Scalable Machine Learning

1. Systems

Alex Smola
Yahoo! Research and ANU

http://alex.smola.org/teaching/berkeley2012
Stat 260 SP 12

Basics

Important Stuff

- Time
  - Class - Tuesday 4-7pm
  - Q&A - Tuesday 1-3pm (Evans Hall 418)
  - Tutor - Dapo Omidiran
- Grading policy
  - Assignments (20), project (45), midterm (15), final exam (20), scribe (3)
  - Exams will be without technology.
    You can bring a paper notebook (8″x10″)

You can get 103%

Important Stuff

- Homework
  - 5 sets of assignments
  - Do it yourself. I will not check plagiarism.
  - Discussing with others is encouraged but you hurt yourself if you don't solve the problems.
- Drop off your homework in class.
  No late drops accepted.
  No exceptions.
- Only the best 4 assignments count.

Can you look at yourself in the mirror?

Important Stuff

- Project
  - Do it well (you get 45% of the score)
  - Start early (you stress puppies, too)
  - Each team member gets the same score
  - Ask me if you're looking for ideas

GSI

- Dapo Omidiran + one more
- Piazza discussion board
  http://tinyurl.com/cs281b-discussion
- Office hours poll
  http://tinyurl.com/cs281b-poll
- Signup list for scribing on Piazza
  TBD

Scalable Machine Learning

- Systems
- Basic Statistics
- Data streams and sketches
- Optimization
- Generalized Linear Models
- Kernels and Regularization
- Recommender Systems
- Graphical Models
- Large Scale Inference
- Applications
- Active Learning / Bandits and Exploration

Scalable Machine Learning

- Systems
- Basic Statistics
- Data streams and sketches
- Optimization
- Generalized Linear Models
- Kernels and Regularization
- Recommender Systems
- Graphical Models
- Large Scale Inference
- Applications
- Active Learning / Bandits and Exploration

for the internet

Scalable Machine Learning

- Systems
- Basic Statistics
- Data streams and sketches
- Optimization
- Generalized Linear Models
- Kernels and Regularization
- Recommender Systems
- Graphical Models
- Large Scale Inference
- Applications
- Active Learning / Bandits and Exploration

for the internet

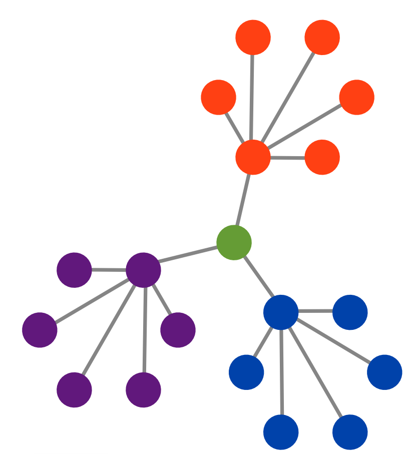
all you need
for a startup

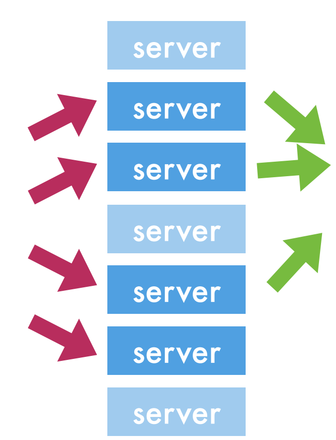
1. Systems

Algorithms run on **MANY REAL and FAULTY boxes** not Turing machines. So we need to deal with it.

Systems

- Hardware
  CPU, RAM, GPU, disks, switches, server centers
- Data
  text, video, images, clicks, networks, location
- Parallelization strategies
  consistent (proportional) hashing, trees, P2P
- Storage
  RAID, GFS, Hadoop, Ceph
- Processing
  MapReduce, Pregel, Dryad, S4
- Databases / (key,value)
  BigTable, Pnuts, Cassandra

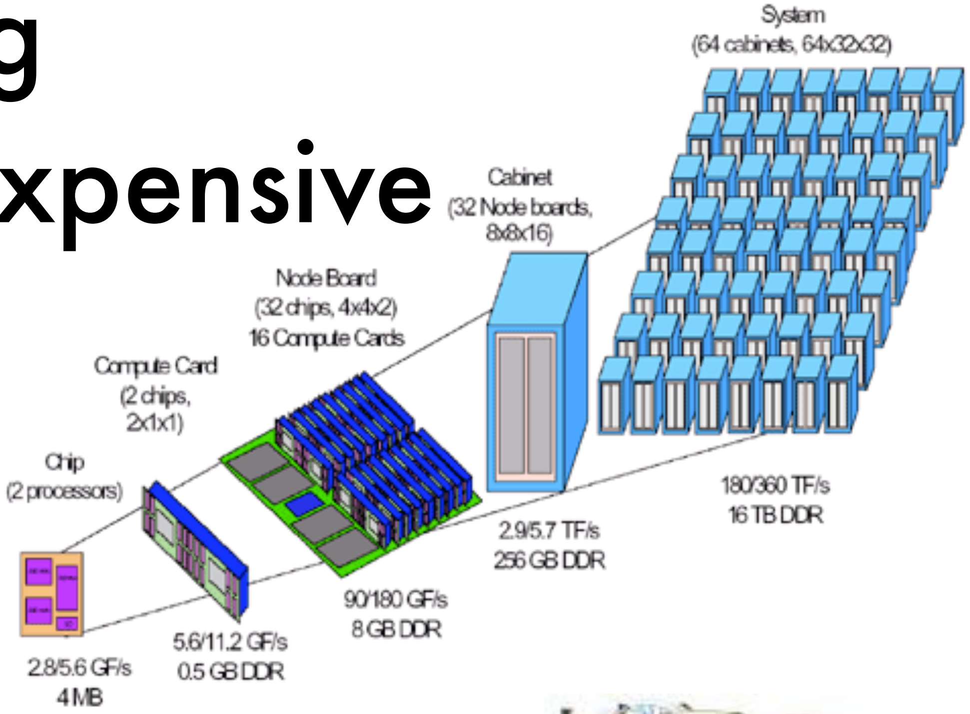




1.1 Hardware

Commodity Hardware

- High Performance Computing
  Very reliable, custom built, expensive



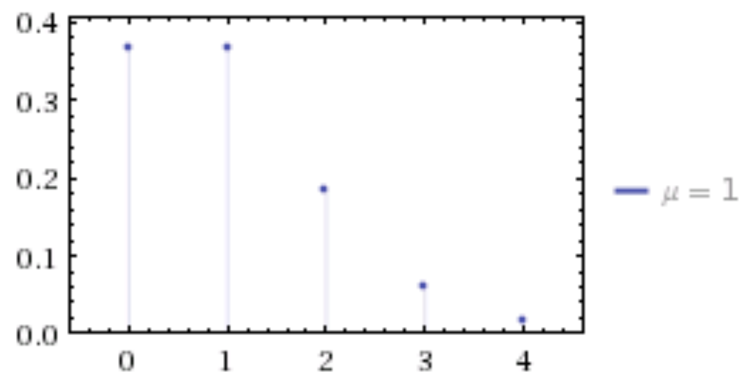
- Consumer hardware
  Cheap, efficient, easy to replicate,
  not very reliable, deal with it!

Fault tolerance

- Performance goal
  - 1 failure per year
  - 1000 machines
- Poisson approximation

not IBM Deskstar!

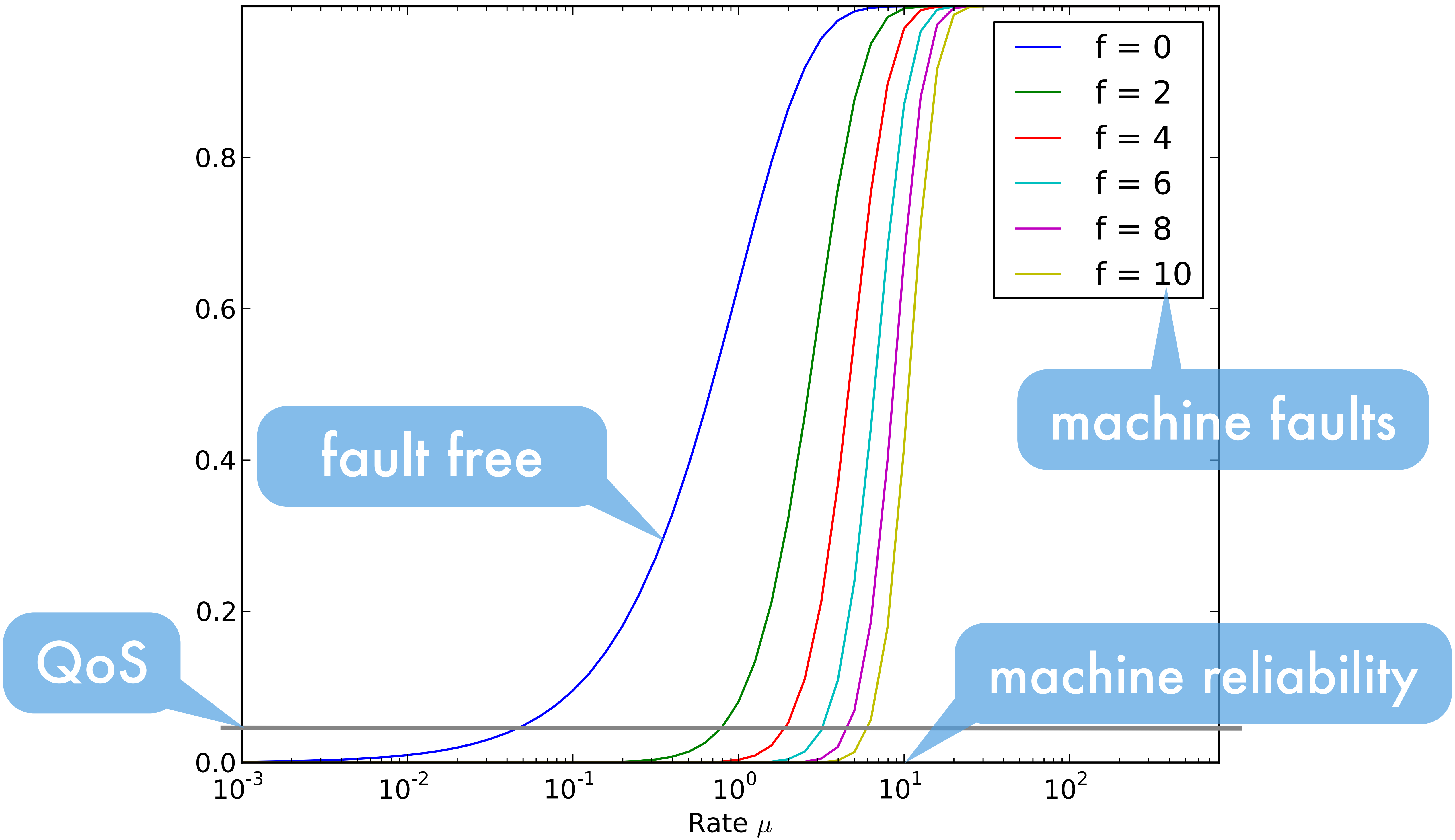
$$\Pr(n) = \frac{1}{n!} e^{-\mu} \mu^n$$



  - Assume failure rate μ per machine
  - Poisson rates of independent random variables are additive, so we can combine
- Fault intolerant engineering
  We need a rate of 1 failure per 1000 years per machine
- Fault tolerance
  Assume we can tolerate k faults among m machines in t time

$$\Pr(f > k) = 1 - \sum_{n=0}^{k} \frac{1}{n!} e^{-\lambda t} (\lambda t)^n$$

Fault tolerance



The Joys of Real Hardware

Typical first year for a new cluster:

~0.5 overheating (power down most machines in <5 mins, ~1-2 days to recover)

~1 PDU failure (~500-1000 machines suddenly disappear, ~6 hours to come back)

~1 rack-move (plenty of warning, ~500-1000 machines powered down, ~6 hours)

~1 network rewiring (rolling ~5% of machines down over 2-day span)

~20 rack failures (40-80 machines instantly disappear, 1-6 hours to get back)

~5 racks go wonky (40-80 machines see 50% packetloss)

~8 network maintenances (4 might cause ~30-minute random connectivity losses)

~12 router reloads (takes out DNS and external vips for a couple minutes)

~3 router failures (have to immediately pull traffic for an hour)

~dozens of minor 30-second blips for dns

~1000 individual machine failures

~thousands of hard drive failures

slow disks, bad memory, misconfigured machines, flaky machines, etc.

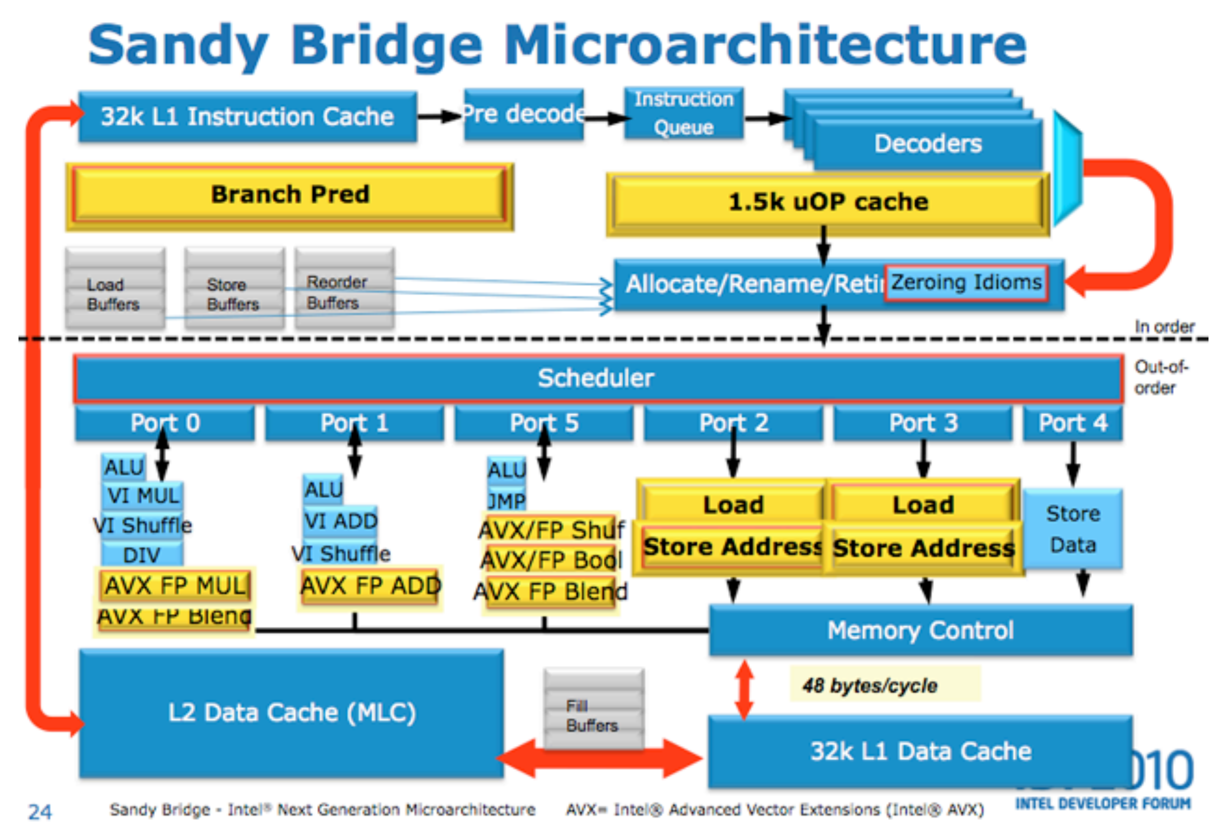
Slide from talk of Jeff Dean

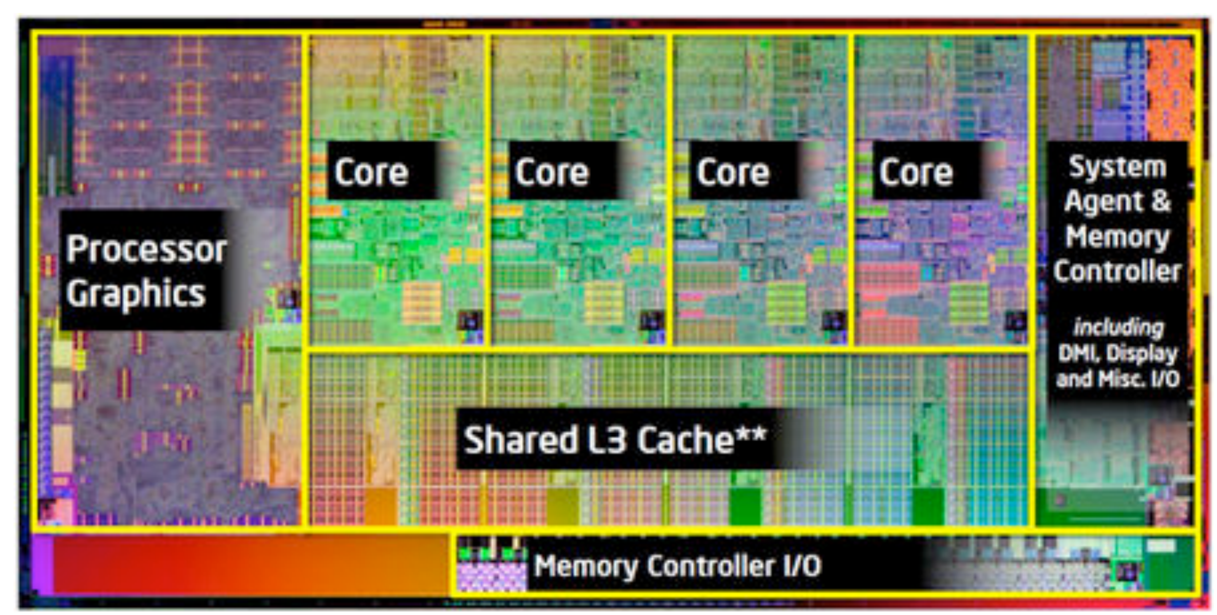
Google

http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/en//people/jeff/stanford-295-talk.pdf

CPU

- Multiple cores (4-8)
- Multiple sockets (1-4) per board
- 2-4 GHz clock
- 10-100W power
- Several cache levels (hierarchical, 8-16MB total)
- Vector processing units (SSE4, AVX) http://software.intel.com/en-us/avx/
- Perform several operations at once
- Use this for fast linear algebra (4-8 multiply adds in one operation)
- Memory interface 20-40GB/s
- Internal bandwidth >100GB/s
- 100+ GFlops for matrix matrix multiply
- Integrated low end GPU





RAM

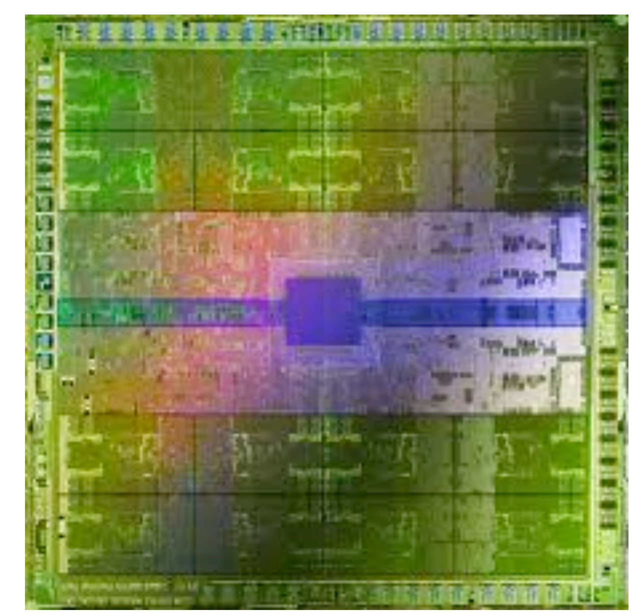
- 2-4 channels (32 bit wide)
- 1GHz speed
- High latency (10ns for DDR3)
- High burst data rate (>10 GB/s)

- Avoid random access in code if possible.
- Memory align variables
- Know your platform (FBDIMM vs. DDR)
  (code may run faster on old MacBookPro than a Xeon)

http://www.anandtech.com/show/3851/everything-you-always-wanted-to-know-about-sdram-memory-but-were-afraid-to-ask

GPU

- Up to 512 cores / 200W
- Cores have hierarchical structure tricky to synchronize threads (interrupts, semaphores, etc.)
- 1-3GB memory (Tesla 6GB)
- 1 TFlop (single precision)
- Memory bandwidth > 100GB/s
- 4GB/s PCI bus bottleneck

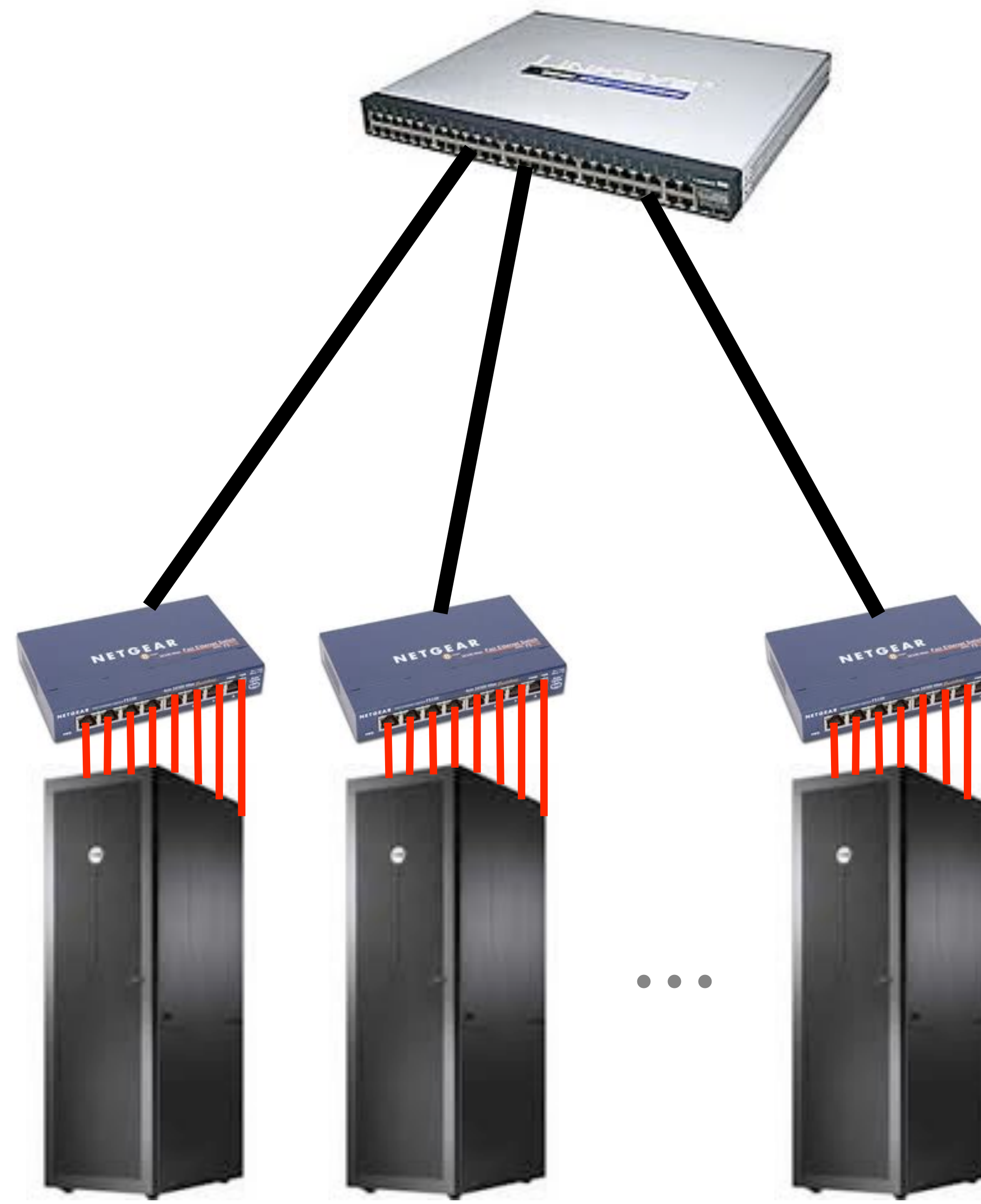


Storage

- Harddisks
  - 3TB of storage (30GB/$)
  - 100 MB/s bandwidth (sequential)
  - 5 ms seek (200 IOPS)
  - cheap
- SSD
  - 100-500 MB storage (1GB/$)
  - 300 MB/s bandwidth (sequential)
  - 50,000 IOPS / 1 ms seek (queueing)
  - Random writes often faster than reads
  - reliable (but limited lifetime - NAND)

Switches & Colos

- In theory perfect point to point bandwidth (e.g. 1Gb Ethernet)
- Big switches are expensive (crossbar bandwidth linear in #ports, price superlinear)
- Real switches have finite buffers
  - many connections to a single machine bad
  - buffer overflow / dropped packets / collision avoidance
- Hierarchical structure
  - more bandwidth within rack
  - lower latency within rack
  - lots of latency between colos
- Hadoop gives you machines where the data is (not necessarily on same rack!)



Numbers Everyone Should Know

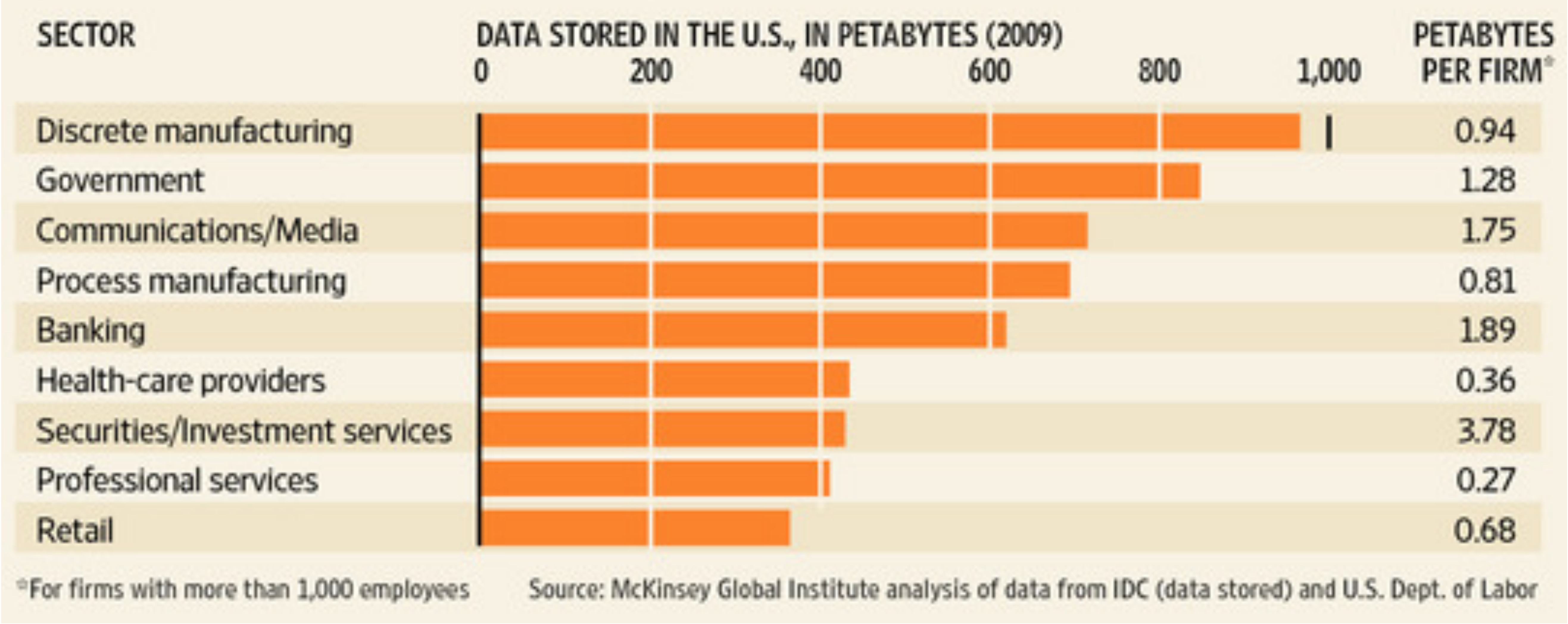
L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	100 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	10,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from network	10,000,000 ns
Read 1 MB sequentially from disk	30,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

Slide from talk of Jeff Dean

http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/en//people/jeff/stanford-295-talk.pdf

1.2 Data

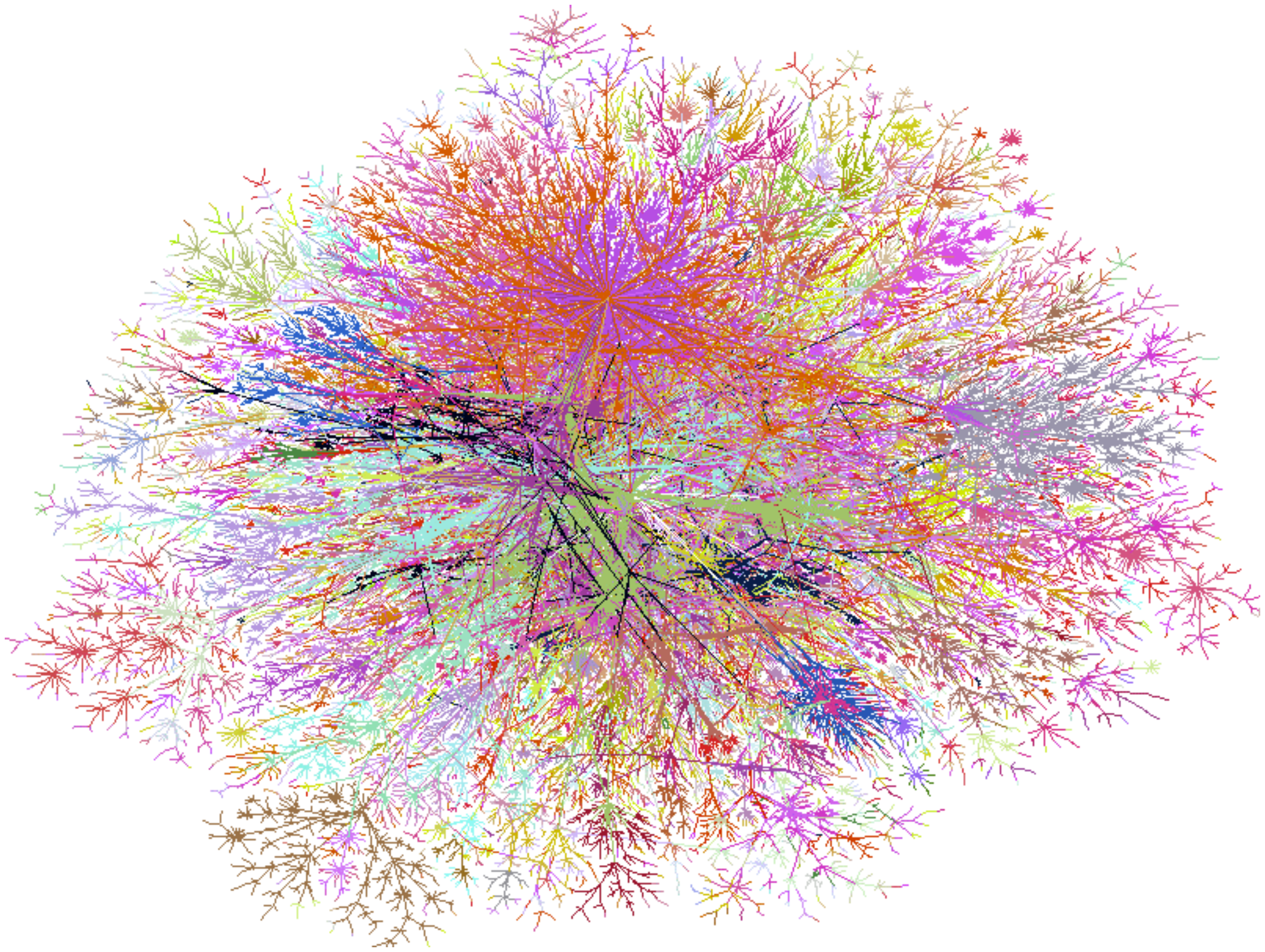
Big Data



we need Big Learning

Data

- Webpages (content, graph)
- Clicks (ad, page, social)
- Users (OpenID, FB Connect)
- e-mails (Hotmail, Y!Mail, Gmail)
- Photos, Movies (Flickr, YouTube, Vimeo ...)
- Cookies / tracking info (see Ghostery)
- Installed apps (Android market etc.)
- Location (Latitude, Loopt, Foursquared)
- User generated content (Wikipedia & co)
- Ads (display, text, DoubleClick, Yahoo)
- Comments (Disqus, Facebook)
- Reviews (Yelp, Y!Local)
- Third party features (e.g. Experian)
- Social connections (LinkedIn, Facebook)
- Purchase decisions (Netflix, Amazon)
- Instant Messages (YIM, Skype, Gtalk)
- Search terms (Google, Bing)
- Timestamp (everything)
- News articles (BBC, NYTimes, Y!News)
- Blog posts (Tumblr, Wordpress)
- Microblogs (Twitter, Jaiku, Meme)



>10B useful webpages

The Web for $100k/month

- Webpages (content, graph)
- Clicks (ad, page, social)
- Users (OpenID, FB Connect)
- e-mails (Hotmail, Y!Mail, Gmail)
- Photos, Movies (Flickr, YouTube, Vimeo ...)
- Cookies / tracking info (see Ghostery)
- Installed apps (Android market etc.)
- Location (Latitude, Loopt, Foursquared)
- User generated content (Wikipedia & co)
- Ads (display, text, DoubleClick, Yahoo)
- Comments (Disqus, Facebook)
- Reviews (Yelp, Y!Local)
- Third party features (e.g. Experian)
- Social connections (LinkedIn, Facebook)
- Purchase decisions (Netflix, Amazon)
- Instant Messages (YIM, Skype, Gtalk)
- Search terms (Google, Bing)
- Timestamp (everything)
- News articles (BBC, NYTimes, Y!News)
- Blog posts (Tumblr, Wordpress)
- Microblogs (Twitter, Jaiku, Meme)

- **10 billion pages**
  (this is a small subset, maybe 10%)
  10k/page = 100TB
  ($10k for disks or EBS 1 month)
- **1000 machines**
  10ms/page = 1 day
  afford 1-10 MIP/page
  ($20k on EC2 for 0.68$/h)
- **10 Gbit link**
  ($10k/month via ISP or EC2)
  - 1 day for raw data
  - 300ms/page roundtrip
  - 1000 servers for 1 month
    ($70k on EC2 for 0.085$/h)

Data - Identity & Graph

- Webpages (content, graph)
- Clicks (ad, page, social)
- Users (OpenID, FB Connect)
- e-mails (Hotmail, Y!Mail, Gmail)
- Photos, Movies (Flickr, YouTube, Vimeo ...)
- Cookies / tracking info (see Ghostery)
- Installed apps (Android market etc.)
- Location (Latitude, Loopt, Foursquared)
- User generated content (Wikipedia & co)
- Ads (display, text, DoubleClick, Yahoo)
- Comments (Disqus, Facebook)
- Reviews (Yelp, Y!Local)
- Third party features (e.g. Experian)
- Social connections (LinkedIn, Facebook)
- Purchase decisions (Netflix, Amazon)
- Instant Messages (YIM, Skype, Gtalk)
- Search terms (Google, Bing)
- Timestamp (everything)
- News articles (BBC, NYTimes, Y!News)
- Blog posts (Tumblr, Wordpress)
- Microblogs (Twitter, Jaiku, Meme)

100M-1B vertices

Crawling Twitter for $10k

- Webpages (content, graph)
- Clicks (ad, page, social)
- Users (OpenID, FB Connect)
- e-mails (Hotmail, Y!Mail, Gmail)
- Photos, Movies (Flickr, YouTube, Vimeo ...)
- Cookies / tracking info (see Ghostery)
- Installed apps (Android market etc.)
- Location (Latitude, Loopt, Foursquared)
- User generated content (Wikipedia & co)
- Ads (display, text, DoubleClick, Yahoo)
- Comments (Disqus, Facebook)
- Reviews (Yelp, Y!Local)
- Third party features (e.g. Experian)
- Social connections (LinkedIn, Facebook)
- Purchase decisions (Netflix, Amazon)
- Instant Messages (YIM, Skype, Gtalk)
- Search terms (Google, Bing)
- Timestamp (everything)
- News articles (BBC, NYTimes, Y!News)
- Blog posts (Tumblr, Wordpress)
- Microblogs (Twitter, Jaiku, Meme)

- 300M users
- Per user 300 queries/h
- 100 edges/query
- 100 edges/account
- Need 100 machines for 2 weeks (crawl it at 10 queries/s)
  - Tweets
  - Inlinks
  - Outlinks
- Cost
  - $3k for computers on EC2
  - Similar for network & storage
  - Need 10k user keys

Data - User generated content

- Webpages (content, graph)
- Clicks (ad, page, social)
- Users (OpenID, FB Connect)
- e-mails (Hotmail, Y!Mail, Gmail)
- Photos, Movies (Flickr, YouTube, Vimeo ...)
- Cookies / tracking info (see Ghostery)
- Installed apps (Android market etc.)
- Location (Latitude, Loopt, Foursquared)
- User generated content (Wikipedia & co)
- Ads (display, text, DoubleClick, Yahoo)
- Comments (Disqus, Facebook)
- Reviews (Yelp, Y!Local)
- Third party features (e.g. Experian)
- Social connections (LinkedIn, Facebook)
- Purchase decisions (Netflix, Amazon)
- Instant Messages (YIM, Skype, Gtalk)
- Search terms (Google, Bing)
- Timestamp (everything)
- News articles (BBC, NYTimes, Y!News)
- Blog posts (Tumblr, Wordpress)
- Microblogs (Twitter, Jaiku, Meme)

>1B images, 40h video/minute

Data - User generated content

- Webpages (content, graph)
- Clicks (ad, page, social)
- Users (OpenID, FB Connect)
- e-mails (Hotmail, Y!Mail, Gmail)
- Photos, Movies (Flickr, YouTube, Vimeo ...)
- Cookies / tracking info (see Ghostery)
- Installed apps (Android market etc.)
- Location (Latitude, Loopt, Foursquared)
- User generated content (Wikipedia & co)
- Ads (display, text, DoubleClick, Yahoo)
- Comments (Disqus, Facebook)
- Reviews (Yelp, Y!Local)
- Third party features (e.g. Experian)
- Social connections (LinkedIn, Facebook)
- Purchase decisions (Netflix, Amazon)
- Instant Messages (YIM, Skype, Gtalk)
- Search terms (Google, Bing)
- Timestamp (everything)
- News articles (BBC, NYTimes, Y!News)
- Blog posts (Tumblr, Wordpress)
- Microblogs (Twitter, Jaiku, Meme)

crawl it

>1B images, 40h video/minute

Data - Messages

- Webpages (content, graph)
- Clicks (ad, page, social)
- Users (OpenID, FB Connect)
- e-mails (Hotmail, Y!Mail, Gmail)
- Photos, Movies (Flickr, YouTube, Vimeo ...)
- Cookies / tracking info (see Ghostery)
- Installed apps (Android market etc.)
- Location (Latitude, Loopt, Foursquared)
- User generated content (Wikipedia & co)
- Ads (display, text, DoubleClick, Yahoo)
- Comments (Disqus, Facebook)
- Reviews (Yelp, Y!Local)
- Third party features (e.g. Experian)
- Social connections (LinkedIn, Facebook)
- Purchase decisions (Netflix, Amazon)
- Instant Messages (YIM, Skype, Gtalk)
- Search terms (Google, Bing)
- Timestamp (everything)
- News articles (BBC, NYTimes, Y!News)
- Blog posts (Tumblr, Wordpress)
- Microblogs (Twitter, Jaiku, Meme)

>1B texts

Data - Messages

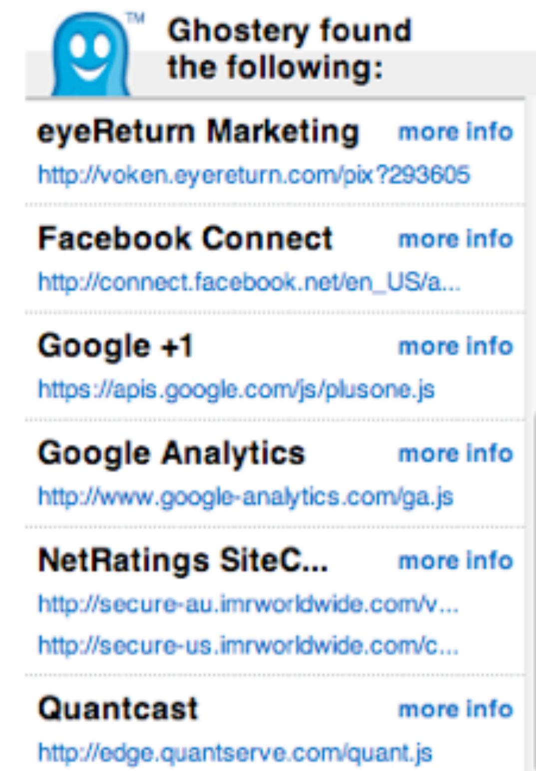
- Webpages (content, graph)
- Clicks (ad, page, social)
- Users (OpenID, FB Connect)
- e-mails (Hotmail, Y!Mail, Gmail)
- Photos, Movies (Flickr, YouTube, Vimeo ...)
- Cookies / tracking info (see Ghostery)
- Installed apps (Android market etc.)
- Location (Latitude, Loopt, Foursquared)
- User generated content (Wikipedia & co)
- Ads (display, text, DoubleClick, Yahoo)
- Comments (Disqus, Facebook)
- Reviews (Yelp, Y!Local)
- Third party features (e.g. Experian)
- Social connections (LinkedIn, Facebook)
- Purchase decisions (Netflix, Amazon)
- Instant Messages (YIM, Skype, Gtalk)
- Search terms (Google, Bing)
- Timestamp (everything)
- News articles (BBC, NYTimes, Y!News)
- Blog posts (Tumblr, Wordpress)
- Microblogs (Twitter, Jaiku, Meme)

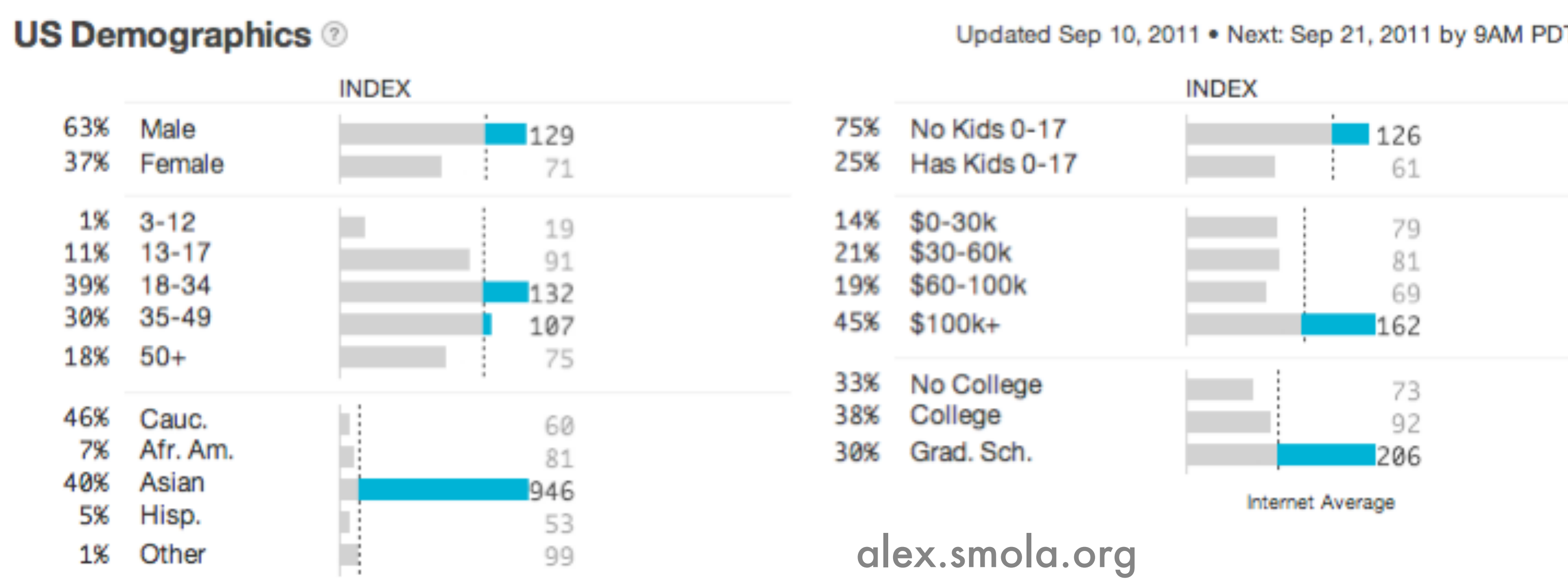
>1B texts

impossible without NDA

Data - User Tracking

- Webpages (content, graph)
- Clicks (ad, page, social)
- Users (OpenID, FB Connect)
- e-mails (Hotmail, Y!Mail, Gmail)
- Photos, Movies (Flickr, YouTube, Vimeo ...)
- Cookies / tracking info (see Ghostery)
- Installed apps (Android market etc.)
- Location (Latitude, Loopt, Foursquared)
- User generated content (Wikipedia & co)
- Ads (display, text, DoubleClick, Yahoo)
- Comments (Disqus, Facebook)
- Reviews (Yelp, Y!Local)
- Third party features (e.g. Experian)
- Social connections (LinkedIn, Facebook)
- Purchase decisions (Netflix, Amazon)
- Instant Messages (YIM, Skype, Gtalk)
- Search terms (Google, Bing)
- Timestamp (everything)
- News articles (BBC, NYTimes, Y!News)
- Blog posts (Tumblr, Wordpress)
- Microblogs (Twitter, Jaiku, Meme)





alex.smola.org

>1B ‘identities’

Data - User Tracking

- Webpages (content, graph)
- Clicks (ad, page, social)
- Users (OpenID, FB Connect)
- e-mails (Hotmail, Y!Mail, Gmail)
- Photos, Movies (Flickr, YouTube, Vimeo ...)
- Cookies / tracking info (see Ghostery)
- Installed apps (Android market etc.)
- Location (Latitude, Loopt, Foursquared)
- User generated content (Wikipedia & co)
- Ads (display, text, DoubleClick, Yahoo)
- Comments (Disqus, Facebook)
- Reviews (Yelp, Y!Local)
- Third party features (e.g. Experian)
- Social connections (LinkedIn, Facebook)
- Purchase decisions (Netflix, Amazon)
- Instant Messages (YIM, Skype, Gtalk)
- Search terms (Google, Bing)
- Timestamp (everything)
- News articles (BBC, NYTimes, Y!News)
- Blog posts (Tumblr, Wordpress)
- Microblogs (Twitter, Jaiku, Meme)

Privacy Information

Privacy Policy:

http://www.facebook.com/policy.php

Data Collected:

Anonymous (browser type, location, page views), Pseudonymous (IP address, "actions taken")

Data Sharing:

Data is shared with third parties.

Data Retention:

Data is deleted from backup storage after 90 days.

Privacy Information

Privacy Policy:

http://www.google.com/intl/en/priv...

Data Collected:

Anonymous (ad serving domains, browser type, demographics, language settings, page views, time/date), Pseudonymous (IP address)

Data Sharing:

Anonymous data is shared with third parties.

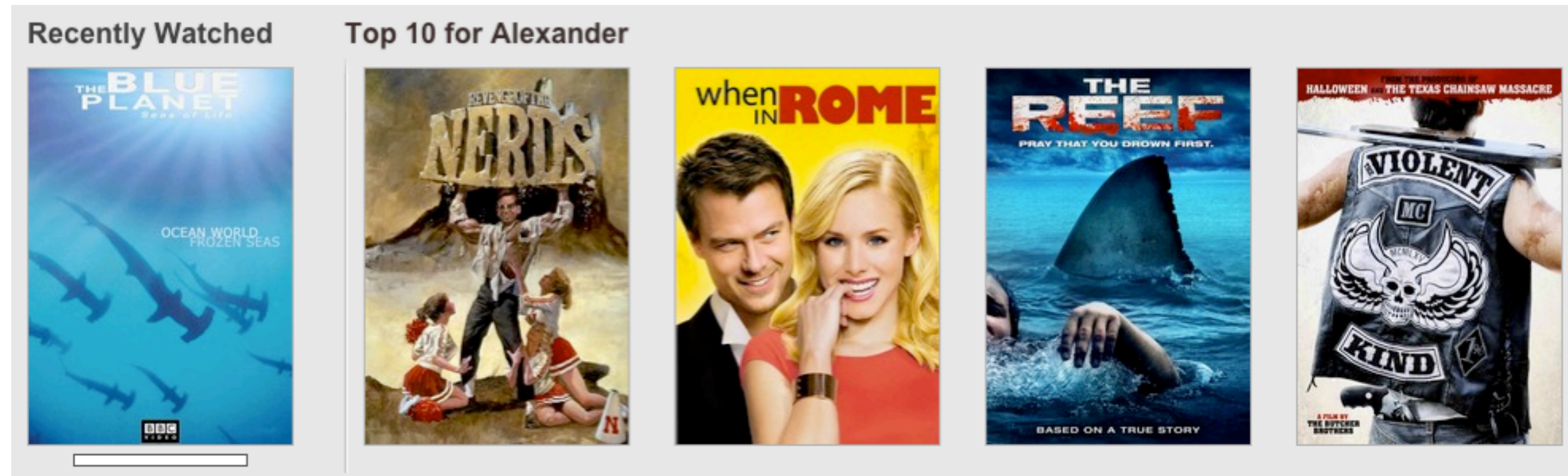
Data Retention:

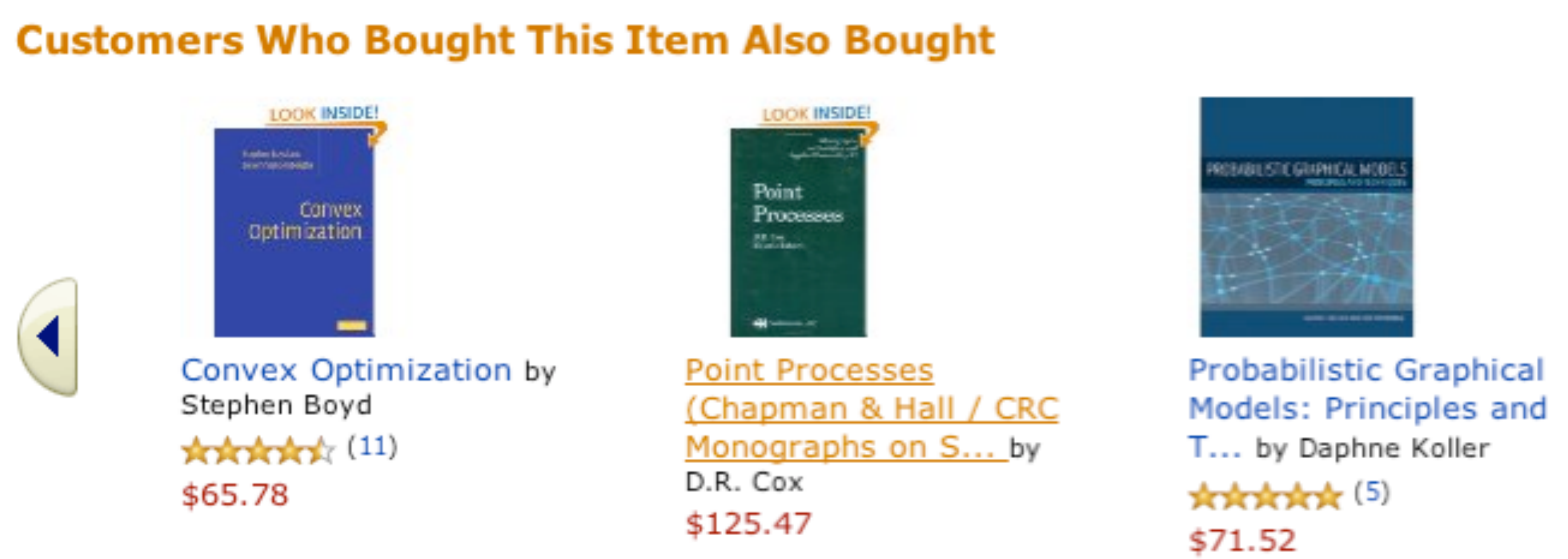
Undisclosed

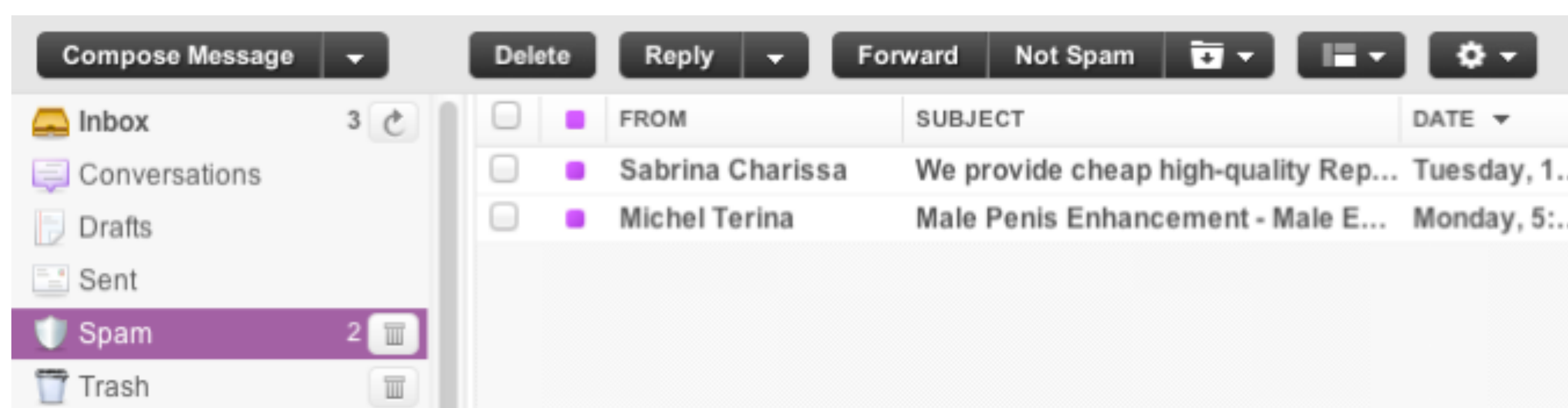
Personalization

- 100-1000M users
  - Spam filtering
  - Personalized targeting & collaborative filtering
  - News recommendation
  - Advertising

- Large parameter space (25 parameters = 100GB)
- Distributed storage (need it on every server)
- Distributed optimization
- Model synchronization
- Time dependence
- Graph structure



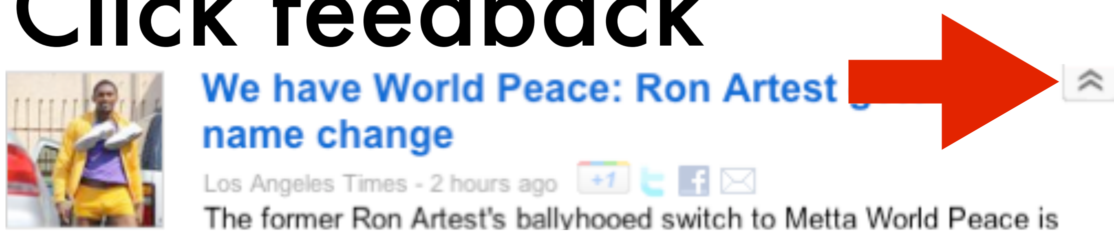




(implicit) Labels   no Labels

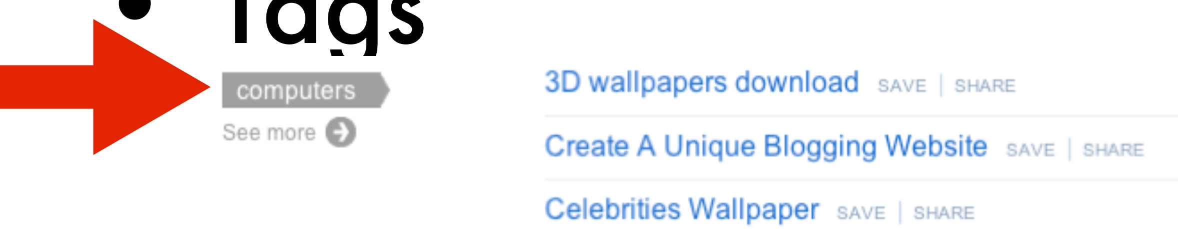
- Ads

- Click feedback



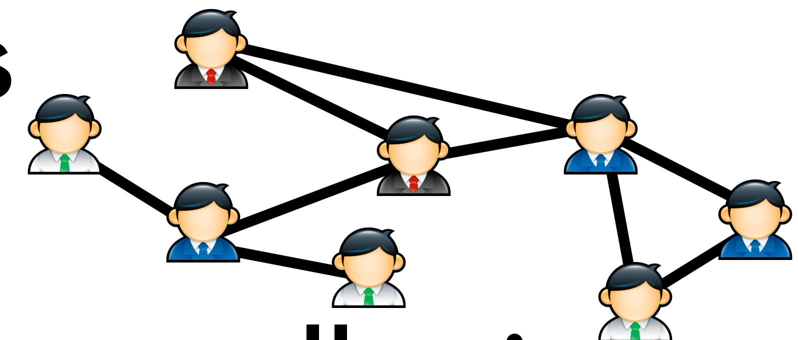
- Emails

- Tags



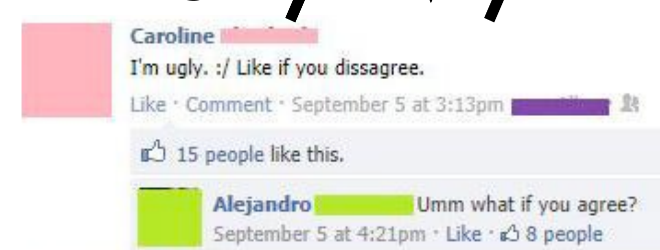
- Editorial data is very expensive! Do not use!

- Graphs

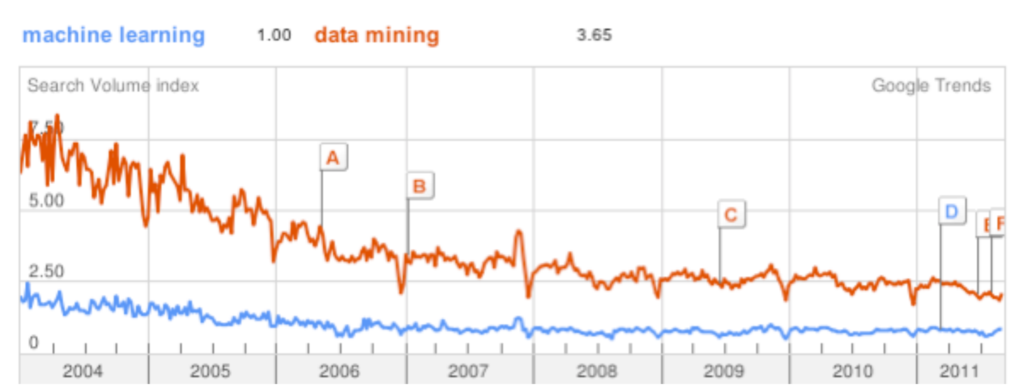


- Document collections

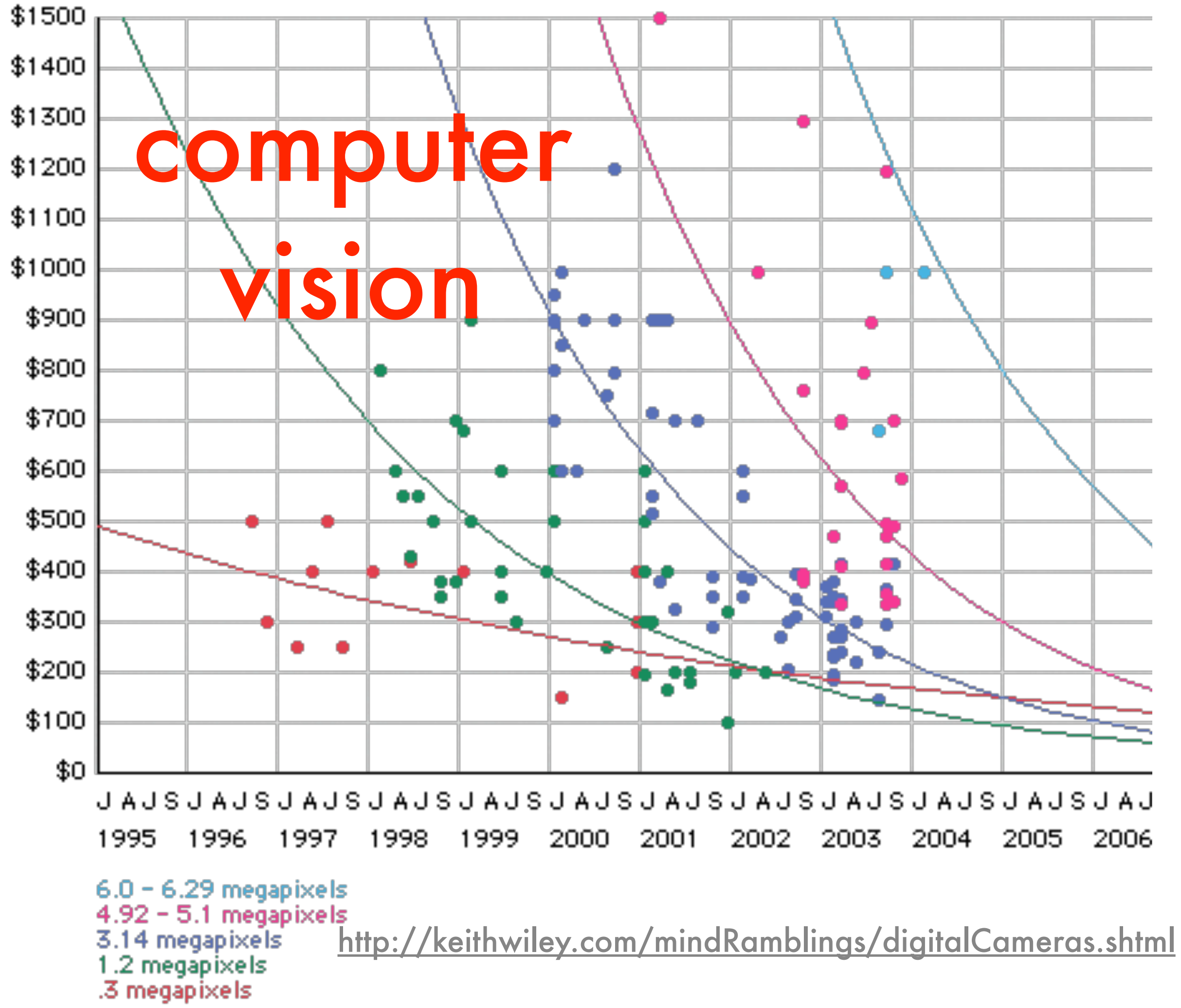
- Email/IM/Discussions

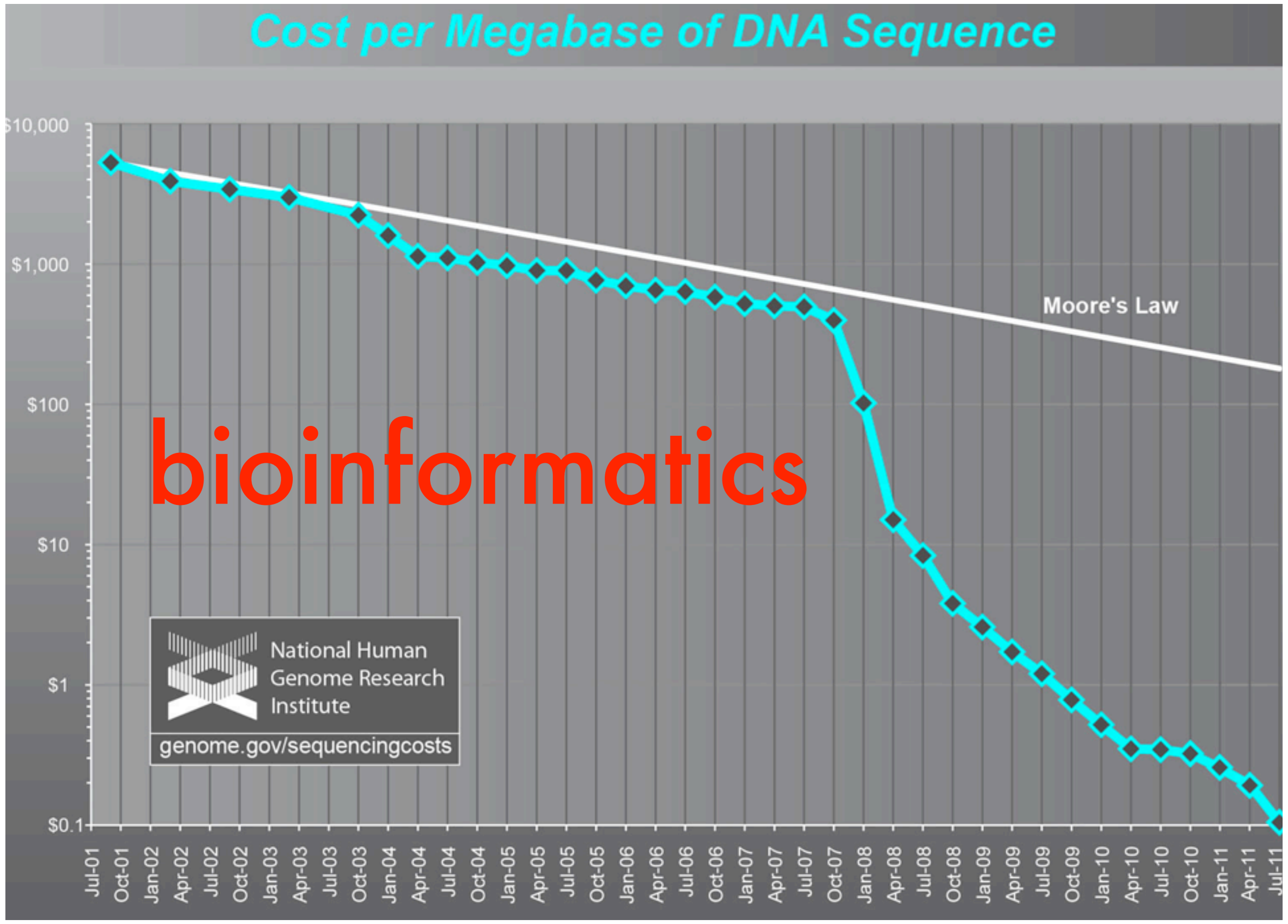


- Query stream



Many more sources



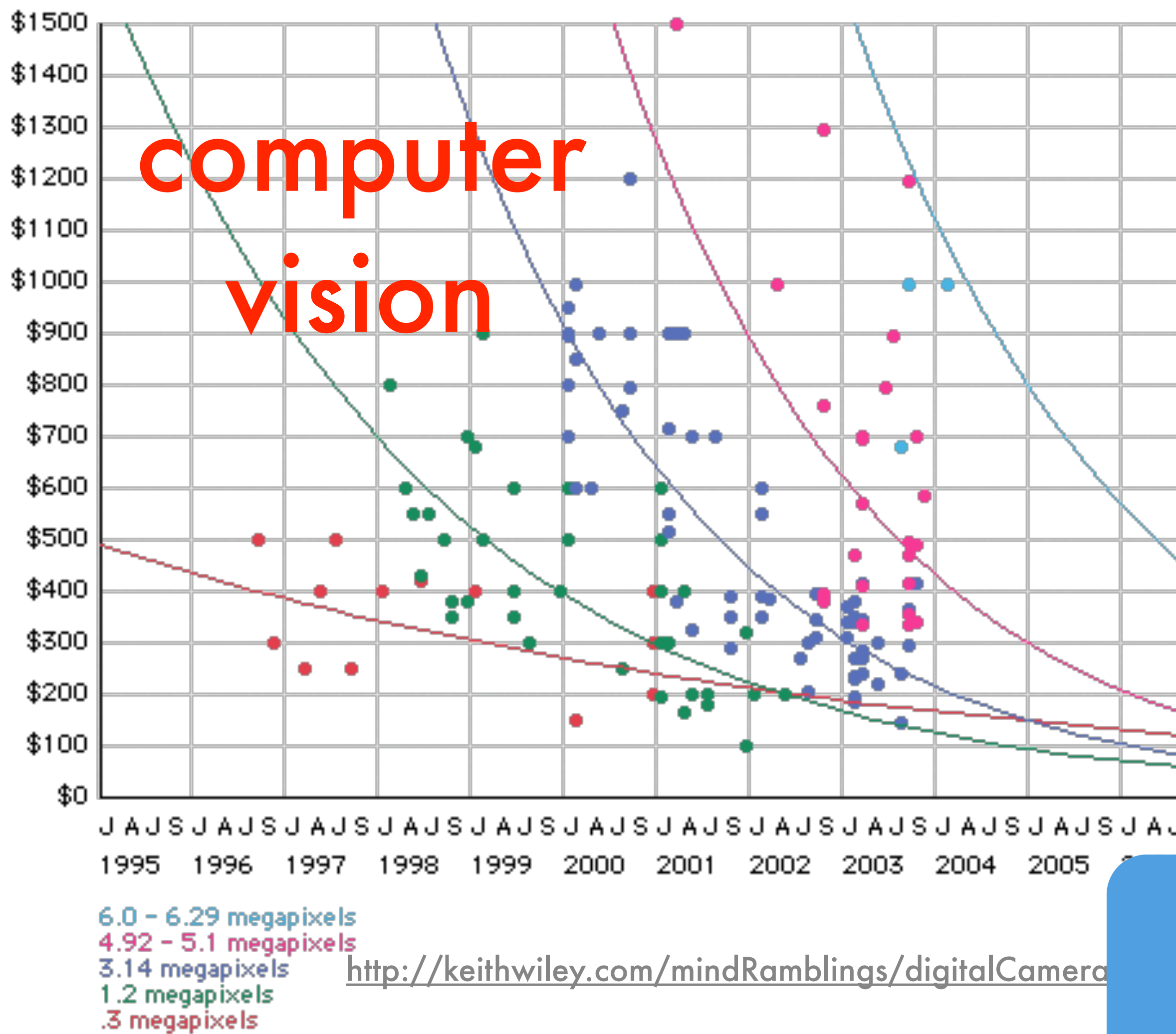


personalized sensors

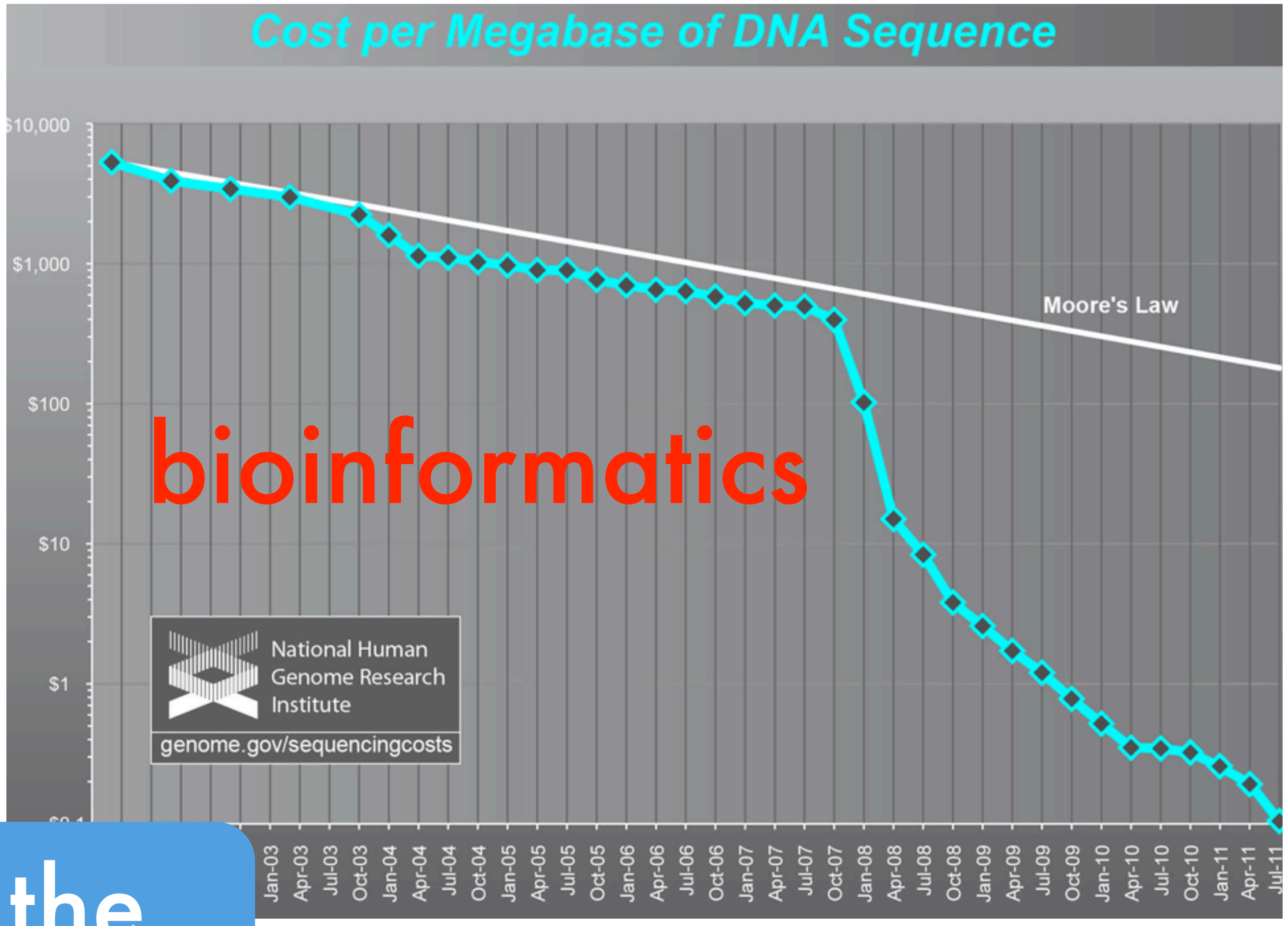
ubiquitous control

Many more sources

computer vision



bioinformatics



in the cloud

personalized sensors

ubiquitous control

1.3 Distribution Strategies

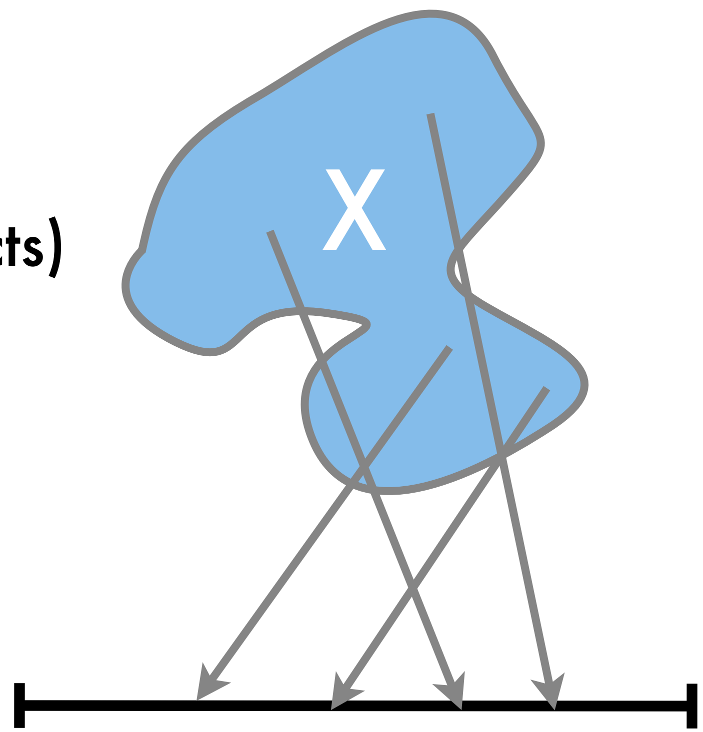
Concepts

- Variable and load distribution
  - Large number of objects (a priori unknown)
  - Large pool of machines (often faulty)
  - Assign objects to machines such that
    - Object goes to the same machine (if possible)
    - Machines can be added/fail dynamically
  - Consistent hashing (elements, sets, proportional)
- Overlay networks (peer to peer routing)
  - Location of object is unknown, find route
  - Store object redundantly / anonymously

symmetric (no master), dynamically scalable, fault tolerant

Hash functions

- Mapping h from domain X to integer range $[1, \ldots N]$
- Goal
  - We want a uniform distribution (e.g. to distribute objects)
- Naive Idea
  - For each new x, compute random h(x)
  - Store it in big lookup table
  - Perfectly random
  - Uses lots of memory (value, index structure)
  - Gets slower the more we use it
  - Cannot be merged between computers
- Better Idea
  - Use random number generator with seed x
  - As random as the random number generator might be ...
  - No memory required
  - Can be merged between computers
  - Speed independent of number of hash calls



Hash function

- n-ways independent hash function
  - Set of hash functions H
  - Draw h from H at random
  - For n instances in X their hash $[h(x_1), \ldots h(x_n)]$ is essentially indistinguishable from n random draws from [1 ... N]
- For a formal treatment see Maurer 1992 (incl. permutations) ftp://ftp.inf.ethz.ch/pub/crypto/publications/Maurer92d.pdf
- For many cases we only need 2-ways independence (harder proof)

$$\text{for all } x, y \quad \Pr_{y \in H} \{h(x) = h(y)\} = \frac{1}{N}$$

- In practice use MD5 or Murmur Hash for high quality https://code.google.com/p/smhasher/
- Fast linear congruential generator $ax + b \bmod c$ for constants a, b, c see http://en.wikipedia.org/wiki/Linear_congruential_generator

1.3.1 Load Distribution

D1 - Argmin Hash

- Consistent hashing

$$m(\text{key}) = \operatorname*{argmin}_{m \in \mathcal{M}} h(\text{key}, m)$$

  - Uniform distribution over machine pool M
  - Fully determined by hash function h. No need to ask master
  - If we add/remove machine m′ all but O(1/m) keys remain

$$\Pr\left\{m(\text{key}) = m'\right\} = \frac{1}{m}$$

- Consistent hashing with k replications

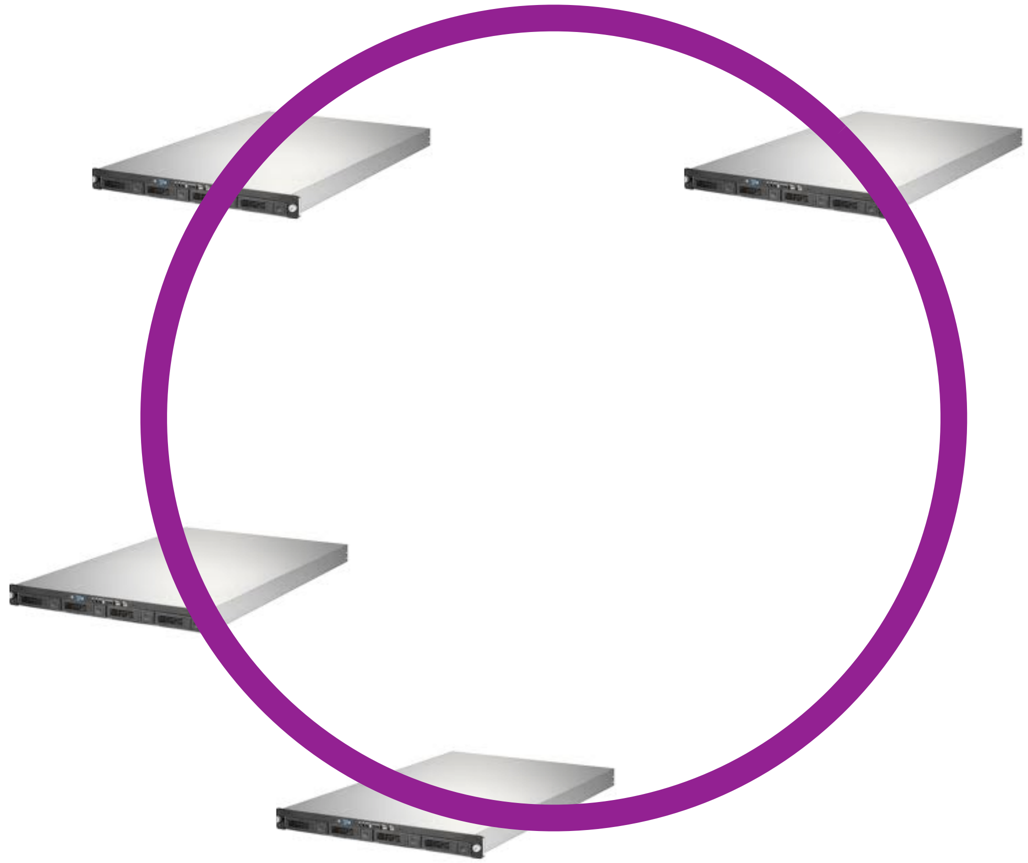
$$m(\text{key}, k) = k \operatorname*{smallest}_{m \in \mathcal{M}} h(\text{key}, m)$$

  - If we add/remove a machine only O(k/m) need reassigning
- Cost to assign is O(m). This can be expensive for 1000 servers

D2 - Distributed Hash Table

- Fixing the O(m) lookup
  - Assign machines to ring via hash h(m)
  - Assign keys to ring
  - Pick machine nearest to key to the left
- O(log m) lookup
- Insert/removal only affects neighbor (however, big problem for neighbor)
- Uneven load distribution (load depends on segment size)
- Insert machine more than once to fix this
- For k term replication, simply pick the k leftmost machines (skip duplicates)

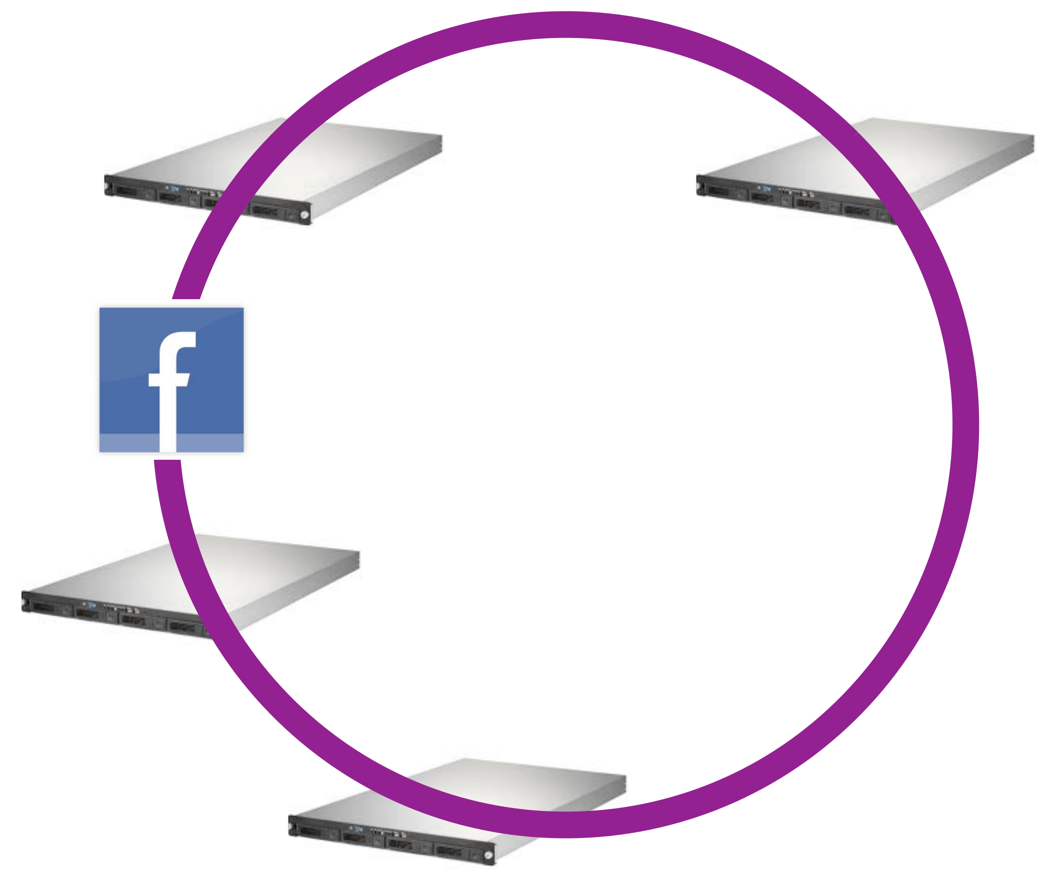
ring of N keys



D2 - Distributed Hash Table

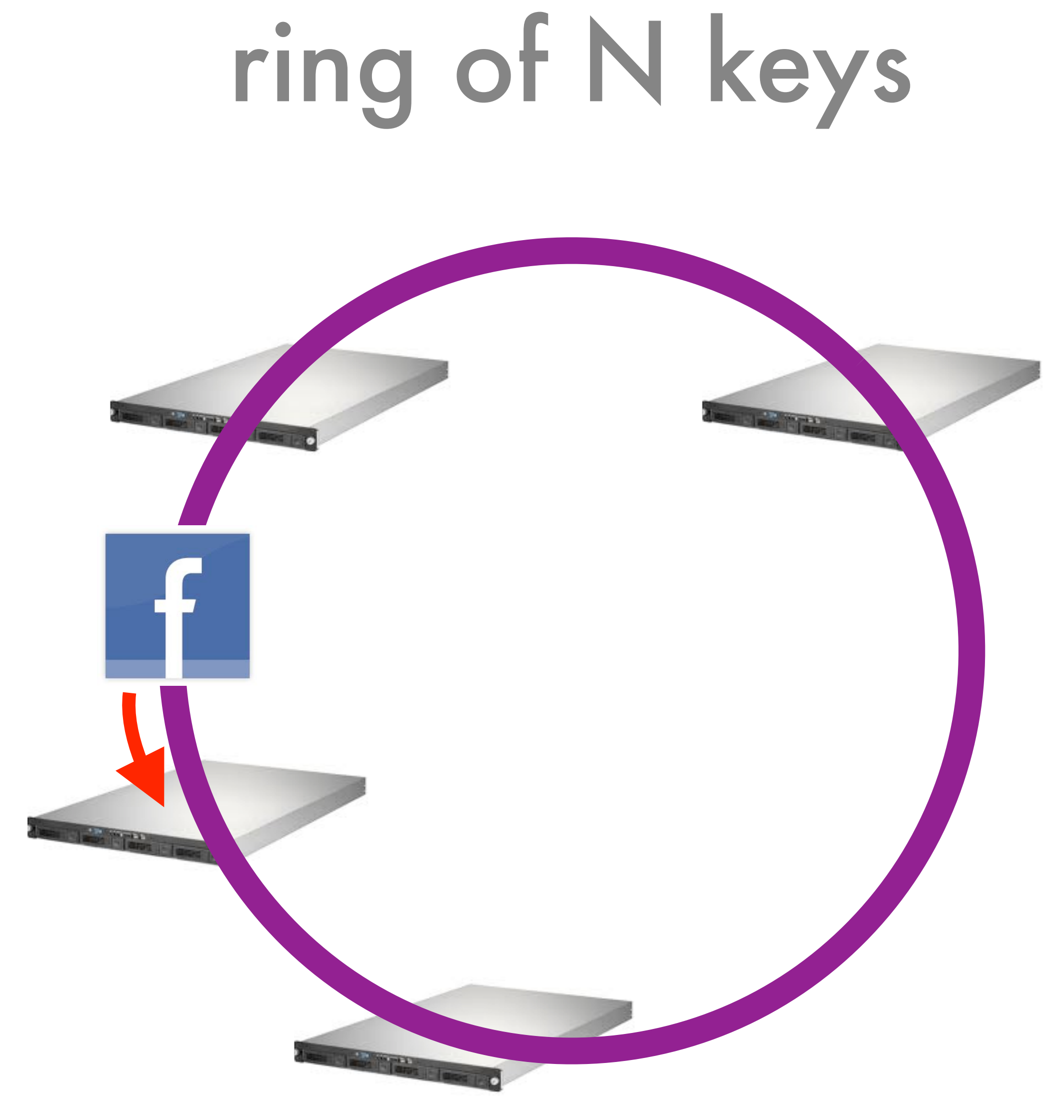
- Fixing the O(m) lookup
  - Assign machines to ring via hash h(m)
  - Assign keys to ring
  - Pick machine nearest to key to the left
- O(log m) lookup
- Insert/removal only affects neighbor (however, big problem for neighbor)
- Uneven load distribution (load depends on segment size)
- Insert machine more than once to fix this
- For k term replication, simply pick the k leftmost machines (skip duplicates)

ring of N keys



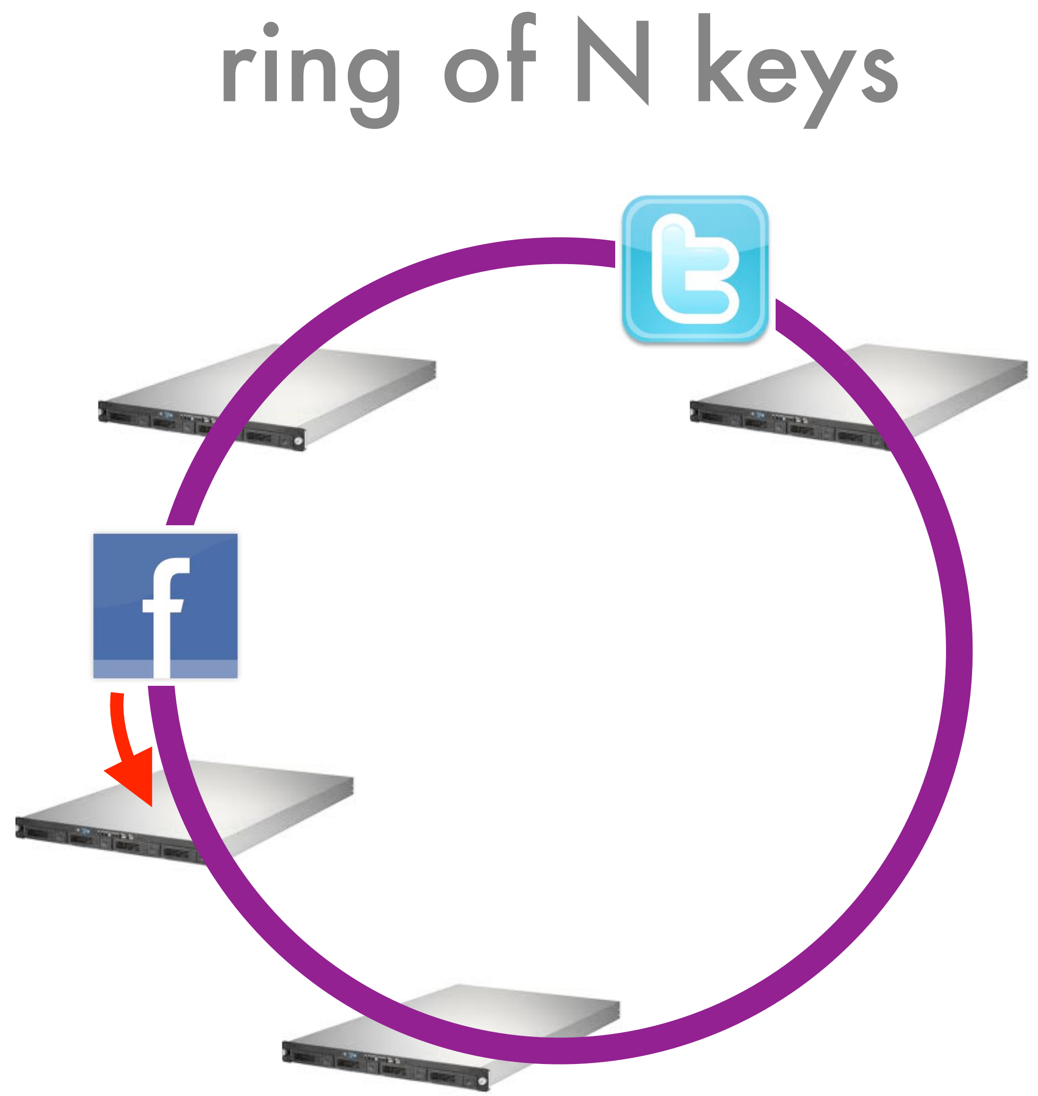
D2 - Distributed Hash Table

- Fixing the O(m) lookup
  - Assign machines to ring via hash h(m)
  - Assign keys to ring
  - Pick machine nearest to key to the left
- O(log m) lookup
- Insert/removal only affects neighbor
  (however, big problem for neighbor)
- Uneven load distribution
  (load depends on segment size)
- Insert machine more than once to fix this
- For k term replication, simply pick the k leftmost machines (skip duplicates)



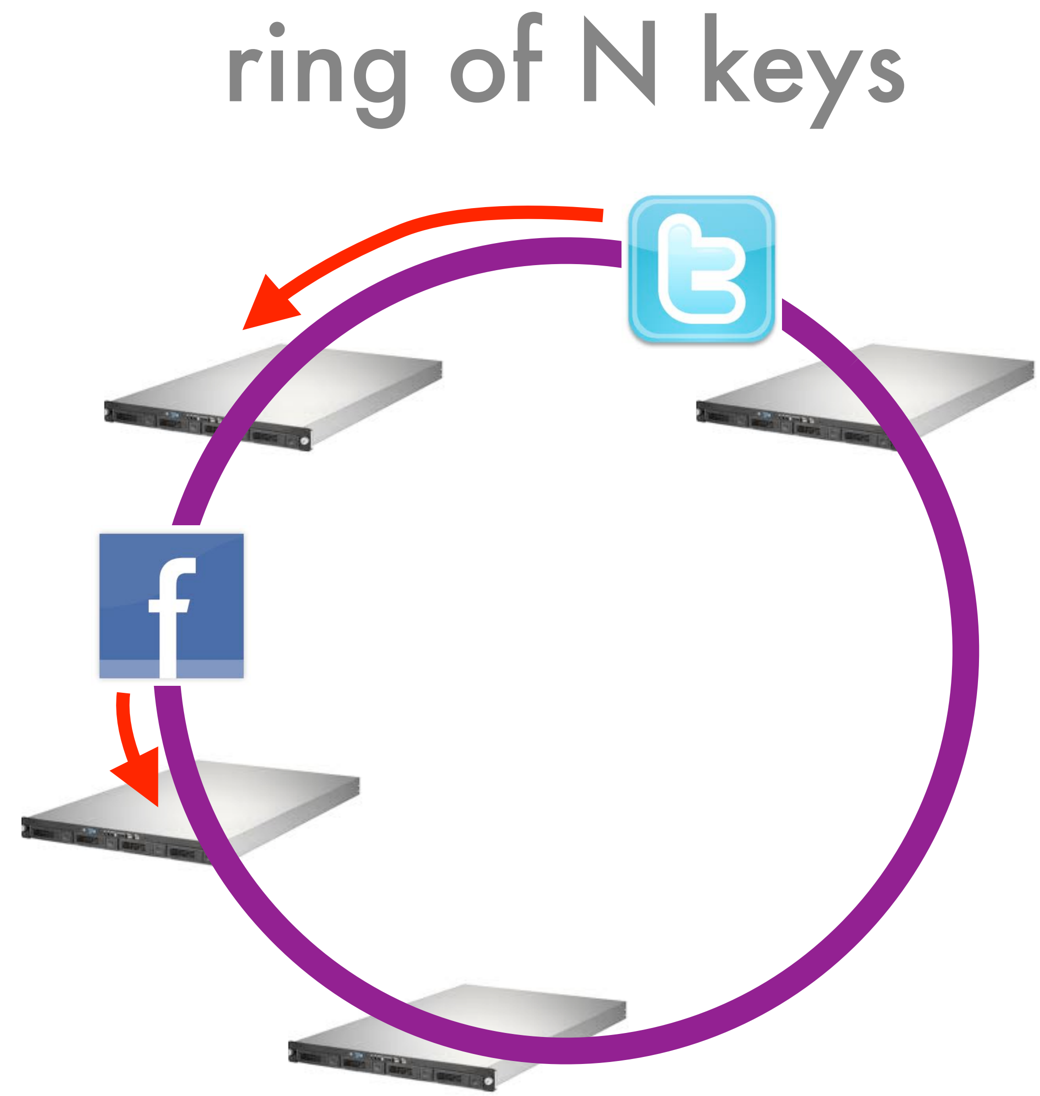
D2 - Distributed Hash Table

- Fixing the O(m) lookup
  - Assign machines to ring via hash h(m)
  - Assign keys to ring
  - Pick machine nearest to key to the left
- O(log m) lookup
- Insert/removal only affects neighbor
  (however, big problem for neighbor)
- Uneven load distribution
  (load depends on segment size)
- Insert machine more than once to fix this
- For k term replication, simply pick the k leftmost machines (skip duplicates)



D2 - Distributed Hash Table

- Fixing the O(m) lookup
  - Assign machines to ring via hash h(m)
  - Assign keys to ring
  - Pick machine nearest to key to the left
- O(log m) lookup
- Insert/removal only affects neighbor
  (however, big problem for neighbor)
- Uneven load distribution
  (load depends on segment size)
- Insert machine more than once to fix this
- For k term replication, simply pick the k leftmost machines (skip duplicates)



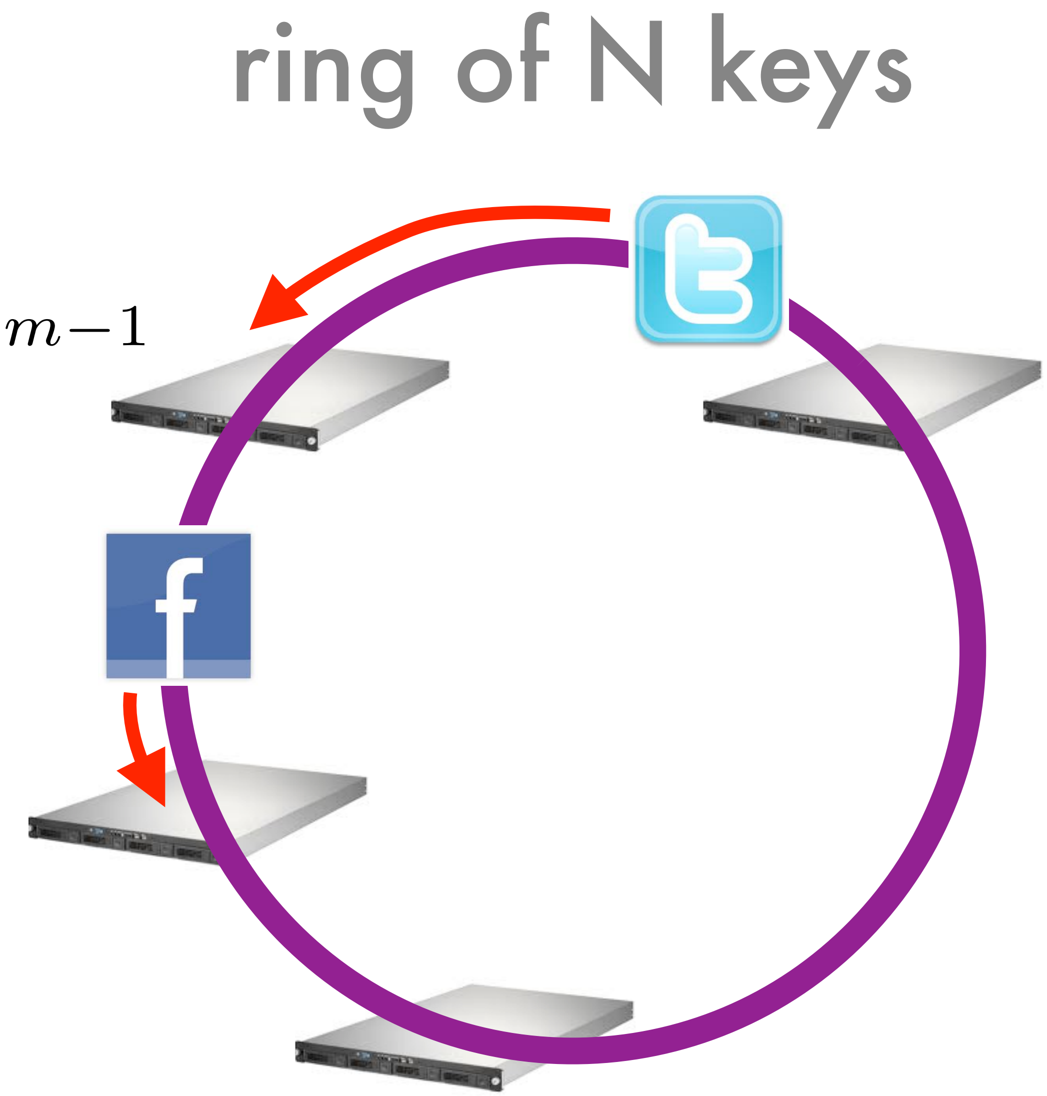
D2 - Distributed Hash Table

- For arbitrary node segment size is minimum over (m-1) independent uniformly distributed random variables

$$\Pr\{x \geq c\} = \prod_{i=2}^{m} \Pr\{s_i \geq c\} = (1-c)^{m-1}$$

- Density is given by derivative
$p(c) = (m-1)(1-c)^{m-2}$
- Expected segment length is $c = \frac{1}{m}$
(follows from symmetry)
- Probability of exceeding expected segment length (for large m)

$$\Pr\left\{x \geq \frac{k}{m}\right\} = \left(1 - \frac{k}{m}\right)^{m-1} \longrightarrow e^{-k}$$

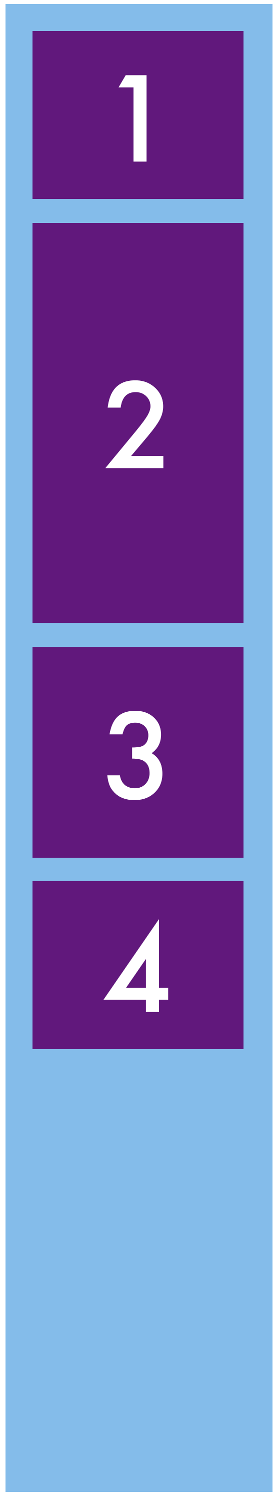


D3 - Proportional Allocation Table

- Assign items according to machine capacity

- Create allocation table with segments proportional to capacity
- Leave space for additional machines
- Hash key h(x) and pick machine covering it
- If failure, re-hash the hash until it hits a bin
- For replication hit k bins in a row

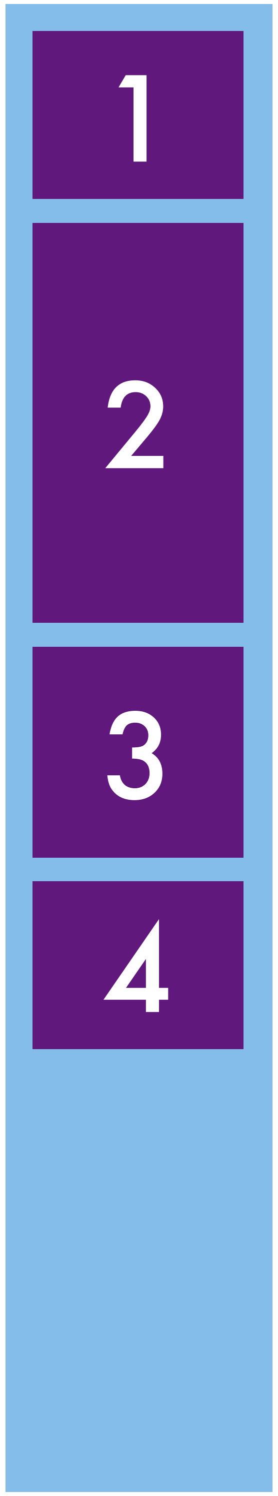
- Proportional load distribution
- Limited scalability
- Need to distribute and update table
- Limit peak load by further delegation (SPOCA - Chawla et al., USENIX 2011)



D3 - Proportional Allocation Table

- Assign items according to machine capacity

- Create allocation table with segments proportional to capacity
- Leave space for additional machines
- Hash key h(x) and pick machine covering it
- If failure, re-hash the hash until it hits a bin
- For replication hit k bins in a row



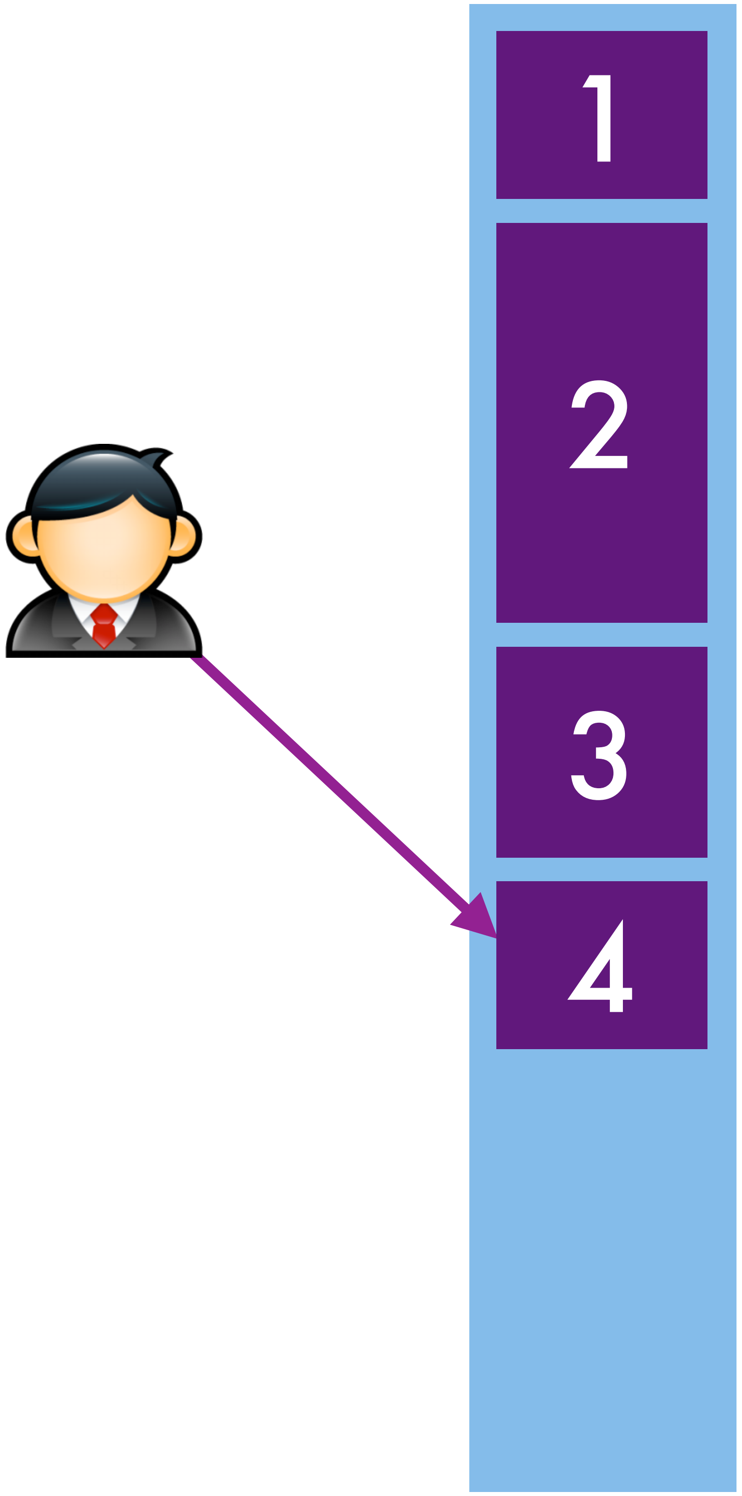
- Proportional load distribution
- Limited scalability
- Need to distribute and update table
- Limit peak load by further delegation (SPOCA - Chawla et al., USENIX 2011)

D3 - Proportional Allocation Table

- Assign items according to machine capacity

- Create allocation table with segments proportional to capacity
- Leave space for additional machines
- Hash key h(x) and pick machine covering it
- If failure, re-hash the hash until it hits a bin
- For replication hit k bins in a row

- Proportional load distribution
- Limited scalability
- Need to distribute and update table
- Limit peak load by further delegation (SPOCA - Chawla et al., USENIX 2011)

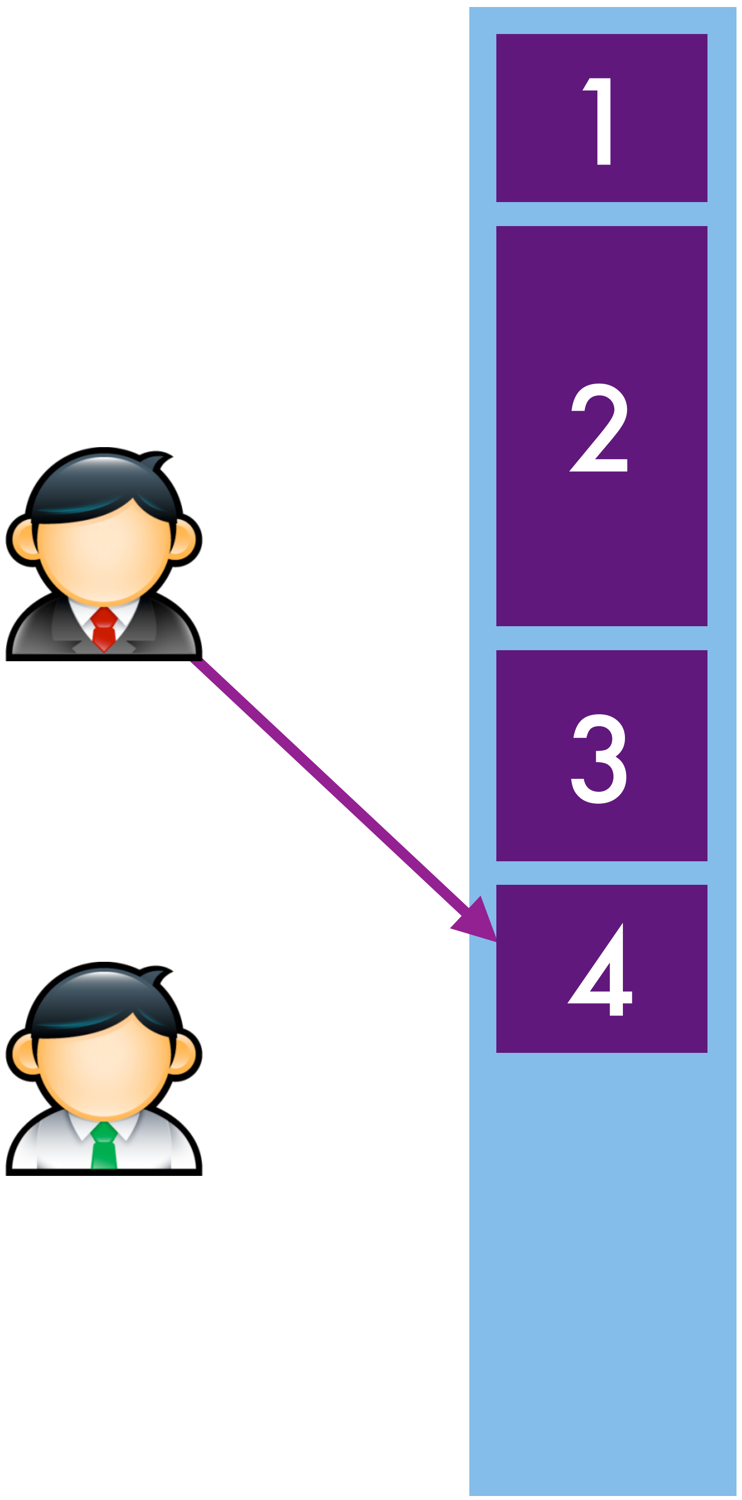


D3 - Proportional Allocation Table

- Assign items according to machine capacity

- Create allocation table with segments proportional to capacity
- Leave space for additional machines
- Hash key h(x) and pick machine covering it
- If failure, re-hash the hash until it hits a bin
- For replication hit k bins in a row

- Proportional load distribution
- Limited scalability
- Need to distribute and update table
- Limit peak load by further delegation (SPOCA - Chawla et al., USENIX 2011)

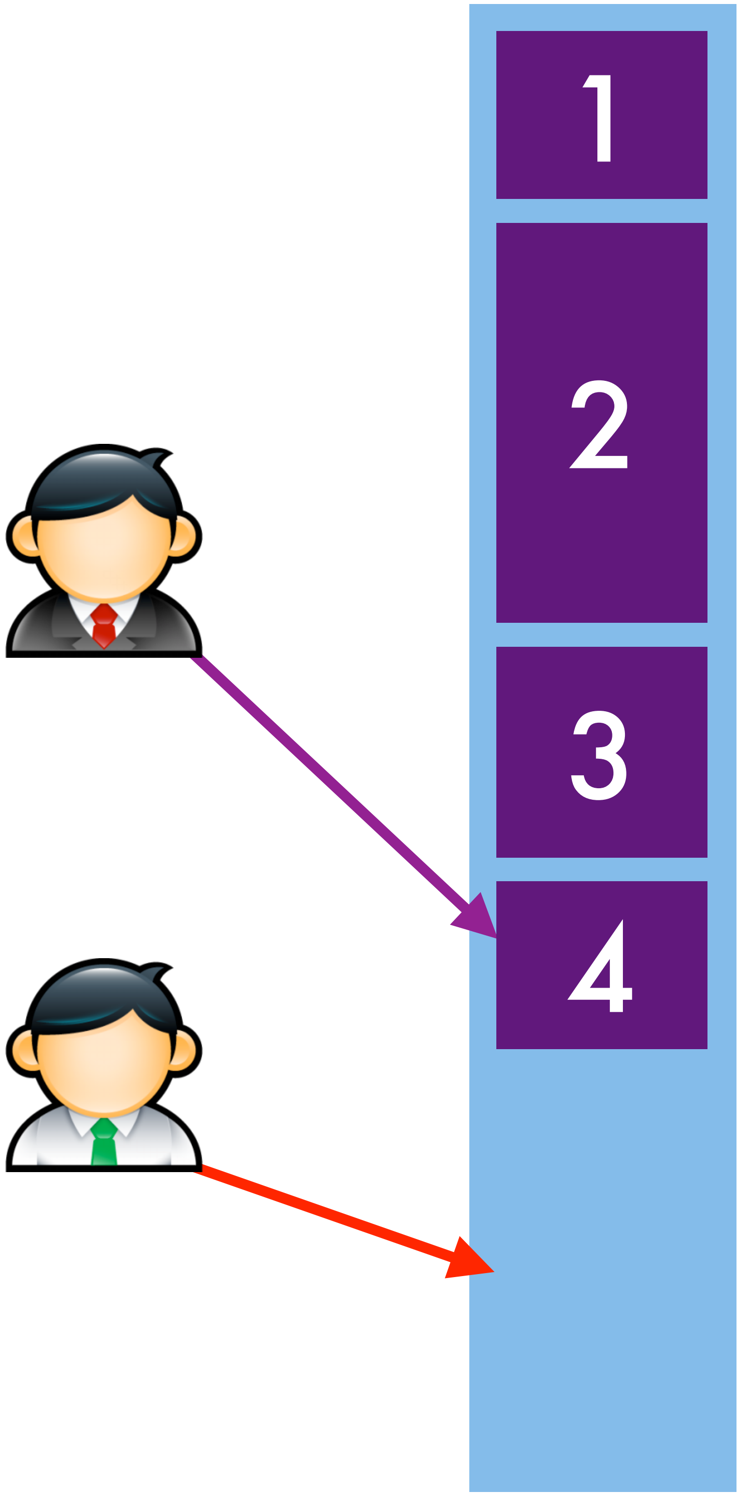


D3 - Proportional Allocation Table

- Assign items according to machine capacity

- Create allocation table with segments proportional to capacity
- Leave space for additional machines
- Hash key h(x) and pick machine covering it
- If failure, re-hash the hash until it hits a bin
- For replication hit k bins in a row

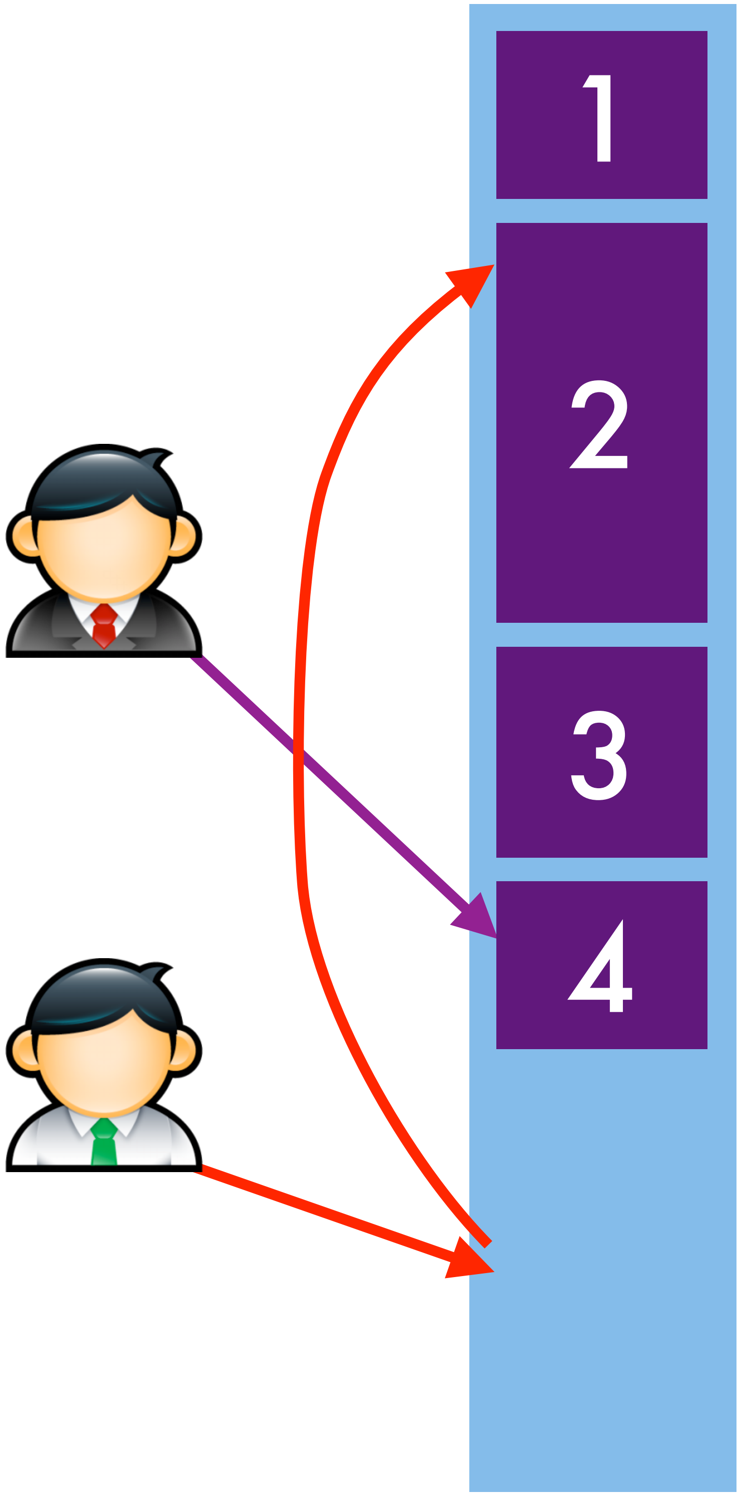
- Proportional load distribution
- Limited scalability
- Need to distribute and update table
- Limit peak load by further delegation (SPOCA - Chawla et al., USENIX 2011)



D3 - Proportional Allocation Table

- Assign items according to machine capacity
- Create allocation table with segments proportional to capacity
- Leave space for additional machines
- Hash key h(x) and pick machine covering it
- If failure, re-hash the hash until it hits a bin
- For replication hit k bins in a row

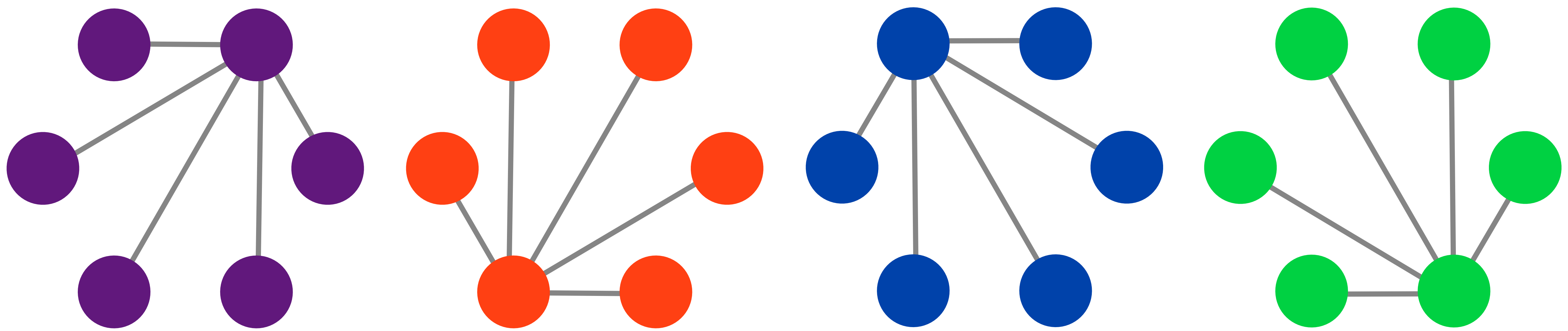
- Proportional load distribution
- Limited scalability
- Need to distribute and update table
- Limit peak load by further delegation (SPOCA - Chawla et al., USENIX 2011)



Random Caching Trees (Karger et al. 1999, Akamai paper)

- Cache / synchronize an object
- Uneven load distribution
- Must not generate hotspot

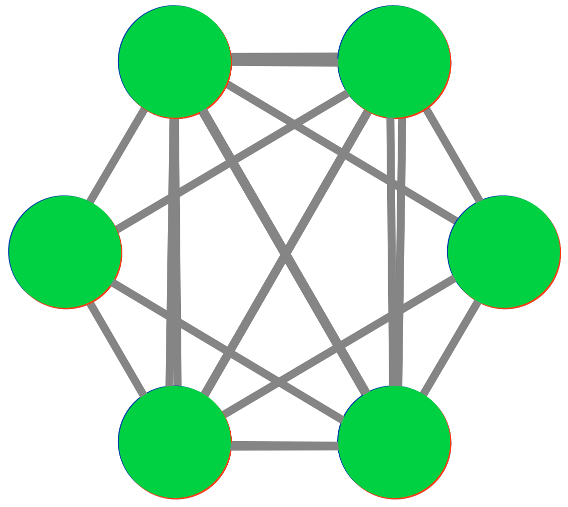
- For given key, pick random order of machines
- Map order onto tree / star via BFS ordering



Random Caching Trees

- Cache / synchronize an object
- Uneven load distribution
- Must not generate hotspot

- For given key, pick random order of machines
- Map order onto tree / star via BFS ordering



1.3.2. Overlay Networks & P2P

Peer to peer

- Large number of (unreliable) nodes
- Find objects in logarithmic time
- Overlay network (no TCP/IP replacement)
  - Logical communications network on top of physical network
  - Pick host to store object by finding machine with nearest hash
  - No need to know who has it to find it
    (route until nobody else is closer)
- Usage
  - Distributed object storage (file sharing)
    Store file on machine(s) k-nearest to key.
  - Load distribution / caching
    Route requests to nearest machines (only log N overhead).
  - Publish / subscribe service

Pastry (Rowstrom & Druschel)

- Node gets random ID (128 bit ensures that we're safe up to 2^{64} nodes)
- State table
  - L/2 left and right nearest nodes
  - Nodes within network neighborhood
  - For each prefix the 2^b neighbors with different digit (if they exist)
- Routing in log N steps for a key
  - Use nearest element in routing table
  - Send routing request there
  - If not available, use nearest element from leaf set

NodeId 10233102

Leaf set

	SMALLER	LARGER	
10233033	10233021	10233120	10233122
10233001	10233000	10233230	10233232

Routing table

-0-2212102	1	-2-2301203	-3-1203203
0	1-1-301233	1-2-230203	1-3-021022
10-0-31203	10-1-32102	2	10-3-23302
102-0-0230	102-1-1302	102-2-2302	3
1023-0-322	1023-1-000	1023-2-121	3
10233-0-01	1	10233-2-32	
0		102331-2-0	
		2	

Neighborhood set

13021022	10200230	11301233	31301233
02212102	22301203	31203203	33213321

Pastry (Rowstrom & Druschel)

- nodeId = pastryInit
  generates node ID, connect to net
- route(key,value)
  route message
- delivered(key,value)
  confirms message delivery
- forward(key,value,nextID)
  forwards to nextID, optionally modify value
- newLeaves(leafSet)
  notify application of new leaves, update routing table as needed

NodeId 10233102

Leaf set

SMALLER		LARGER	
10233033	10233021	10233120	10233122
10233001	10233000	10233230	10233232

Routing table

-0-2212102	1	-2-2301203	-3-1203203
0	1-1-301233	1-2-230203	1-3-021022
10-0-31203	10-1-32102	2	10-3-23302
102-0-0230	102-1-1302	102-2-2302	3
1023-0-322	1023-1-000	1023-2-121	3
10233-0-01	1	10233-2-32	
0		102331-2-0	
		2	

Neighborhood set

13021022	10200230	11301233	31301233
02212102	22301203	31203203	33213321

Pastry

- Add node
  - Generate key
  - Find route to nearest node
  - All nodes on route send routing table to new node
  - Compile routing table from messages
  - Send routing table back to nodes on path
- Nodes fail silently
- Update table
  - Prefer near nodes (hence the neighborhood set)
  - Repair when nodes fail (route to neighbors)
- Analysis
  - $O(\log_b N)$ nonempty rows in routing table
    (uniform key distribution, average distance is concentrated)
  - Tolerates up to L/2 local failures (very unlikely to happen) to recover network
  - Finding k nearest neighbors is nontrivial

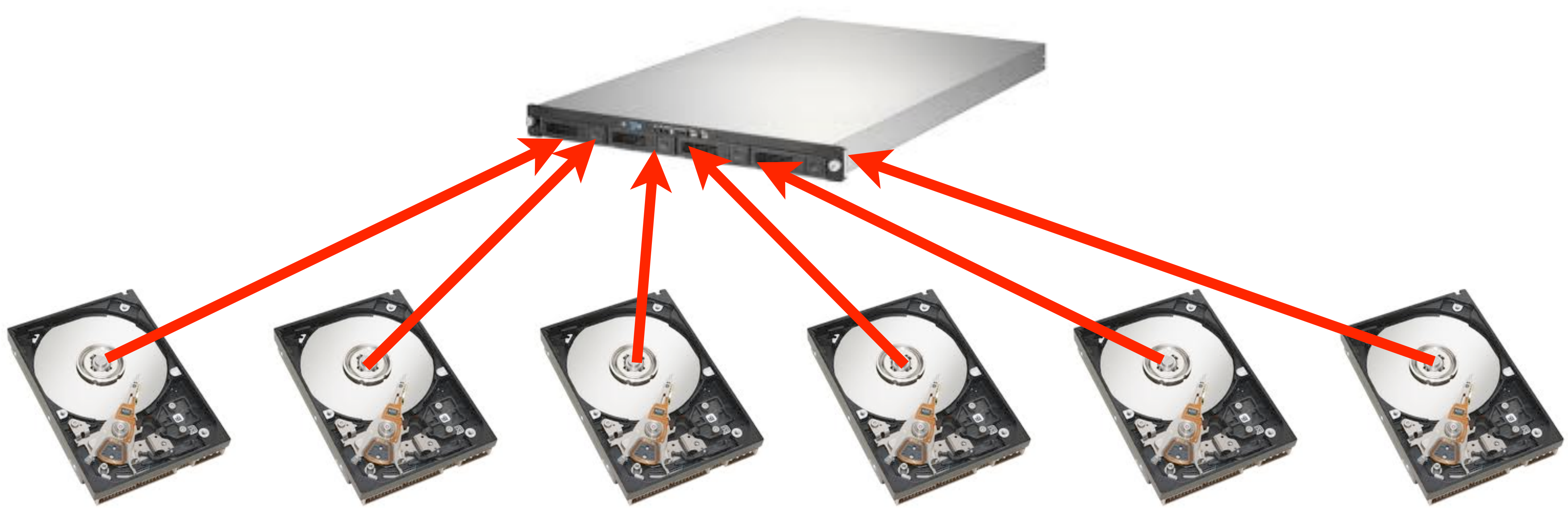
More stuff (take a systems class!)

- Gossip protocols
  Information distribution via random walks
  (see e.g. Kempe, Kleinberg, Gehrke, etc.)
- Time synchronization / quorums
  Byzantine fault tolerance (Lamport / Paxos)
  Google Chubby, Yahoo Zookeeper
- Serialization
  Thrift, JSON, Protocol buffers, Avro
- Interprocess communication
  MPI (do not use), OpenMP, ICE

1.4 Storage

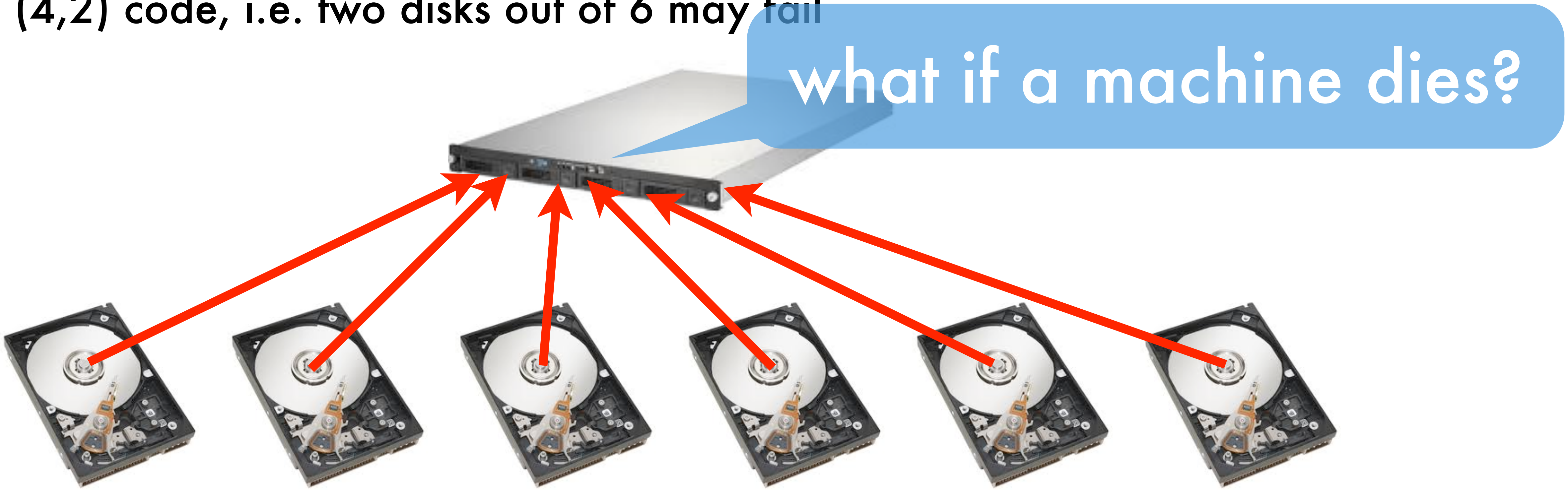
RAID

- Redundant array of inexpensive disks
  - Aggregate storage of many disks
  - Aggregate bandwidth of many disks
  - Fault tolerance (optional)
- RAID 0 - stripe data over disks (good bandwidth, faulty)
- RAID 1 - mirror disks (mediocre bandwidth, fault tolerance)
- RAID 5 - stripe data with 1 disk for parity (good bandwidth, fault tolerance)
- Even better - use error correcting code for fault tolerance, e.g. (4,2) code, i.e. two disks out of 6 may fail



RAID

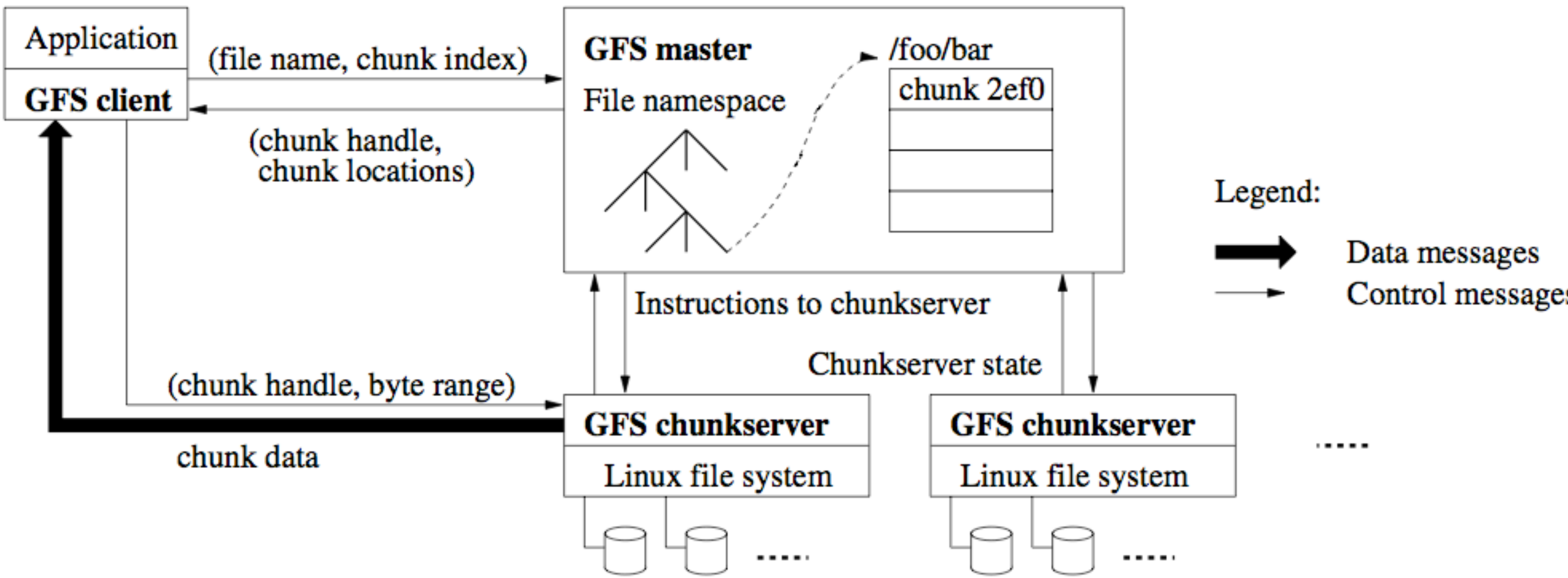
- Redundant array of inexpensive disks
  - Aggregate storage of many disks
  - Aggregate bandwidth of many disks
  - Fault tolerance (optional)
- RAID 0 - stripe data over disks (good bandwidth, faulty)
- RAID 1 - mirror disks (mediocre bandwidth, fault tolerance)
- RAID 5 - stripe data with 1 disk for parity (good bandwidth, fault tolerance)
- Even better - use error correcting code for fault tolerance, e.g. (4,2) code, i.e. two disks out of 6 may fail



Distributed replicated file systems

- Internet workload
  - Bulk sequential writes
  - Bulk sequential reads
  - No random writes (possibly random reads)
  - High bandwidth requirements per file
  - High availability / replication
- Non starters
  - Lustre (high bandwidth, but no replication outside racks)
  - Gluster (POSIX, more classical mirroring, see Lustre)
  - NFS/AFS/whatever - doesn’t actually parallelize

Google File System / HDFS

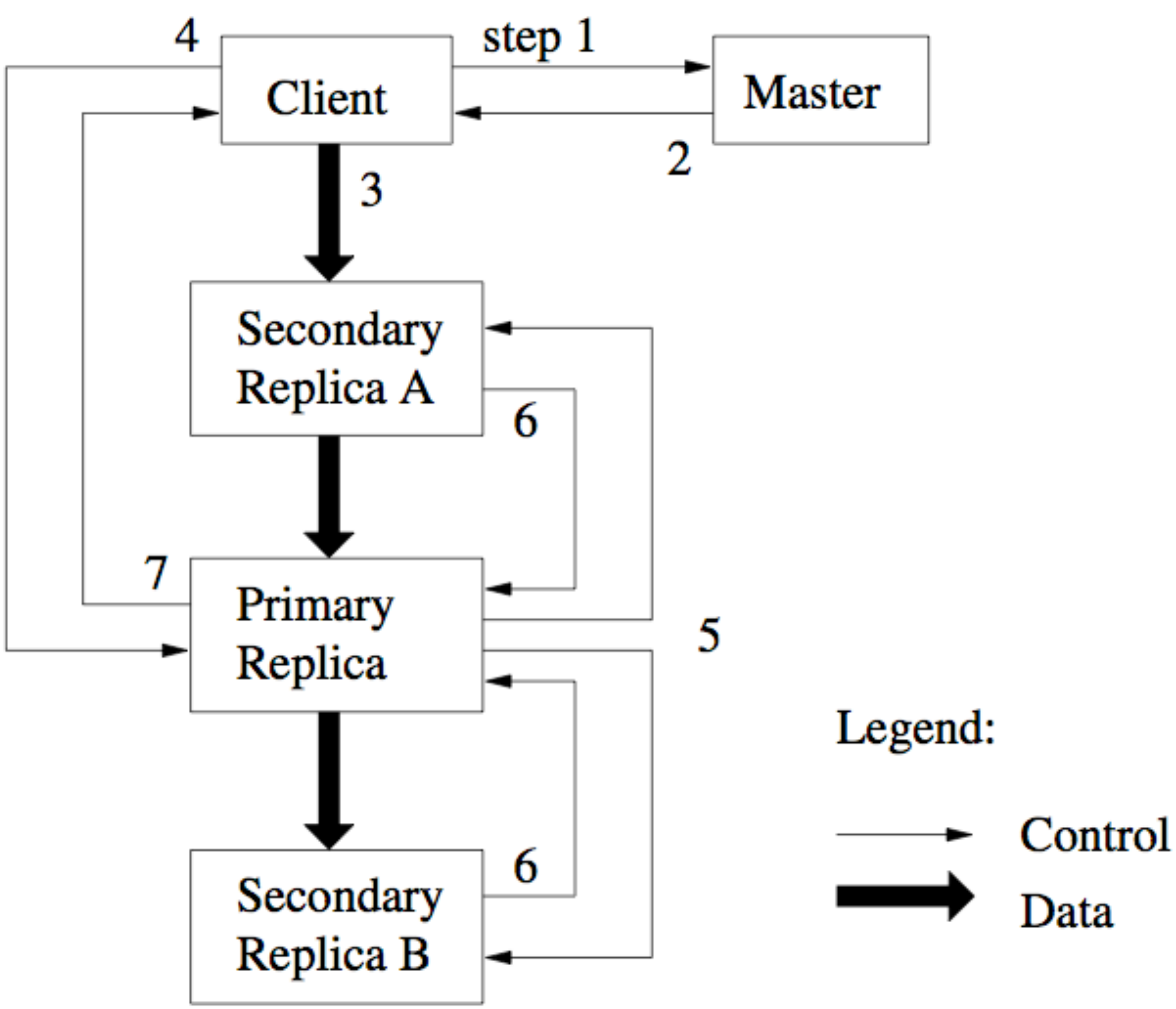


Ghemawat, Gobioff, Leung, 2003

- Chunk servers hold blocks of the file (64MB per chunk)
- Replicate chunks (chunk servers do this autonomously). More bandwidth and fault tolerance
- Master distributes, checks faults, rebalances (Achilles heel)
- Client can do bulk read / write / random reads

Google File System / HDFS

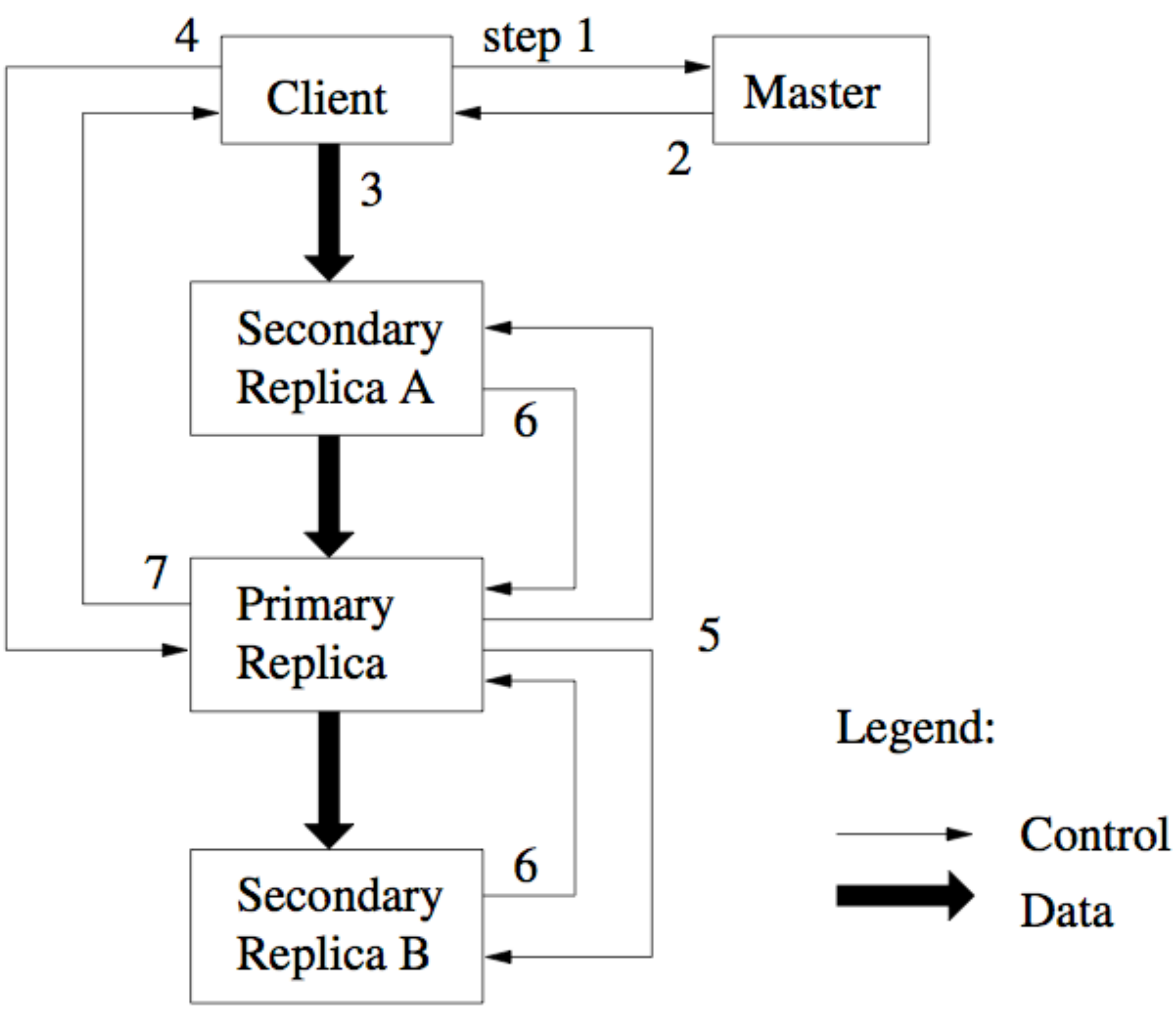
1. Client requests chunk from master
2. Master responds with replica location
3. Client writes to replica A
4. Client notifies primary replica
5. Primary replica requests data from replica A
6. Replica A sends data to Primary replica (same process for replica B)
7. Primary replica confirms write to client



Google File System / HDFS

1. Client requests chunk from master
2. Master responds with replica location
3. Client writes to replica A
4. Client notifies primary replica
5. Primary replica requests data from replica A
6. Replica A sends data to Primary replica (same process for replica B)
7. Primary replica confirms write to client

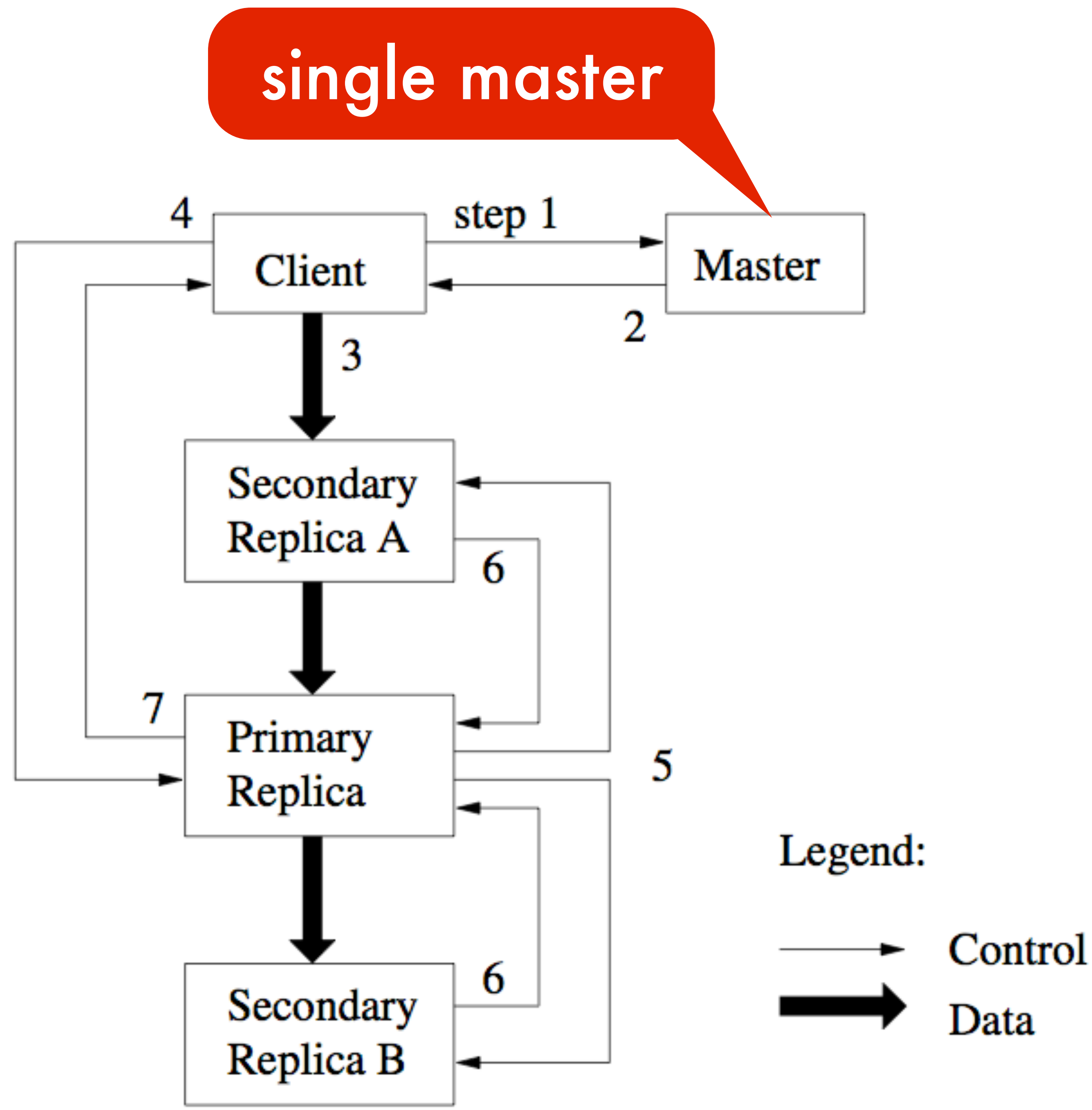
- Master ensures nodes are live
- Chunks are checksummed
- Can control replication factor for hotspots / load balancing
- Deserialize master state by loading data structure as flat file from disk (fast)



Google File System / HDFS

1. Client requests chunk from master
2. Master responds with replica location
3. Client writes to replica A
4. Client notifies primary replica
5. Primary replica requests data from replica A
6. Replica A sends data to Primary replica (same process for replica B)
7. Primary replica confirms write to client

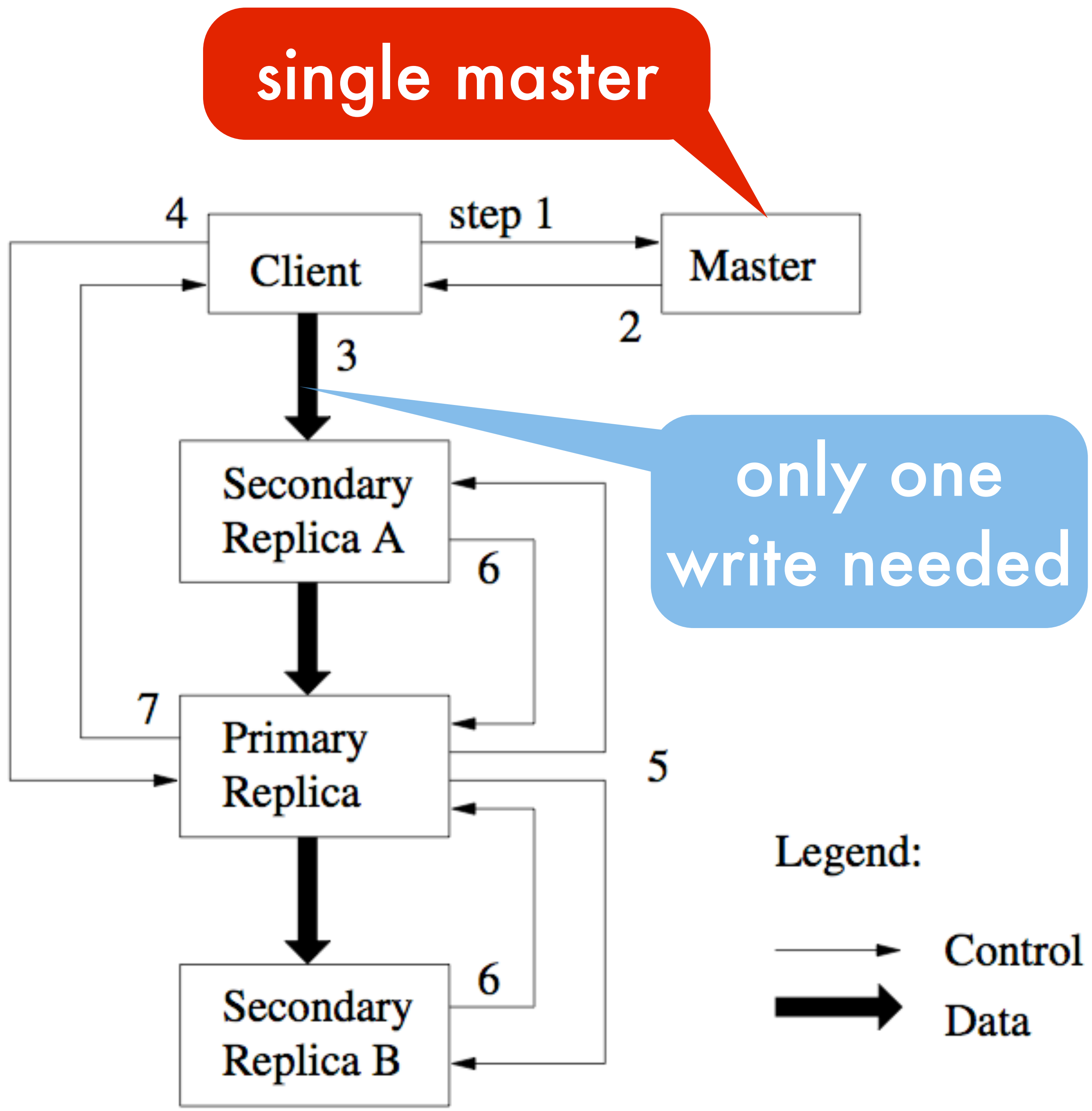
- Master ensures nodes are live
- Chunks are checksummed
- Can control replication factor for hotspots / load balancing
- Deserialize master state by loading data structure as flat file from disk (fast)



Google File System / HDFS

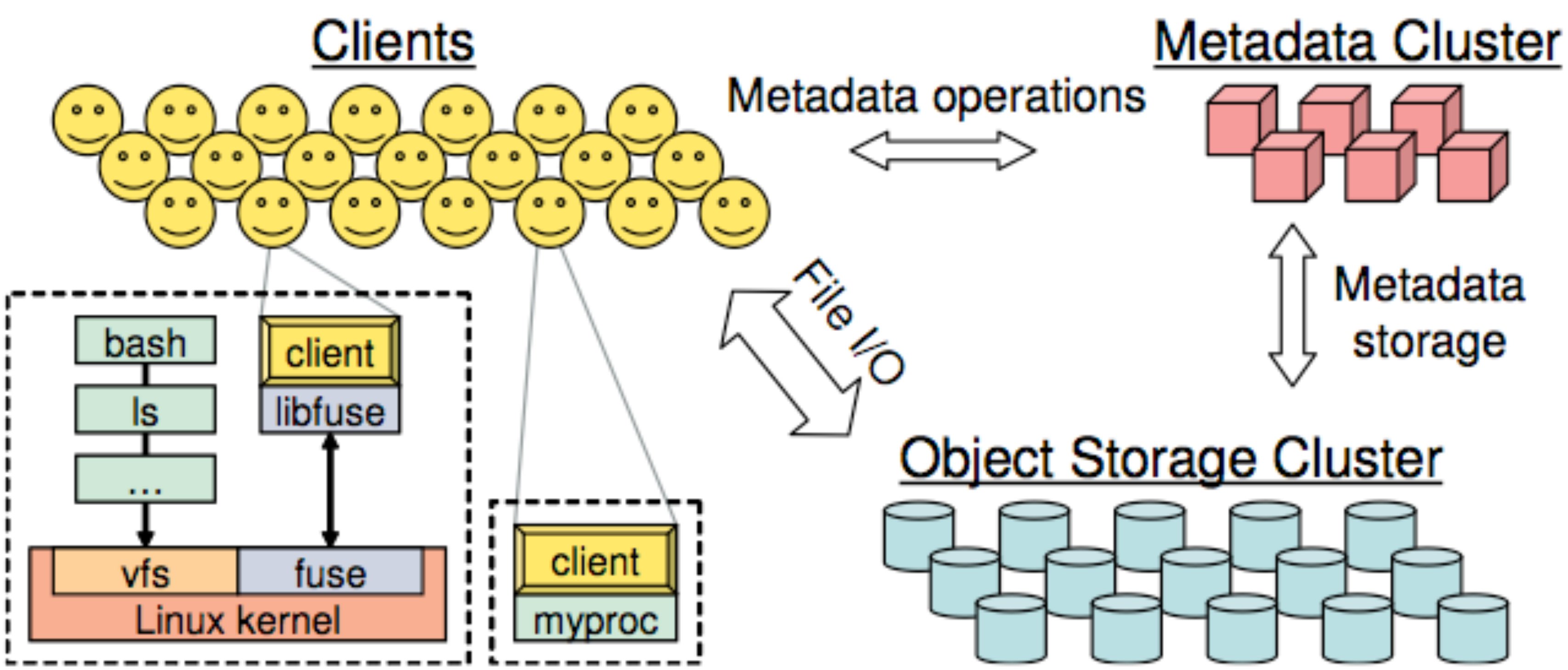
1. Client requests chunk from master
2. Master responds with replica location
3. Client writes to replica A
4. Client notifies primary replica
5. Primary replica requests data from replica A
6. Replica A sends data to Primary replica (same process for replica B)
7. Primary replica confirms write to client

- Master ensures nodes are live
- Chunks are checksummed
- Can control replication factor for hotspots / load balancing
- Deserialize master state by loading data structure as flat file from disk (fast)



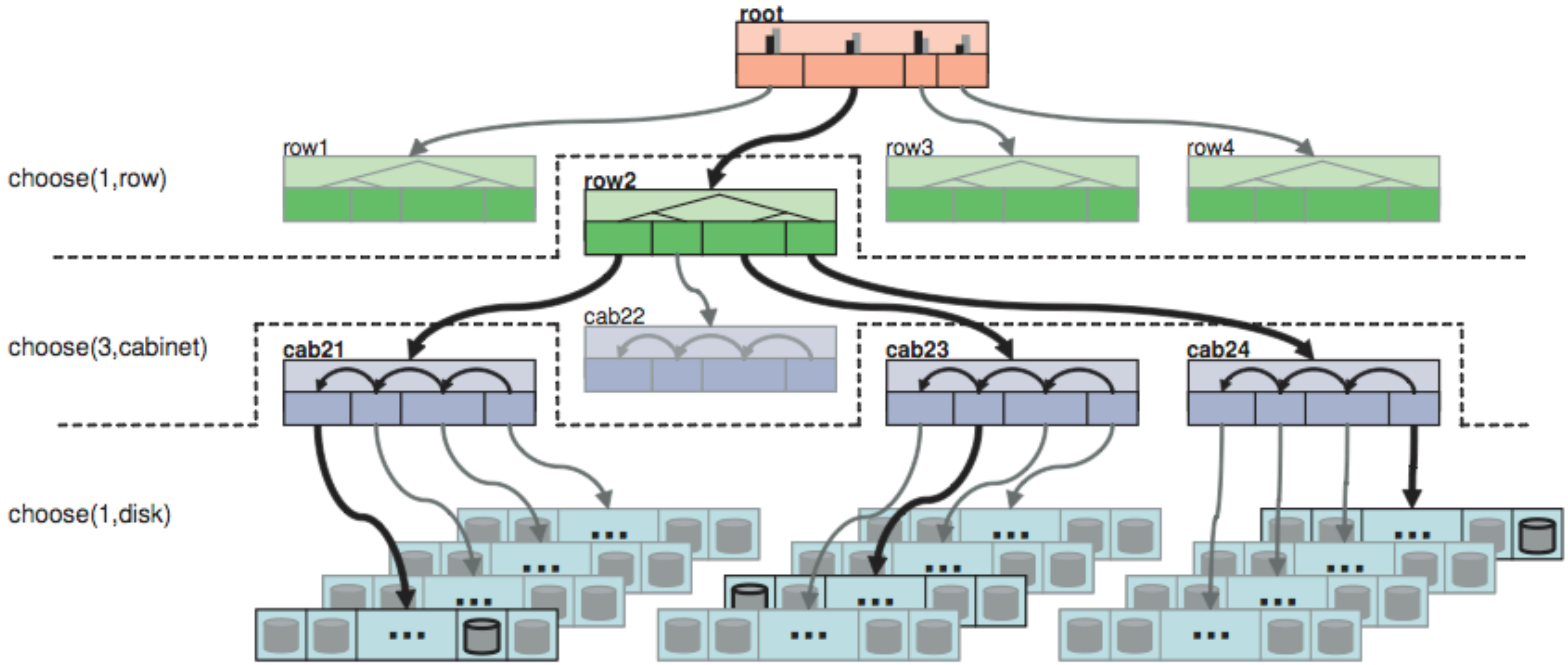
CEPH/CRUSH

- No single master
- Chunk servers deal with replication / balancing on their own
- Chunk distribution using proportional consistent hashing
- Layout plan for data - effectively a sampler with given marginals
  Research question - can we adjust the probabilities based on statistics?



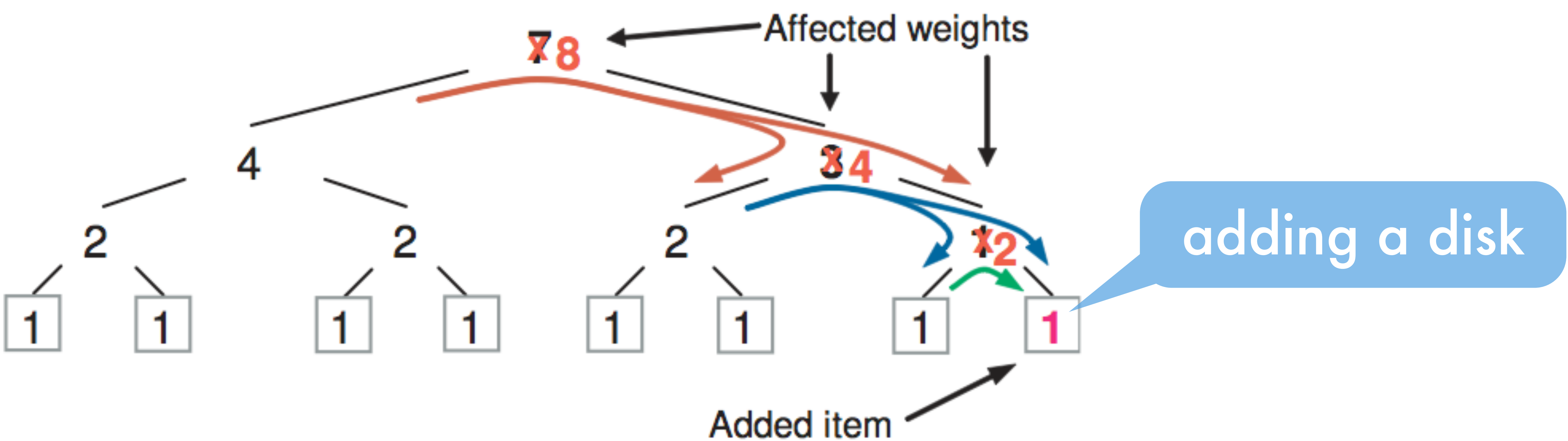
http://ceph.newdream.org (Weil et al., 2006)

CEPH/CRUSH



- Various sampling schemes (ensure that no unneccessary data is moved)
- In the simplest case proportional consistent hashing from pool of objects (pick k disks out of n for block with given ID)
- Can incorporate replication/bandwidth scaling like RAID (stripe block over several disks, error correction)

CEPH/CRUSH

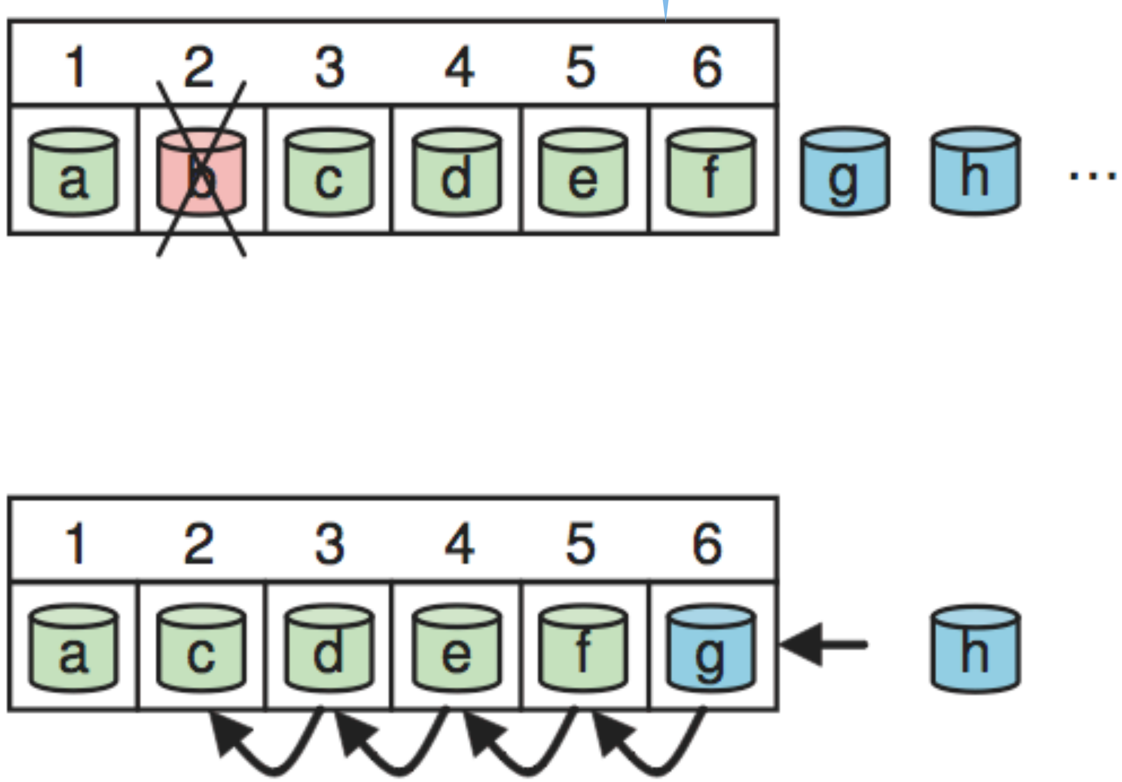


- Various sampling schemes (ensure that no unneccessary data is moved)
- In the simplest case proportional consistent hashing from pool of objects (pick k disks out of n for block with given ID)
- Can incorporate replication/bandwidth scaling like RAID (stripe block over several disks, error correction)

CEPH/CRUSH fault recovery

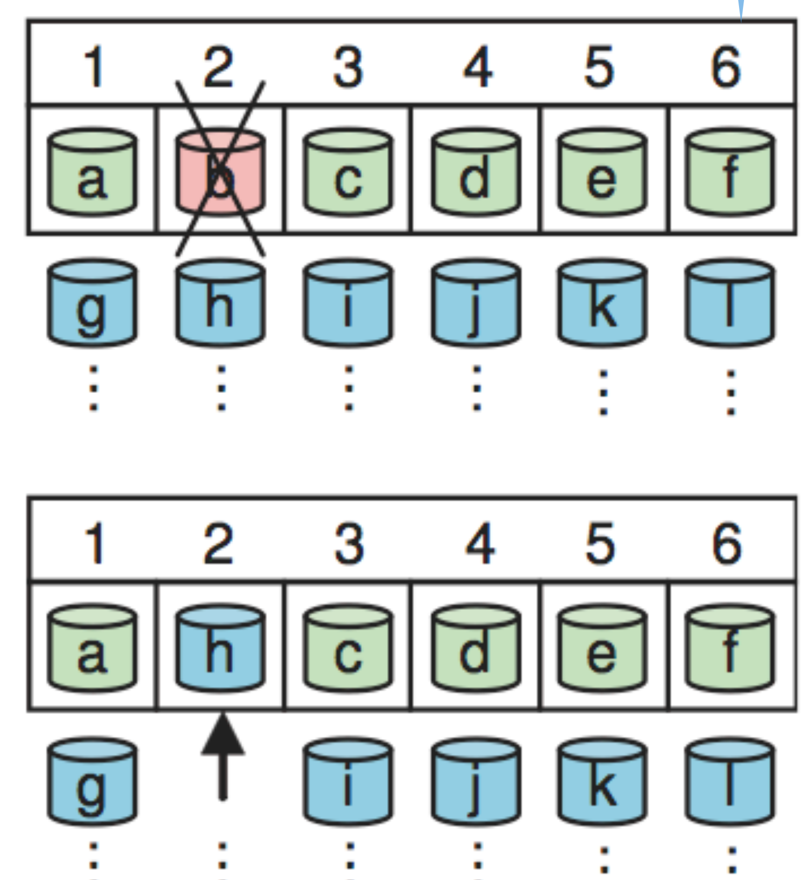
plain replication

$r' = r + f$



striped data

$r' = r + f_r n$

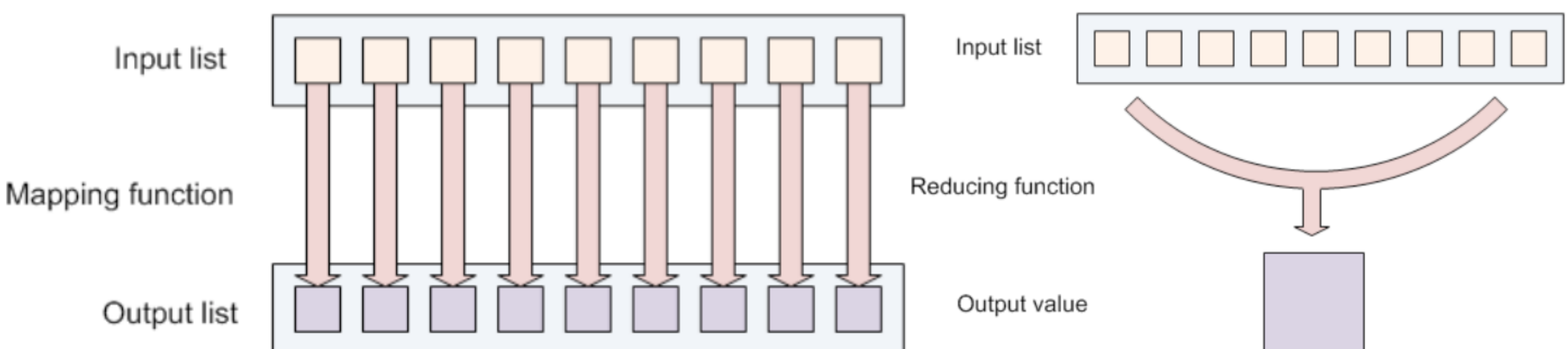


- Hadoop patch available - use instead of HDFS

1.5 Processing

Map Reduce

- 1000s of (faulty) machines
- Lots of jobs are mostly embarrassingly parallel (except for a sorting/transpose phase)
- Functional programming origins
  - Map(key,value)
    processes each (key,value) pair and outputs a new (key,value) pair
  - Reduce(key,value)
    reduces all instances with same key to aggregate



from Ramakrishnan, Sakrejda, Canon, DoE 2011

Map Reduce

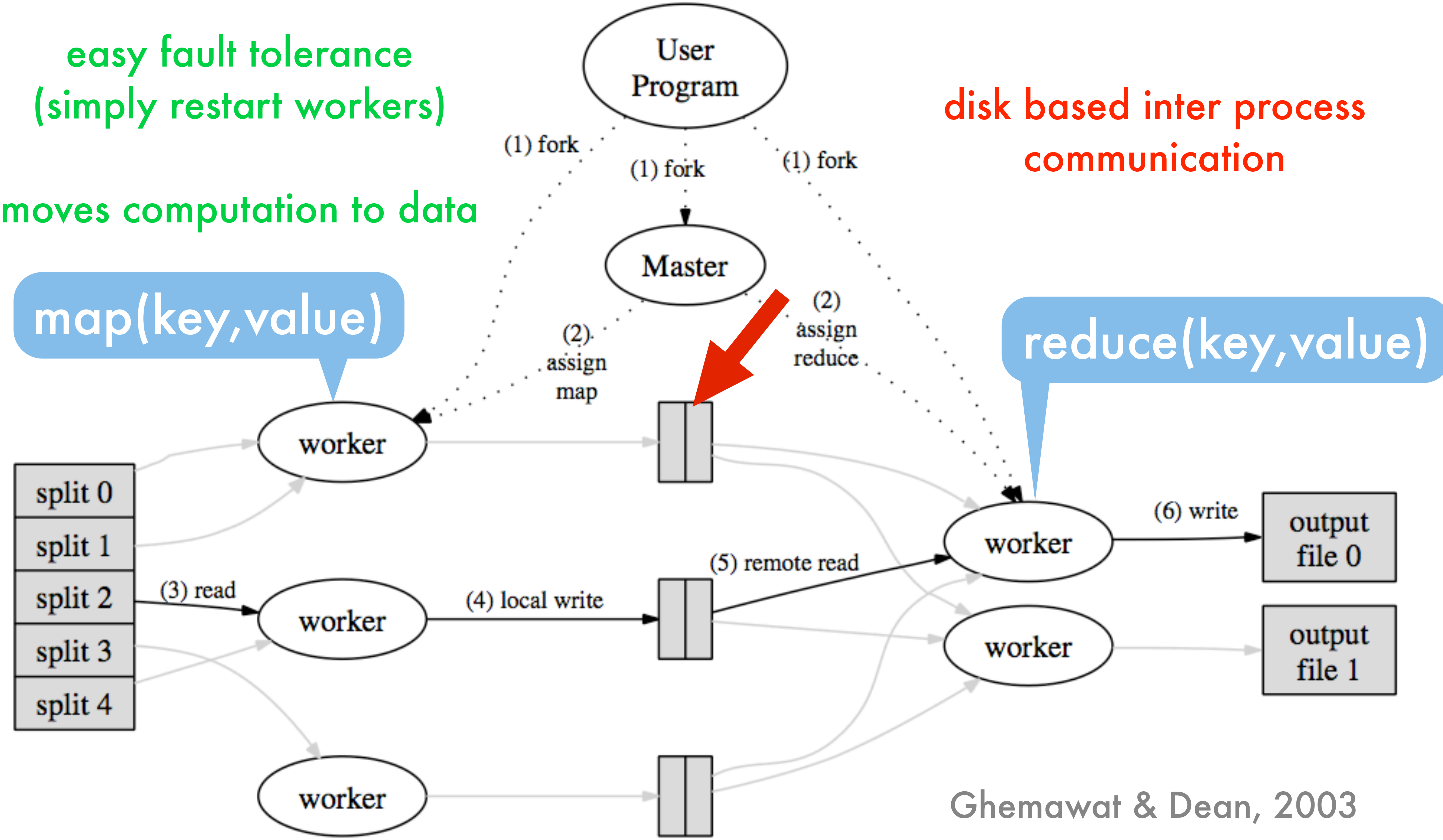
- 1000s of (faulty) machines
- Lots of jobs are mostly embarrassingly parallel
(except for a sorting/transpose phase)
- Functional programming origins
    - Map(key,value)
    processes each (key,value) pair and outputs a new (key,value) pair
    - Reduce(key,value)
    reduces all instances with same key to aggregate
- Example - extremely naive wordcount
    - Map(docID, document)
    for each document emit many (wordID, count) pairs
    - Reduce(wordID, count)
    sum over all counts for given wordID and emit (wordID, aggregate)

Map Reduce

easy fault tolerance
(simply restart workers)

moves computation to data

disk based inter process communication

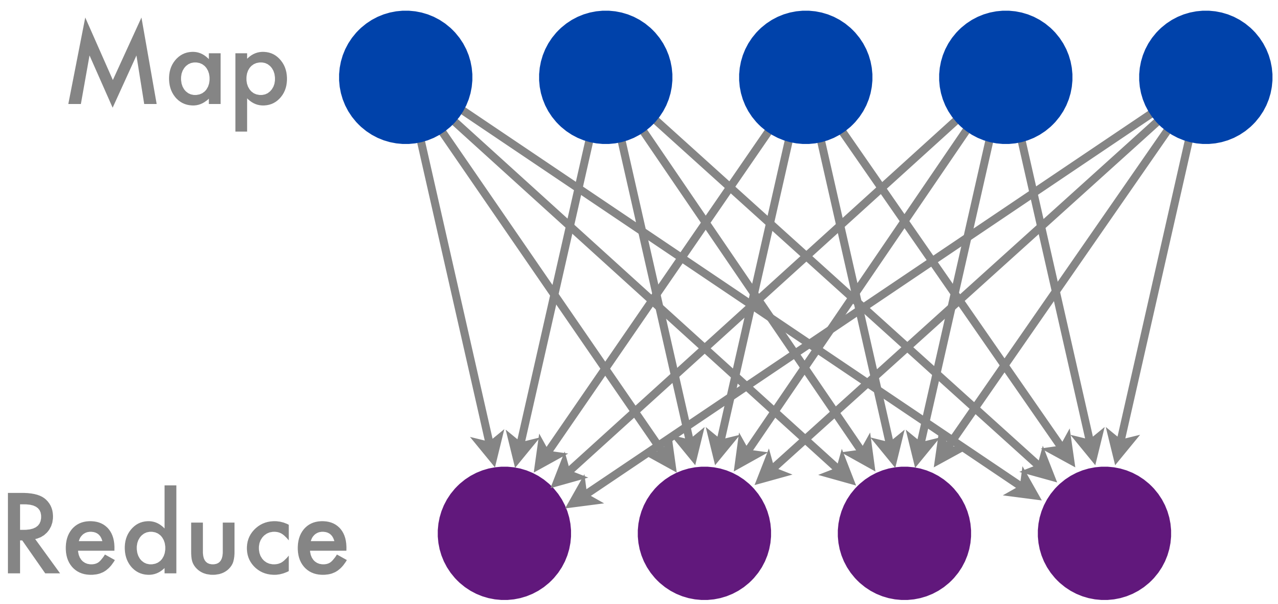


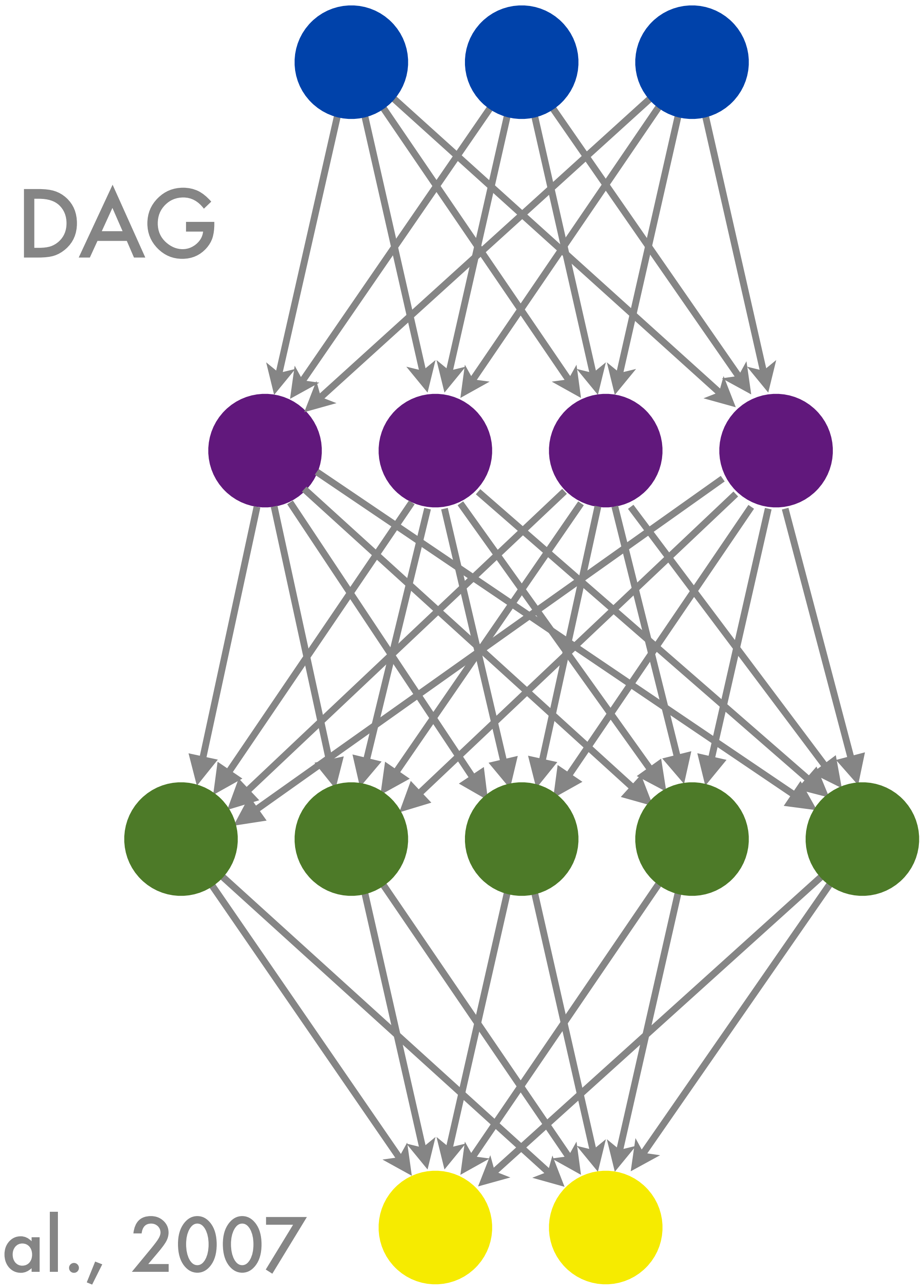
Ghemawat & Dean, 2003

Map Combine Reduce

- Combine aggregates keys before sending to the reducer (saves bandwidth)
- Map must be stateless in blocks
- Reduce must be commutative in data
- Fault tolerance
  - Start jobs where the data is
    (move code note data - nodes run the file system, too)
  - Restart machines if maps fail (have replicas)
  - Restart reducers based on intermediate data
- Good fit for many algorithms
- Good if only a small number of MapReduce iterations needed
- Need to request machines at each iteration (time consuming)
- State lost in between maps
- Communication only via file I/O

Dryad

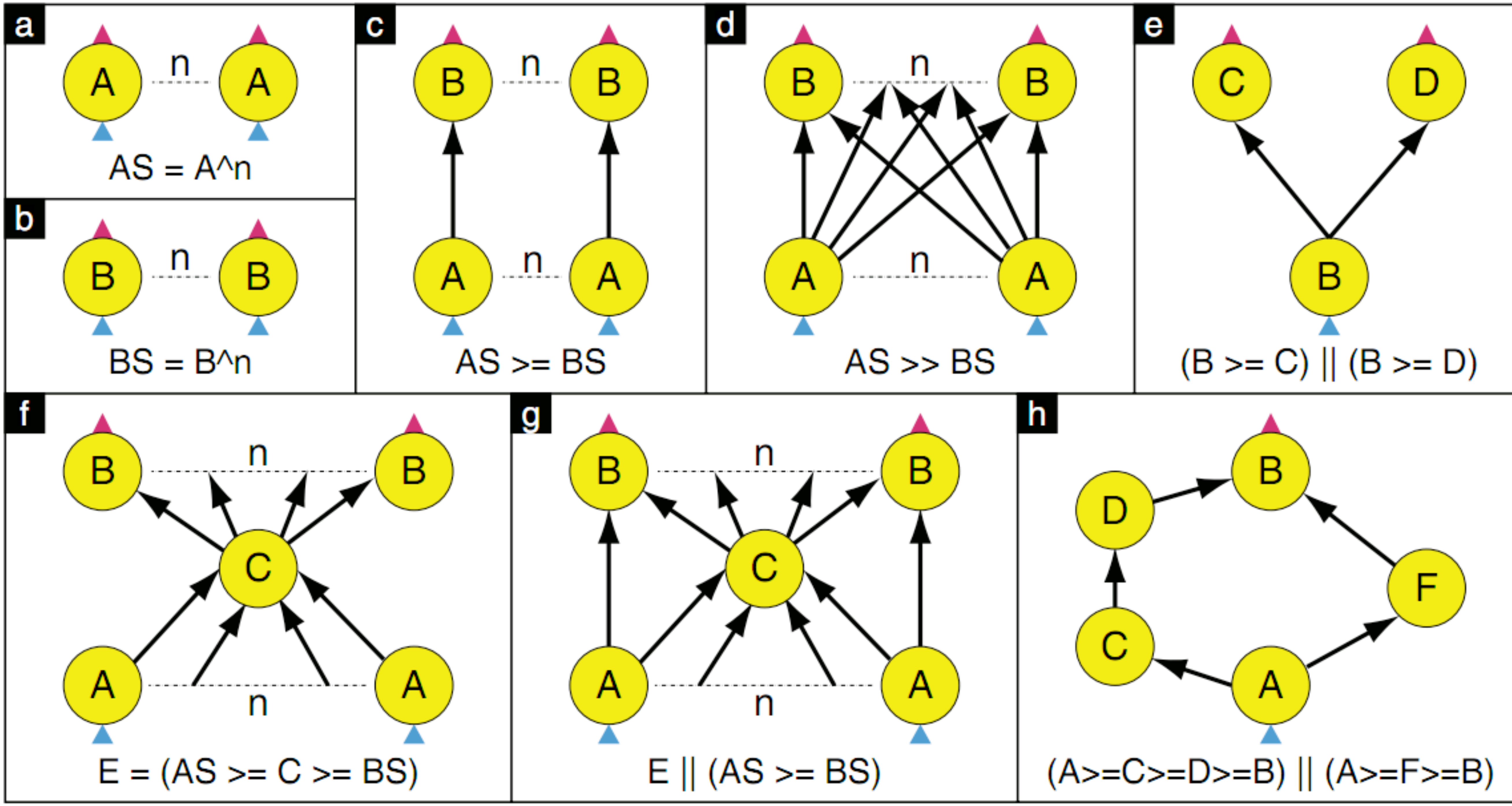




- Directed acyclic graph
- System optimizes parallelism
- Different types of IPC (memory FIFO/network/file)
- Tight integration with .NET (allows easy prototyping)

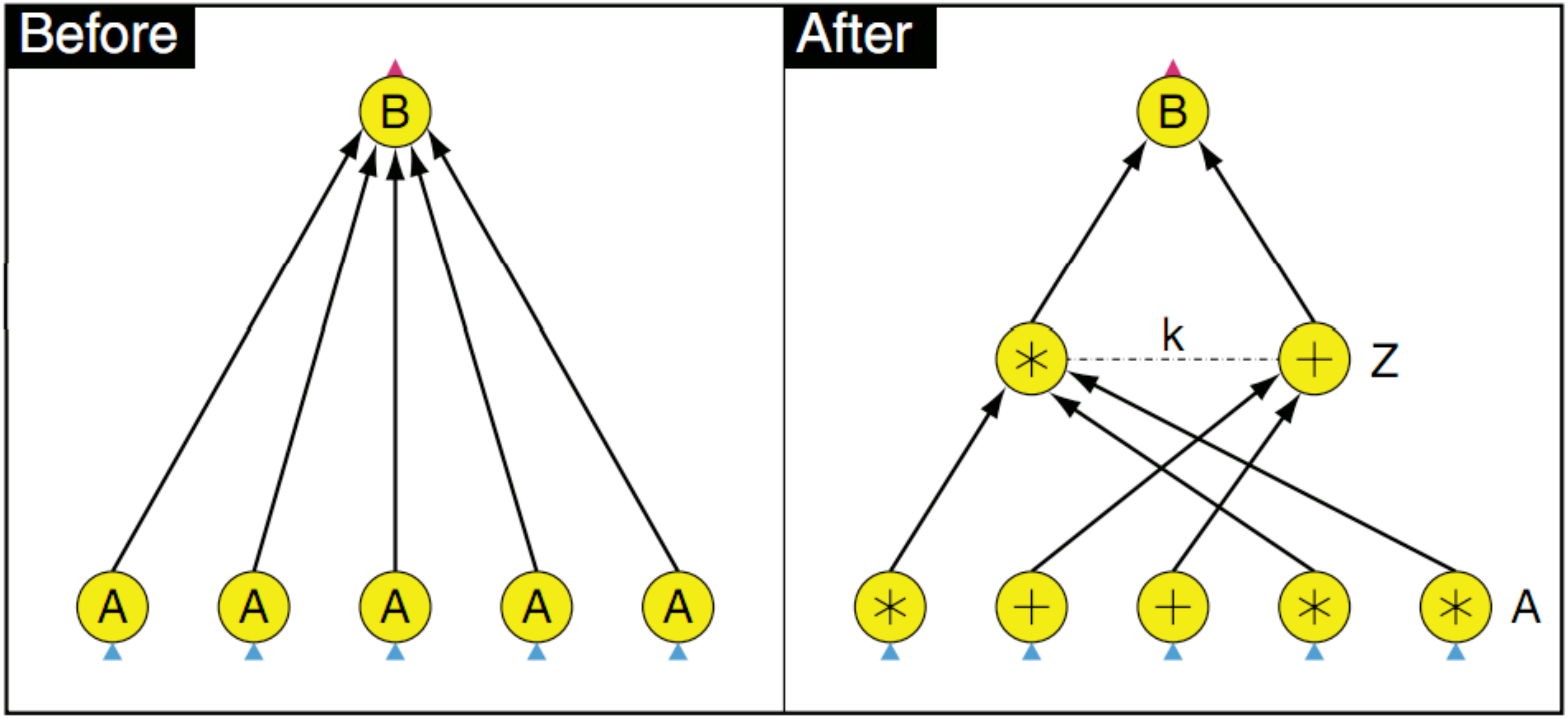
Isard et al., 2007

DRYAD



graph description language

DRYAD



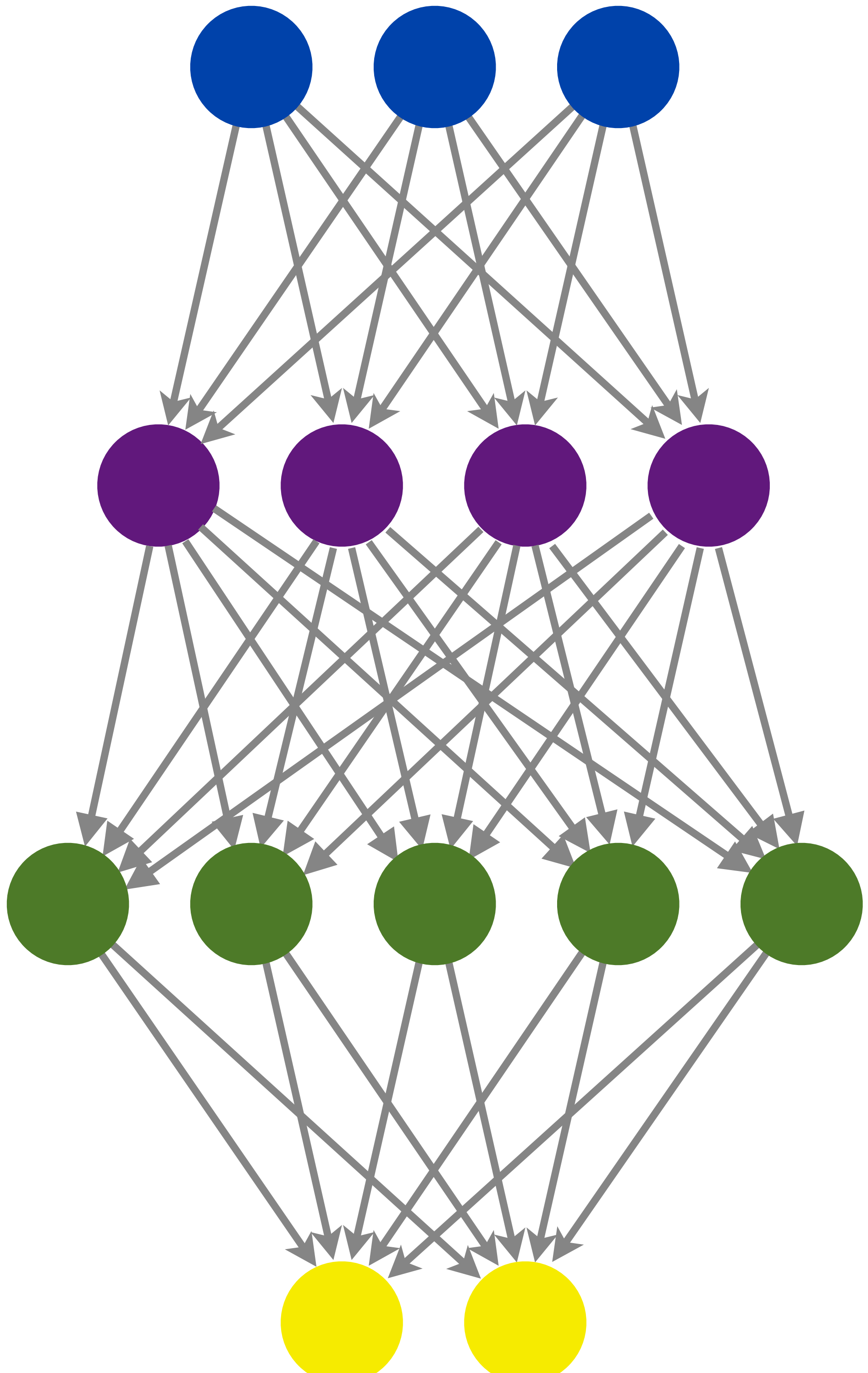
automatic graph refinement

S4

- Directed acyclic graph (want Dryad-like features)
- Real-time processing of data (as stream)
- Scalability (decentralized & symmetric)
- Fault tolerance
- Consistency for keys

- Processing elements
  - Ingest (key, value) pair
  - Capabilities tied to ID
  - Clonable (for scaling)

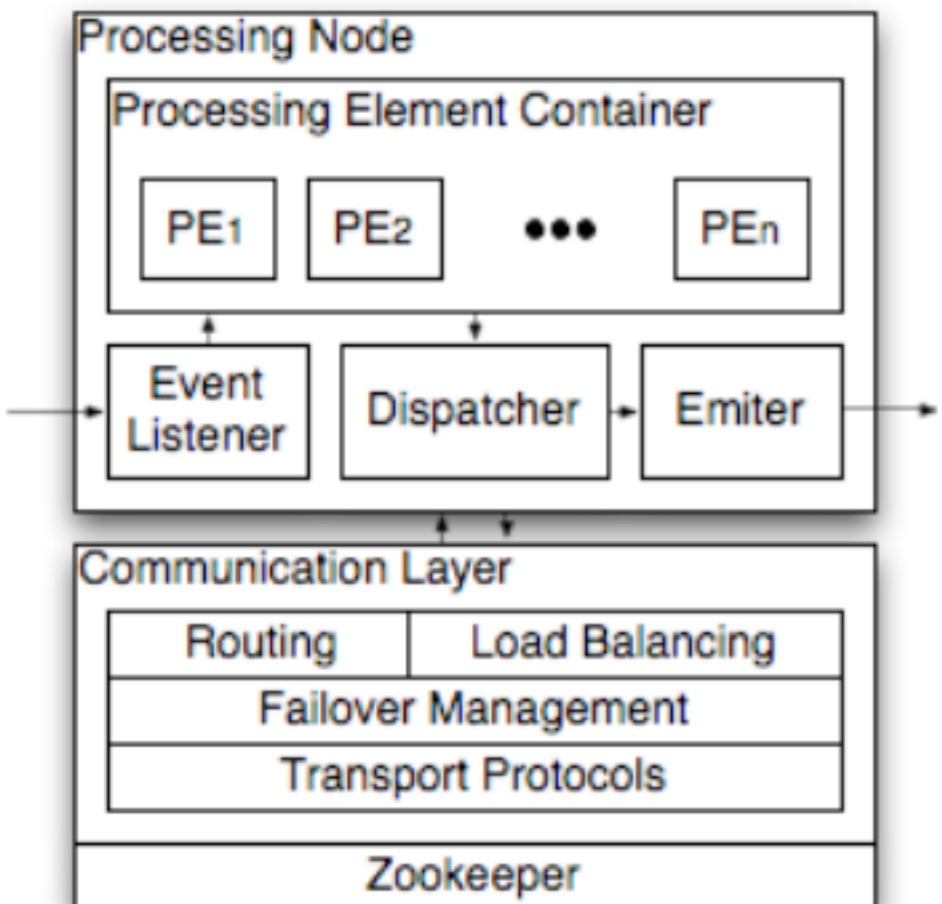
- Simple implementation e.g. via consistent hashing

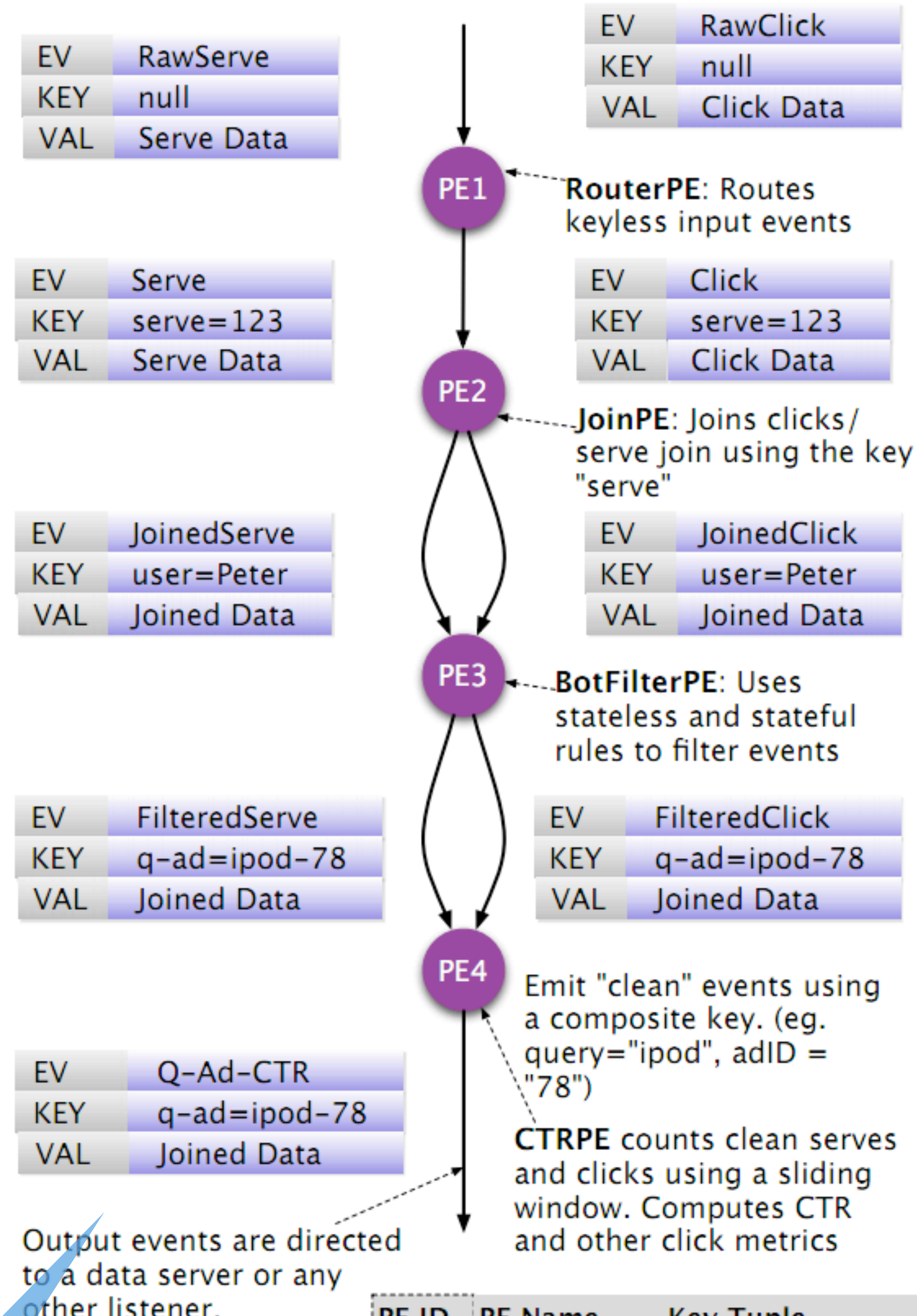


http://s4.io Neumeyer et al, 2010

S4

processing element





PE ID	PE Name	Key Tuple
PE1	RouterPE	null
PE2	JoinPE	serve=123
PE3	BotFilterPE	user="Peter"
PE4	CTRPE	q-ad=ipod-78

click through rate estimation

Alternative

build your own
e.g. based on IPC framework

only do this if you REALLY know what you’re doing

1.6 Data(bases/storage)

Distributed Data Stores

- SQL
  - rich query syntax (it's a programming language)
  - expensive to scale (consistency, fault tolerance)
- (key, value) storage
  - simple protocol: put(key, value), get(key)
  - lightweight scaling
- Row database (BigTable, HBase)
  - create/change/delete rows, create/delete column families
  - timestamped data (can keep several versions)
  - scalable on GoogleFS
- Intermediate variants
  - replication between COLOs
  - variable consistency guarantees

(key,value) storage

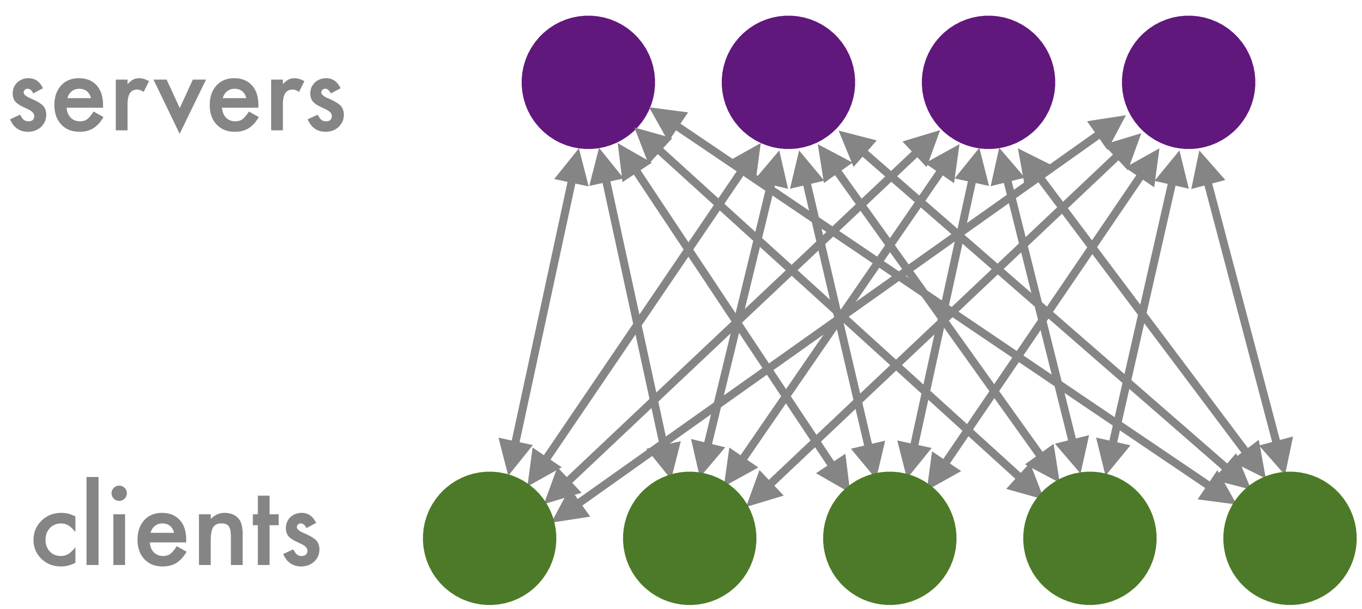
- Protocol
  - put(key, value, version)
  - (value, version) = get(key)
- Attributes
  - persistence (recover data if machine fails)
  - replication (distribute copies / parts over many machines)
  - high availability (network partition tolerant, always writable)
  - transactions (confirmed operations)
  - rack locality (exploit communications topology/replication)

Comparison of NoSQL Systems

Project Name	Type	Persistence	Replication	High Availability	Transactions	Rack-locality Awareness	Implemen
AllegroGraph	Graph database	Yes	No - v5, 2010	Yes	Yes	No	Common Lisp
Apache Jackrabbit	Key-value & Hierarchical & Document	Yes	Yes	Yes	Yes	likely	Java
Apstrata	Document & Key-Value	Yes	Yes	Yes	Yes	No	Java
Berkeley DB/Dbm/Ndbm (bdb)1.x	Key-value	Yes	No	No	No	No	C
Berkeley DB Sleepycat/Oracle Berkeley DB 2.x	Key-value	Yes	Yes	Unknown	Yes	No	C, C++, or Jav
Cassandra	Key-value	Yes	Yes	Distributed	Eventually consistent	Yes	Java
Chordless	Key-Value with RPC	Yes	Yes	Yes	Yes	No	Java
Citrusleaf	Document & Key-value	Yes	Yes	Distributed	Yes	No	C
CouchDB	Document	Yes	Yes	replication + load balancing	Atomicity is per document, per CouchDB instance	No	Erlang
GT.M	Key-value	Yes	Yes	Yes	Yes	Depends on user configuration	C (small bits o
Project Name	Type	Persistence	Replication	High Availability	Transactions	Rack-locality Awareness	Implemen
HBase	Key-value	Yes. Major version upgrades require re-import.	See HDFS, S3 or EBS.	maybe with Zookeeper in 0.21?	Unknown	See HDFS, S3 or EBS.	Java
HyperGraphDB	Graph Database	Yes	Yes	Unknown	Yes	Unknown	Java/C++
Hypertable	Key-value	Yes	Yes, with KosmosFS and Ceph	coming in 2.0	coming	Yes, with KosmosFS	C++
InfoGrid	Graph database with web frontend	Yes (pluggable: native, SQL)	Yes	Unknown	Yes	Unknown	Java
Information Management System IBM IMS aka DB1	Key-value. Multi-level	Yes	Yes	Yes, with HALDB	Yes, with IMS TM	Unknown	Assembler
Keyspace (Scalien)	Key-value	Yes	Yes (Paxos)	fail-over	Unknown	Yes	C/C++
MarkLogic	XML	Yes	Yes (automatic)	Yes	Yes	Configurable	C++
M/DB	Document & Key-value	Yes	Yes (via GT.M)	Yes (via GT.M)	Yes (via GT.M)	No	M
Membase	Key-value	Yes	Yes	Yes	Yes	No	C++, C, Pytho
Memcache	Key-value	No	No	No	Yes	No	C
MongoDB	Document (JSON)	Yes	Yes	fail-over	Single document atomicity	No	C++
Project Name	Type	Persistence	Replication	High Availability	Transactions	Rack-locality Awareness	Implemen
Neo4j	Graph database	Yes	Yes	Yes, Read slaves	Yes	No	Java
OrientDB	Document database	Yes	In development	In development	Yes	No	Java
Project Voldemort	Key-value	Yes	Yes	Distributed	Unknown	No	Java
Redis	Key-value	Yes. But last few queries can be lost.	Yes	No	Yes	No	Ansi-C
Riak (Basho)	Document & Key-value	Yes	Yes	Yes (including write-availability)	Eventually Consistent	Unknown	Erlang, C
Sausalito	XML Data Model (XDM)	Yes (AWS S3)	Yes (see S3)	Unknown	Unknown	see S3	XQuery
Sherpa (aka PNUTS) (Yahoo!)	Document & Key-value	Yes	Yes	Yes	Yes	Yes	C++ ?
SimpleDB (Amazon.com)	Document & Key-value	Yes	Yes (automatic)	Yes	Unknown	likely	Erlang
sones GraphDB	OO & Graph database	Yes	Yes	Yes	Yes	Unknown	C# .Net
TigerLogic	XQuery Data Model				Yes		XQuery
Tokyo Cabinet	Key-value	Yes	No	No	Yes	No	C
VertexDB	Graph database	Yes	No		Yes		C
Project Name	Type	Persistence	Replication	High Availability	Transactions	Rack-locality Awareness	Implemen

memcached

- Protocol (no versioning)
  - put(key, value)
  - value = get(key) (returns error if key non-existent)



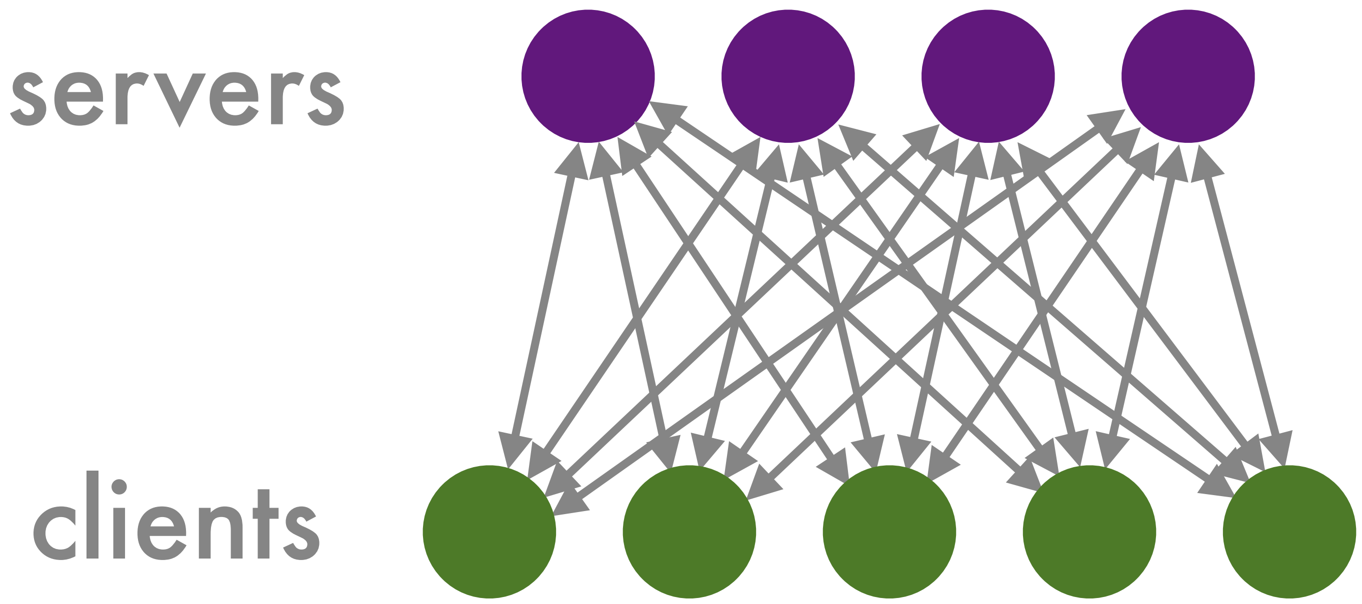
- Load distribution by consistent hashing

$$m(\text{key}) = \underset{m \in \mathcal{M}}{\operatorname{argmin}} \, h(m, \text{key})$$

  - cache dynamic content
  - disposable distributed storage (e.g. for gradient aggregation)

memcached

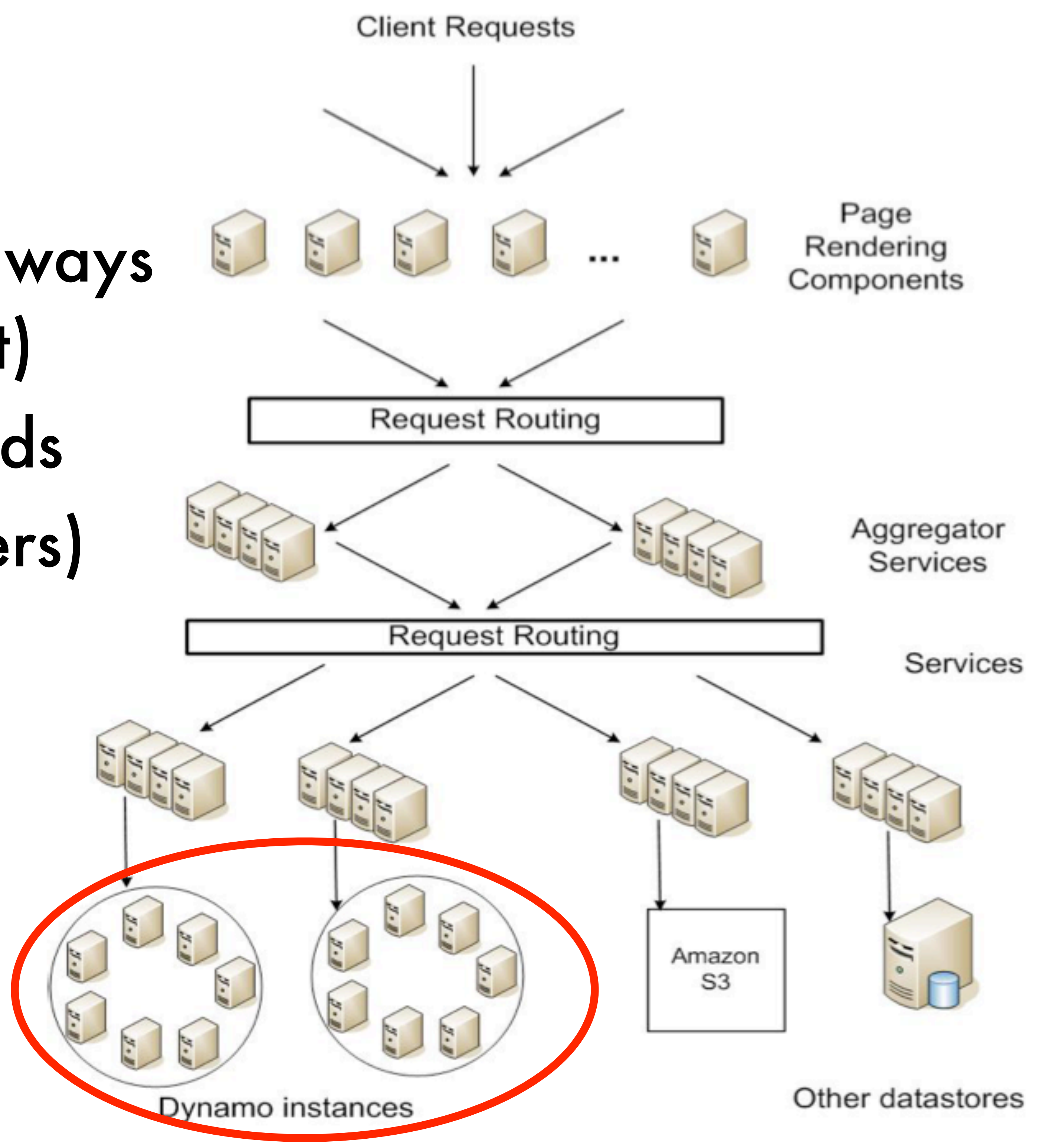
- Protocol (no versioning)
  - put(key, value)
  - value = get(key) (returns error if key not existent)



- Example: distributed subgradients (much faster than MapReduce)
  - Clients writes put([clientID,blockID], gradient) for all blockIDs
  - Client reads get([clientID,blockID]) for all clientID & aggregates
  - Update parameters based on aggregate gradient & broadcast

Amazon Dynamo

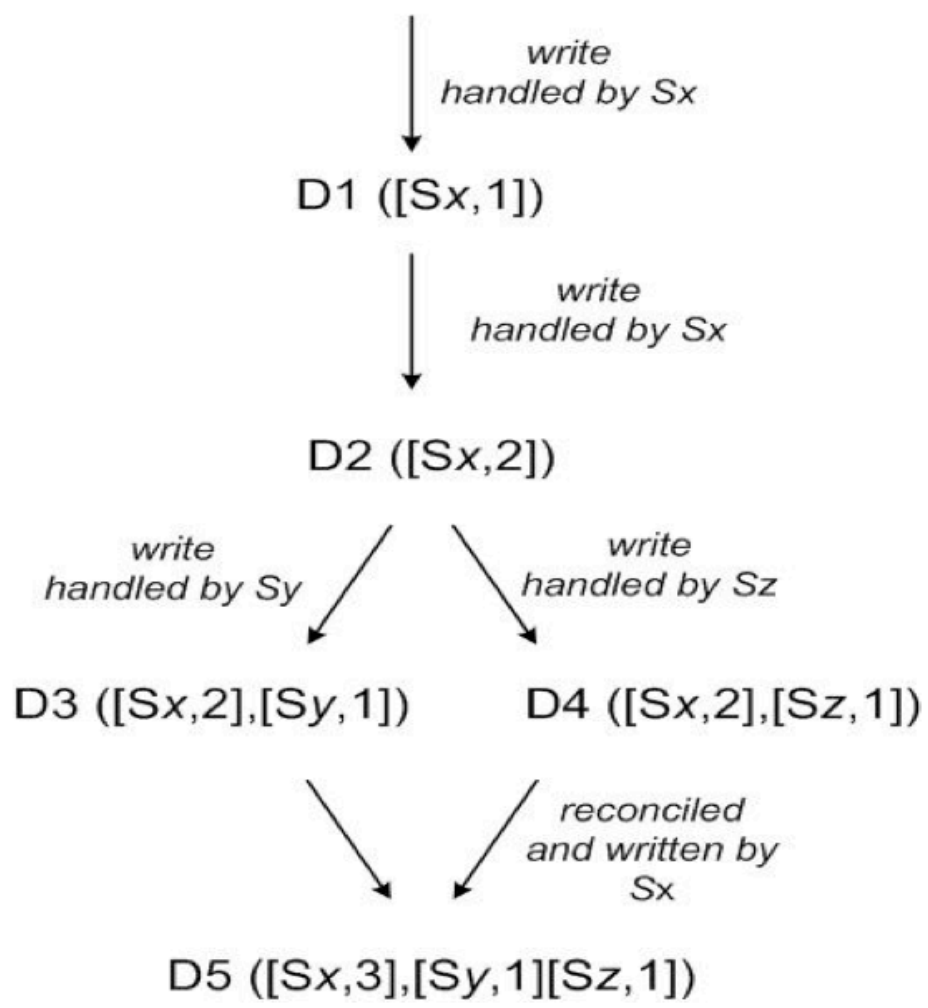
- (key, value) storage
- scalable
- high availability (we can always add to the shopping basket)
- reconcile inconsistent records
- persistent (do not lose orders)



Cassandra is more or less open source version with columns added (and ugly load balancing)

DeCandia et al., 2007

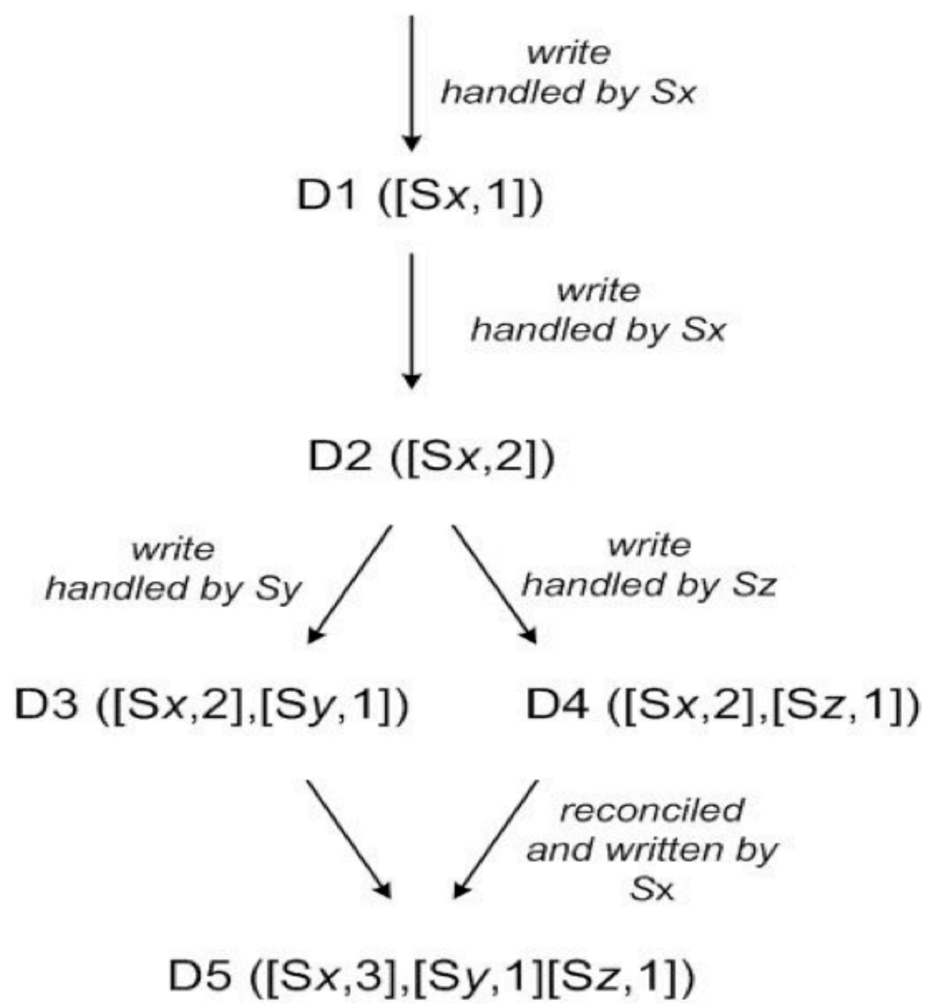
Amazon Dynamo



vector clocks to handle versions

Problem	Technique	Advantage
Partitioning	Consistent Hashing	Incremental Scalability
High Availability for writes	Vector clocks with reconciliation during reads	Version size is decoupled from update rates.
Handling temporary failures	Sloppy Quorum and hinted handoff	Provides high availability and durability guarantee when some of the replicas are not available.
Recovering from permanent failures	Anti-entropy using Merkle trees	Synchronizes divergent replicas in the background.
Membership and failure detection	Gossip-based membership protocol and failure detection.	Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.

Amazon Dynamo



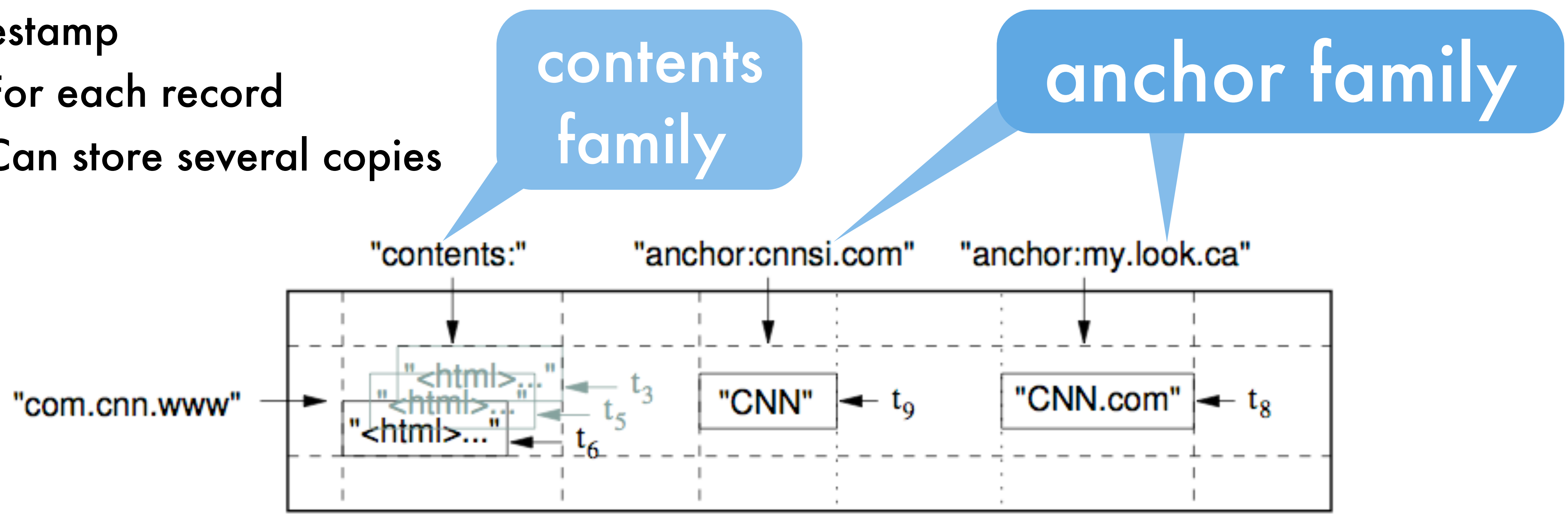
vector clocks to handle versions

Problem	Technique	Advantage
Partitioning	Consistent Hashing	Incremental Scalability
High Availability for writes	Vector clocks with reconciliation during reads	Version size is decoupled from update rates.
Handling temporary failures	Sloppy Quorum and hinted handoff	Provides high availability and durability guarantee when some of the replicas are not available.
Recovering from permanent failures	Anti-entropy using Merkle trees	Synchronizes divergent replicas in the background.
Membership and failure detection	Gossip-based membership protocol and failure detection.	Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.

opportunity for machine learning

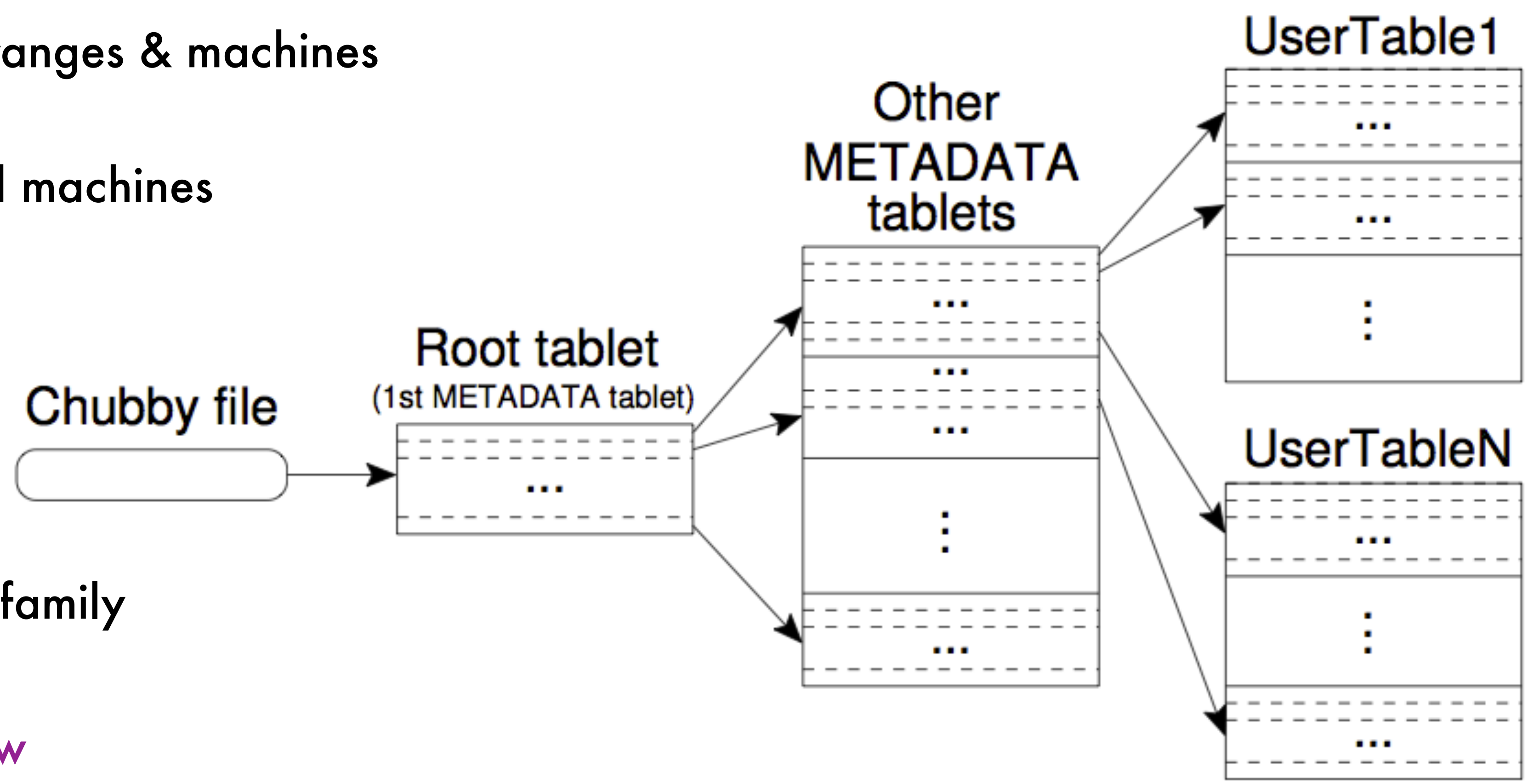
Google Bigtable / HBase

- Row oriented database
  - Partition by row key into tablets
  - Servers hold (preferably) contiguous range of tablets
  - Master assigns tablets to servers
  - Persistence by writing to GoogleFS
- Column families
  - Access control
  - Arbitrary number of columns per family
- Timestamp
  - For each record
  - Can store several copies



Internals

- Chubby / Zookeeper (global consensus server using Paxos)
- Hierarchy
  - Root tablet
    Contains all metadata tablet ranges & machines
  - Metadata tablets
    Contains all tablet ranges and machines
  - User tablets
    Contains the actual data
- Operations
  - Look up row key
  - Row range read
  - Read over columns in column family
  - Time ranged queries
  - Operations are atomic per row
  - Single server per tablet
- Disk/memory trade off
  - Bloom filter to determine which block to read
  - Write diffs only - for lookup traverse from present to past (we will use this for particle filter later)
  - Compaction operator aggregates



NoSQL vs. RDBMS

- RDBMS provides too much
  - ACID transactions
  - Complex query language
  - Lots and lots of knobs to turn
- RDBMS provides too little
  - Lack of (cost-effective) scalability, availability
  - Not enough schema/data type flexibility
- NoSQL
  - Lots of optimization and tuning possible for analytics (Column stores, bitmap indices)
  - Flexible programming model (Group By vs. Map-Reduce; multi-dimensional OLAP)
- But many good ideas to borrow
  - Declarative language
  - parallelization and optimization techniques
  - value of data consistency …

courtesy of Raghu Ramakrishnan

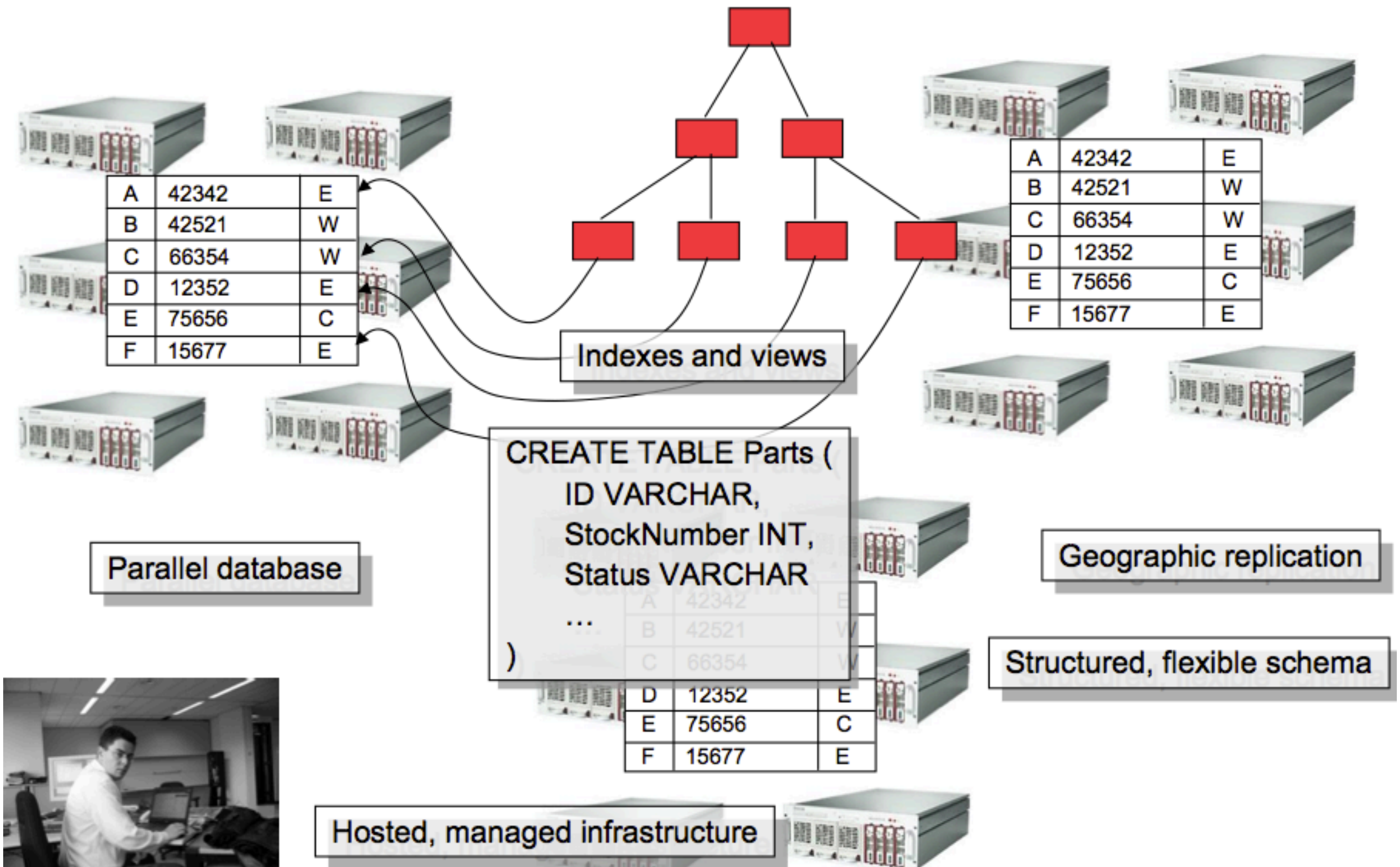
NoSQL vs. RDBMS

- RDBMS provides too much
  - ACID transactions
  - Complex query language
  - Lots and lots of knobs to turn
- RDBMS provides too little
  - Lack of (cost-effective) scalability, availability
  - Not enough schema/data type flexibility
- NoSQL
  - Lots of optimization and tuning possible for analytics (Column stores, bitmap indices)
  - Flexible programming model (Group By vs. Map-Reduce; multi-dimensional OLAP)
- But many good ideas to borrow
  - Declarative language
  - parallelization and optimization techniques
  - value of data consistency …

fix by cluster of RDBMS servers

courtesy of Raghu Ramakrishnan

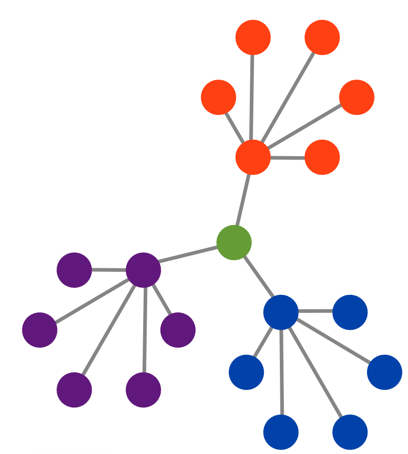
Yahoo high availability storage

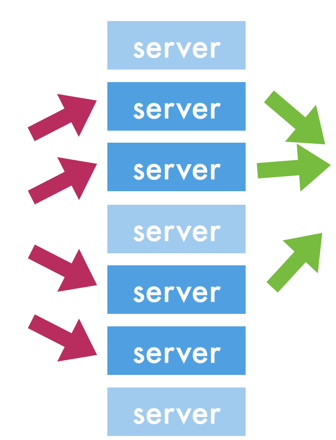


courtesy of Raghu Ramakrishnan

Systems

- Hardware
  CPU, RAM, GPU, disks, switches, server centers
- Data
  text, video, images, clicks, networks, location
- Parallelization strategies
  consistent (proportional) hashing, trees, P2P
- Storage
  RAID, GFS, Hadoop, Ceph
- Processing
  MapReduce, Pregel, Dryad, S4
- Databases / (key,value)
  BigTable, Pnuts, Cassandra





Further reading

- Consistent hashing (Karger et al.)
  http://www.akamai.com/dl/technical_publications/ConsistenHashingandRandomTreesDistributedCachingprotocolsforrelievingHotSpotsontheworldwideweb.pdf
- Stateless Proportional Caching (Chawla et al.)
  http://www.usenix.org/event/atc11/tech/final_files/Chawla.pdf
  http://www.usenix.org/event/atc11/tech/slides/chawla.pdf
- Pastry P2P routing (Rowstron and Druschel)
  http://research.microsoft.com/en-us/um/people/antr/PAST/pastry.pdf
  http://research.microsoft.com/en-us/um/people/antr/pastry/
- MapReduce (Dean and Ghemawat)
  http://labs.google.com/papers/mapreduce.html
- Google File System (Ghemawat, Gobioff, Leung)
  http://labs.google.com/papers/gfs.html
- Amazon Dynamo (deCandia et al.)
  http://cs.nyu.edu/srg/talks/Dynamo.ppt
  http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf
- BigTable (Chang et al.)
  http://labs.google.com/papers/bigtable.html
- CEPH filesystem (proportional hashing, file system)
  http://ceph.newdream.net/
  http://ceph.newdream.net/papers/weil-crush-sc06.pdf

Further reading

- CPUS
  http://www.anandtech.com/show/3922/intels-sandy-bridge-architecture-exposed
  http://www.anandtech.com/show/4991/arms-cortex-a7-bringing-cheaper-dualcore-more-power-efficient-highend-devices
- NVIDIA CUDA
  http://www.nvidia.com/object/cuda_home_new.html
- ATI Stream Computing
  http://www.amd.com/US/PRODUCTS/TECHNOLOGIES/STREAM-TECHNOLOGY/Pages/stream-technology.aspx
- Microsoft Dryad (Isard et al.)
  http://connect.microsoft.com/Dryad
- Yahoo S4 (Neumayer et al.)
  http://s4.io/
  http://slidesha.re/uSdSjL (slides)
  http://4lunas.org/pub/2010-s4.pdf (paper)
- Memcached
  http://memcached.org/
- Linked.In Voldemort (key,value) storage
  http://project-voldemort.com/design.php
- PNUTS distributed storage (Cooper et al.)
  http://www.brianfrankcooper.net/pubs/pnuts.pdf
- SSDs (solid state drives)
  http://www.anandtech.com/bench/SSD/65