

- Google, of course!
- Docker (container management)
- CloudFlare (Content delivery Network)
- Digital Ocean (Virtual Machine hosting)
- Dropbox (Cloud storage/file sharing)
- … and many more!

Why use Go?

- Easy concurrency w/ goroutines (lightweight threads)
- Garbage collection and memory safety
- Libraries provide easy RPC
- Channels for communication between goroutines

Collaboration

- Working together important
  - Discuss course material
  - Work on problem debugging
- Parts **must** be your own work
  - Midterm, final, solo projects
- Team projects: both students should understand entire project
- What we hate to say: we run cheat checkers…
- Please *do not* put code on *public* repositories
- Partner problems: *Please address them early*

Policies: Write Your Own Code

Programming is an individual creative process. At first, discussions with friends is fine. When writing code, however, the program must be your own work.

Do not copy another person's programs, comments, README description, or any part of submitted assignment. This includes character-by-character transliteration but also derivative works. Cannot use another's code, etc. even while "citing" them.

Writing code for use by another or using another's code is academic fraud in context of coursework.

Do not publish your code e.g., on Github, during/after course!

Late Work

- 72 late hours to use throughout the semester
  - (but not beyond December 6)
- After that, each additional day late will incur a 10% lateness penalty
  - (1 min late counts as 1 day late)
- Submissions late by 3 days or more will no longer be accepted
  - (Fri and Sat count as days)
- In case of illness or extraordinary circumstance (e.g., emergency), talk to us early!

Assignment 1

- Learn how to program in Go
  - Implement “sequential” MapReduce
  - Instructions on assignment web page
  - Due September 20, 23:59

Case Study: MapReduce

(Data-parallel programming at scale)

Application: Word Count

```
SELECT count(word) FROM data
    GROUP BY word
```

```
cat data.txt
    | tr -s '[[:punct:][:space:]]' '\n'
    | sort | uniq -c
```

Using partial aggregation

1. Compute word counts from individual files
2. Then merge intermediate output
3. Compute word count on merged outputs

Using partial aggregation

1. In parallel, send to worker:
   - Compute word counts from individual files
   - Collect result, wait until all finished
2. Then merge intermediate output
3. Compute word count on merged intermediates

MapReduce: Programming Interface

`map(key, value) -> list(<k', v'>)`

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

`reduce(key, list<value>) -> <k', v'>`

- Applies aggregation function to values
- Outputs result

MapReduce: Programming Interface

```
map(key, value):
   for each word w in value:
      EmitIntermediate(w, "1");


reduce(key, list(values):
   int result = 0;
   for each v in values:
      result += ParseInt(v);
   Emit(AsString(result));
```

MapReduce: Optimizations

`combine(list<key, value>) -> list<k,v>`

- Perform partial aggregation on mapper node:

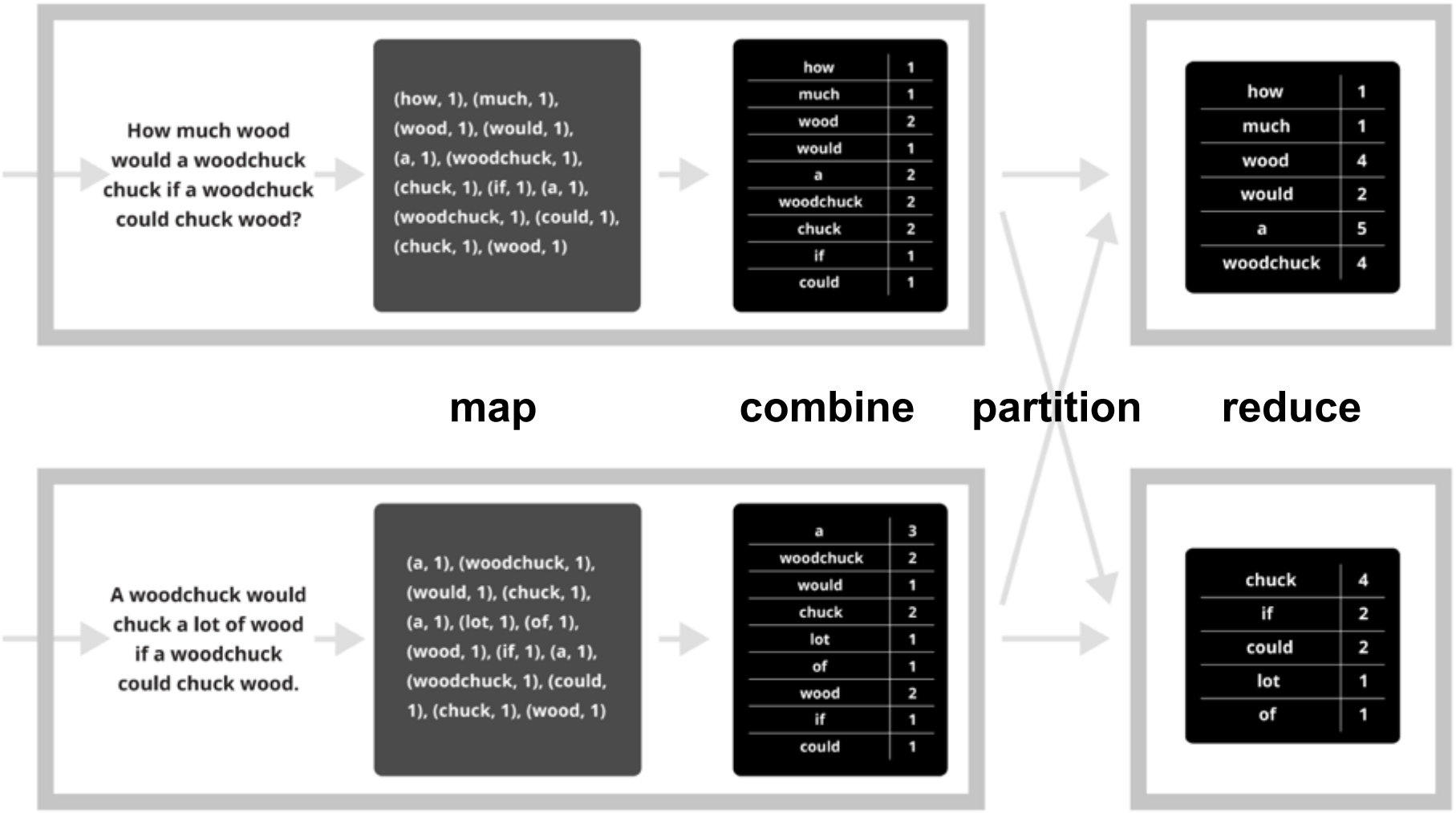
  <the, 1>, <the, 1>, <the, 1> → <the, 3>

- combine() should be commutative and associative

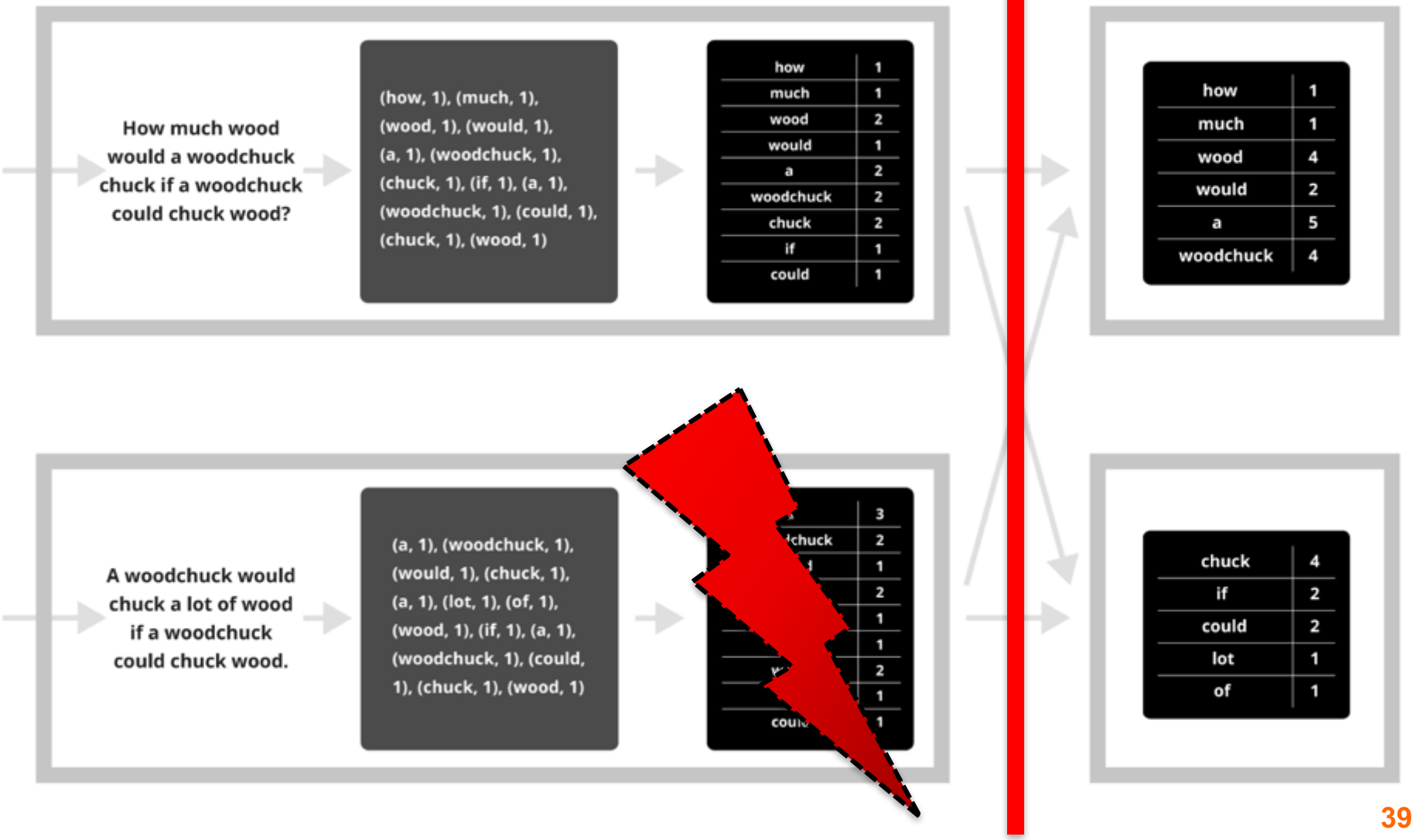
`partition(key, int) -> int`

- 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

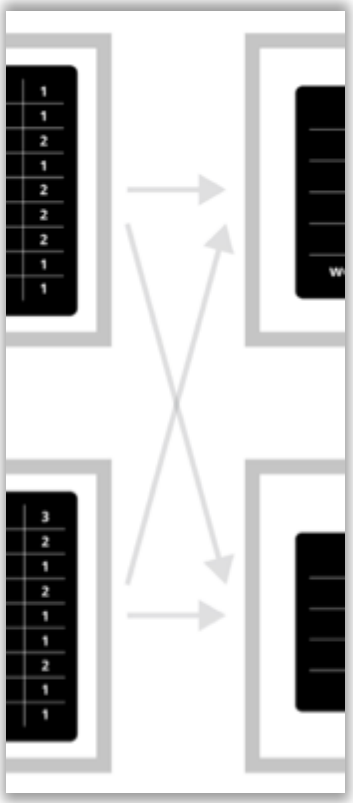
Putting it together…



Synchronization Barrier



Fault Tolerance in MapReduce

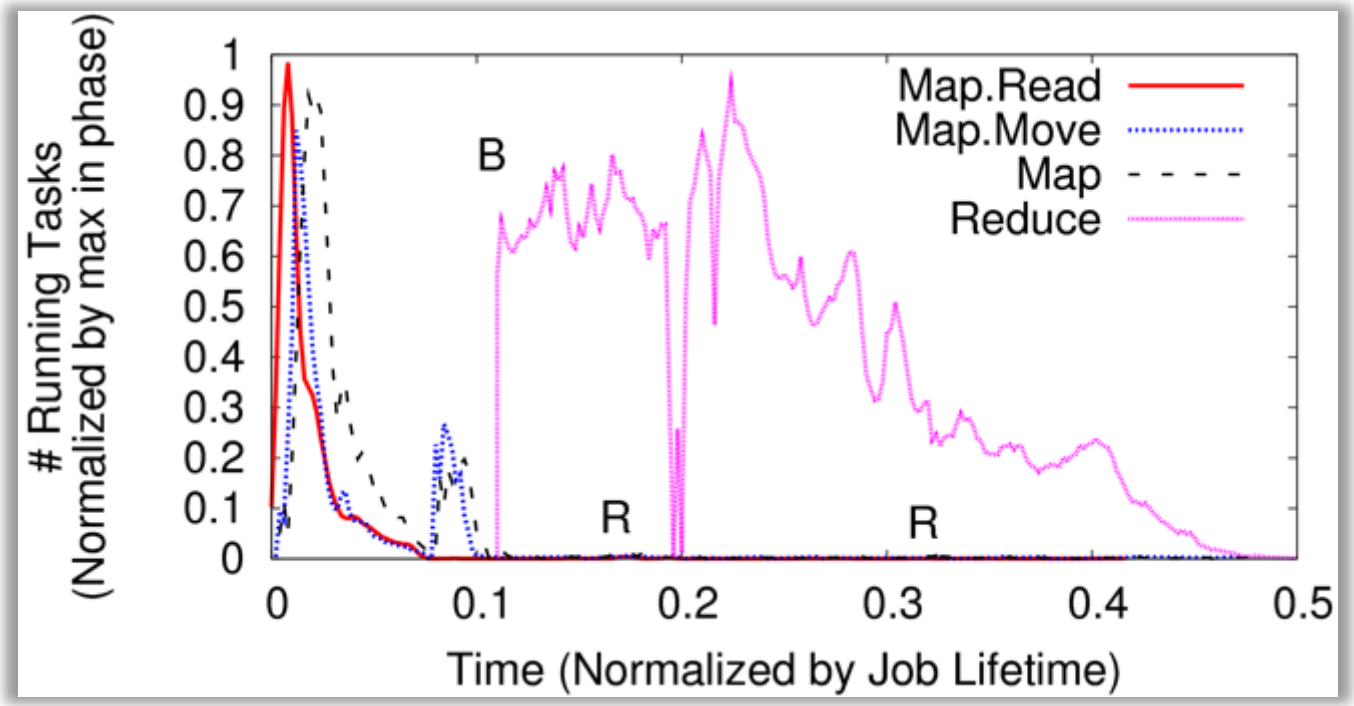


- Map worker writes intermediate output to local disk, separated by partitioning. Once completed, tells master node.
- Reduce worker told of location of map task outputs, pulls their partition's data from each mapper, execute function across data
- Note:
  - "All-to-all" shuffle b/w mappers and reducers
  - Written to disk ("materialized") b/w *each* stage

Fault Tolerance in MapReduce

- Master node monitors state of system
  - If master failures, 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 are reset to idle (and re-executed)
  - Completed tasks had been written to global file system

Straggler Mitigation in MapReduce



- Tail latency means some workers finish late
- For slow map tasks, execute in parallel on second map worker as “backup”, race to complete task

You'll build (simplified) MapReduce!

- Assignment 1: Sequential MapReduce
  - Learn to program in Go!
  - Due September 20

- Assignment 2: Distributed MapReduce
  - Learn Go's concurrency, network I/O, and RPCs
  - Due October 15