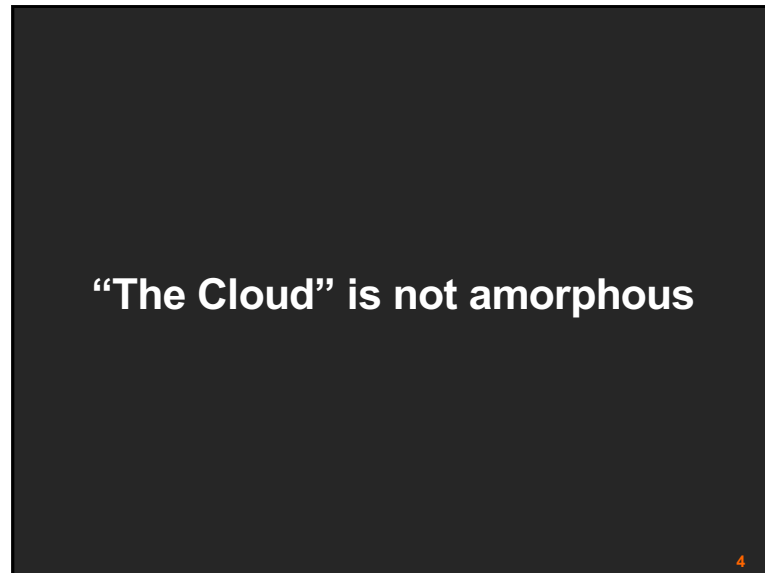


# Distributed Systems



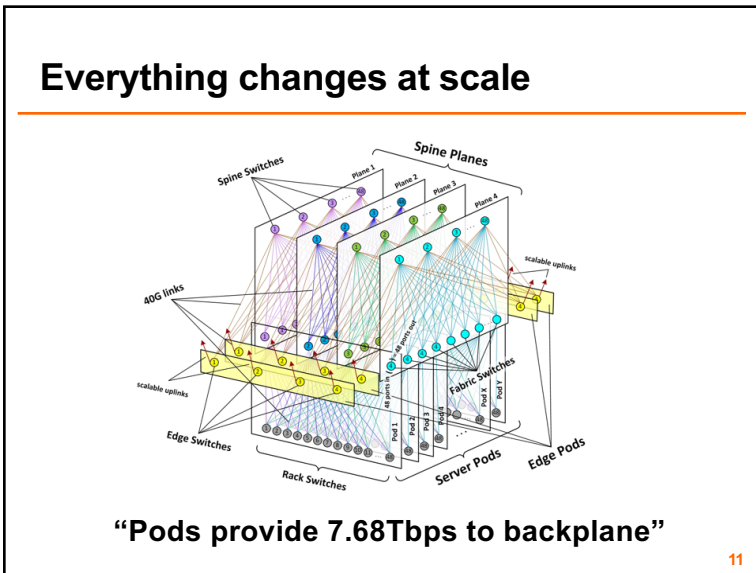
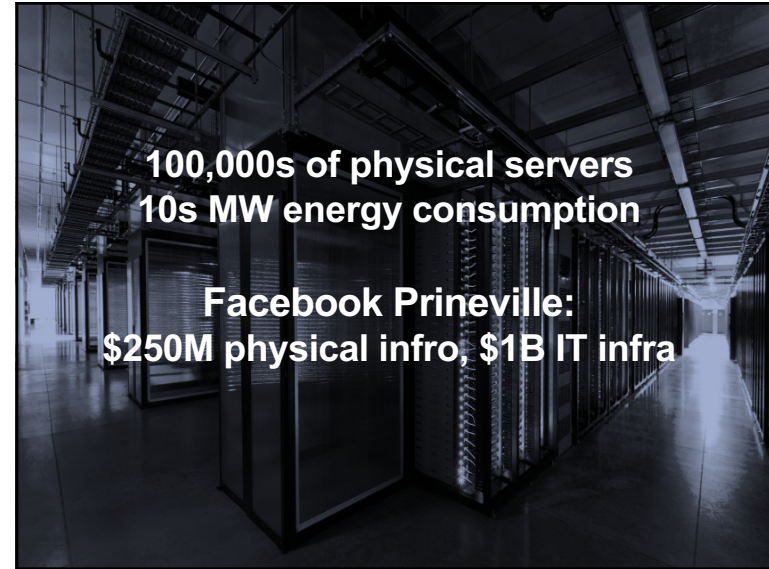
COS 418: *Distributed Systems*  
Lecture 1

Mike Freedman









- ### 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
- 12

# Research results matter: NoSQL

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

Giuseppe DeCandia, Deniz Hastorun, Madan Jambani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Plichin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels  
Amazon.com

Distribut  
David Ka

### Abstract

We describe a family of data stores that can be used in the network. Our very large networks but space can be used to have complete in network. The main work primary and. The network work ing network, and. Our caching pat the we self-manage back function is ree function changes. back function, we not require any an network. We believe gones to be useful if service may be exp

### 1 Introduction

In this paper, we describe a family of data stores that can be used in the network. Our very large networks but space can be used to have complete in network. The main work primary and. The network work ing network, and. Our caching pat the we self-manage back function is ree function changes. back function, we not require any an network. We believe gones to be useful if service may be exp

### 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 very 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.4.1 [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 tornadoes. 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

may be parti an message in a central pool and partitioned sign copies with

re-represented Replication is available from the. Weak consistency is a property of the copy which is not guaranteed to be the most up-to-date or the most recent.

just as designing applied data applications. We may read weakly but may conflict the application. For applications, the data is not consistent, and we need to deal with the data as it is being

# Research results matter: Paxos

## The Part-T Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems

Brian M. Ouy

## 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, such as *rank*, could be implemented without having an

of nodes connected by network connections that bring messages over the network. We assume nodes can crash, but we assume the network may lose messages out of order. We assume that nodes join and leave the system eventually. If computation in each node which receives at least one message from the other nodes can state in its. No other nodes can directly be that can be used to be by means of remote is are called clients; the our method. Ideally, concern for availability in but supports our model of than uses our technology

# 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

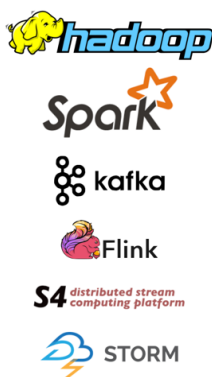
MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the 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 ter-

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-



# Course Organization



## Learning the material: People

---

- Lecture
  - Professors Mike Freedman, Kyle Jamieson
  - Slides available on course website
  - Office hours immediately after lecture
- Precept:
  - TAs Themis Melissaris, Daniel Suo
- Main Q&A forum: [www.piazza.com](http://www.piazza.com)
  - Graded on class participation: so ask & answer!
  - No anonymous posts or questions
  - Can send private messages to instructors

17

## Learning the Material: Books

---

- Lecture notes!
- No required textbooks.
  - Programming reference:
    - *The Go Programming Language*, Alan Donovan and Brian Kernighan ([www.gopl.io](http://www.gopl.io), \$17 Amazon!)
  - Topic reference:
    - *Distributed Systems: Principles and Paradigms*. Andrew S. Tanenbaum and Maaten Van Steen
    - *Guide to Reliable Distributed Systems*. Kenneth Birman

18

## Grading

---

- Five assignments (5% for first, then 10% each)
  - 90% 24 hours late, 80% 2 days late, 50% >5 days late
  - **THREE** free late days (we'll figure which one is best)
  - Only failing grades I've given are for students who don't (try to) do assignments
- Two exams (50% total)
  - Midterm exam before spring break (25%)
  - Final exam during exam period (25%)
- Class participation (5%)
  - In lecture, precept, and Piazza

19

## 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!

20

## Assignment 1

---

- Learn how to program in Go
  - Implement “sequential” Map/Reduce
  - Instructions on assignment web page
  - Due September 28 (two weeks)

21

## meClickers®: Quick Surveys

---

- Why are you here?
  - A. Needed to satisfy course requirements
  - B. Want to learn Go or distributed programming
  - C. Interested in concepts behind distributed systems
  - D. Thought this course was “Bridges”

22

## Case Study: MapReduce

(Data-parallel programming at scale)

23

## Application: Word Count

---

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

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

24



## Using partial aggregation

---

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

25

## 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

26

## 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

27

## 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));
```

28

## MapReduce: Optimizations

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

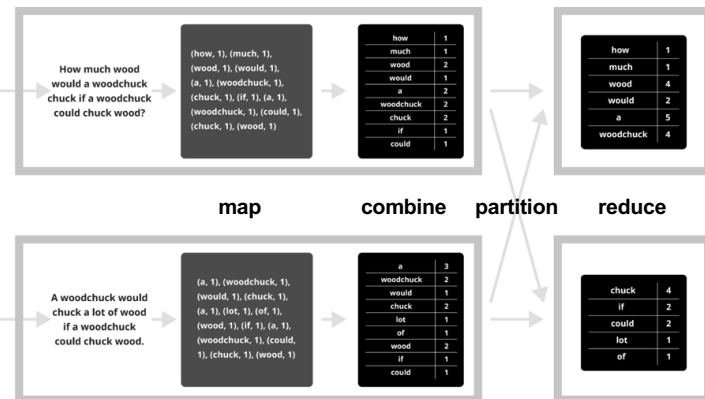
- Perform partial aggregation on mapper node:  
`<the, 1>, <the, 1>, <the, 1> -> <the, 3>`
- `reduce()` should be commutative and associative

`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`

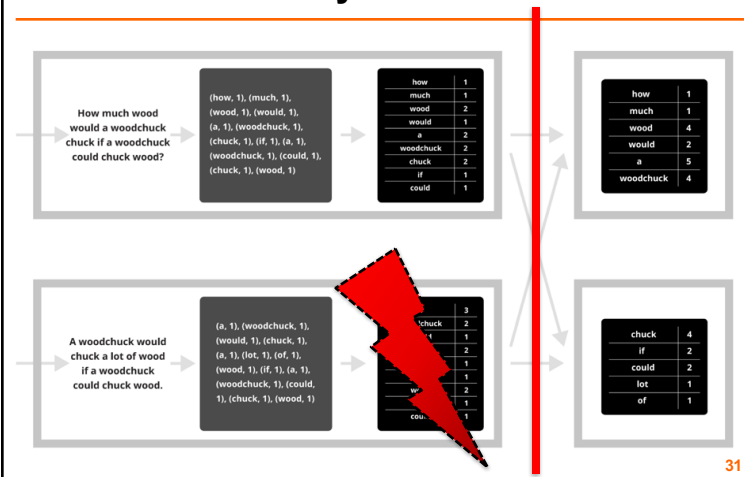
29

## Putting it together...



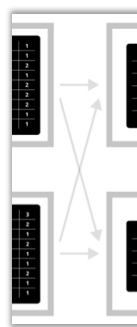
30

## Synchronization Barrier



31

## 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

32

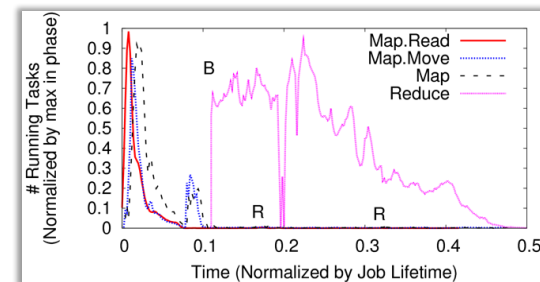


## Fault Tolerance in MapReduce

- 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

33

## 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

34

## You’ll build (simplified) MapReduce!

- Assignment 1: Sequential Map/Reduce
  - Learn to program in Go!
  - Due September 28 (two weeks)
- Assignment 2: Distributed Map/Reduce
  - Learn Go’s concurrency, network I/O, and RPCs
  - Due October 19

35

## This Friday “Grouped” Precept, Room CS 105



“Program your next service in Go”  
Sameer Ajmani

Manages Go lang team @ Google  
(Earlier: PhD w/ Barbara Liskov @ MIT)

36