MACHINE LEARNING DEPARTMENT

3.1 Architecture

3 Systems

Alexander Smola

Introduction to Machine Learning 10-701

http://alex.smola.org/teaching/10-701-15

Carnegie Mellon University


Real Hardware
google.com/datacenters
Google
tweden
049

Machines

Bulk transfer is at least 10x faster

- CPU
  - 8-64 cores (Intel/AMD servers)
  - 2-3 GHz (close to 1 IPC per core peak) - over 100 GFlops/socket
  - 8-32 MB Cache (essentially accessible at clock speed)
  - Vectorized multimedia instructions (AVX 256bit wide, e.g. add, multiply, logical)

- RAM
  - 16-256 GB depending on use
  - 3-8 memory banks (each 32bit wide - atomic writes!)
  - DDR3 (up to 100GB/s per board, random access 10x slower)

- Harddisk
  - 4 TB/disk
  - 100 MB/s sequential read from SATA2
  - 5ms latency for 10,000 RPM drive, i.e. random access is slow

- Solid State Drives
  - 500 MB/s sequential read
  - Random writes are really expensive (read-erase-write cycle for a block)

The real joy of 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.

Jeff Dean's Stanford slides

Why a single machine is not enough

- Data (lower bounds)
  - 10-100 Billion documents (webpages, e-mails, ads, tweets)
  - 100-1000 Million users on Google, Facebook, Twitter, Hotmail
  - 1 Million days of video on YouTube
  - 100 Billion images on Facebook
- Processing capability for single machine 1TB/hour
  But we have much more data
- Parameter space for models is too big for a single machine
  Personalize content for many millions of users
- Process on many cores and many machines simultaneously

Cloud pricing

- Google Compute Engine and Amazon EC2

Instance type	Virtual Cores	Memory	Price (US$)/Hour (US hosted)
n1-standard-1	1	3.75GB	$0.070
n1-standard-2	2	7.5GB	$0.140
n1-standard-4	4	15GB	$0.280
n1-standard-8	8	30GB	$0.560
n1-standard-16	16	60GB	$1.120

$10,000/year

- Storage

Standard Provisioned Space	$0.04 GB / month
SSD Provisioned Space	$0.325 GB / month
Snapshot storage	$0.125 GB / month
IO operations	No additional charge

Spot instances much cheaper

Amazon EBS General Purpose (SSD) volumes
- $0.10 per GB-month of provisioned storage

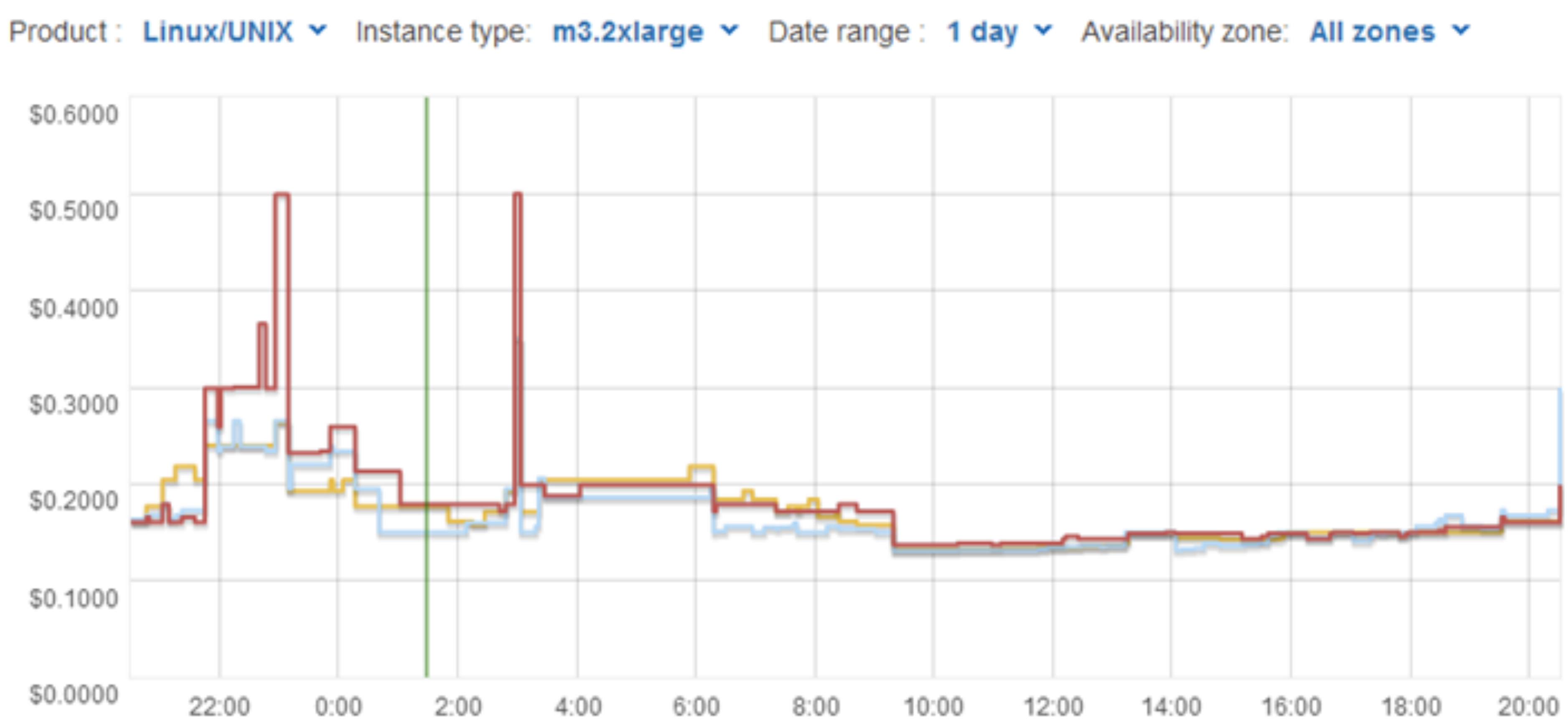
Amazon EBS Provisioned IOPS (SSD) volumes
- $0.125 per GB-month of provisioned storage
- $0.10 per provisioned IOPS-month

Amazon EBS Magnetic volumes
- $0.05 per GB-month of provisioned storage
- $0.05 per 1 million I/O requests

Amazon EBS Snapshots to Amazon S3
- $0.095 per GB-month of data stored

Real Hardware

- Can and will fail
- Spot instances much cheaper (but can lead to preemption). Design algorithms for it!



Carnegie Mellon University

Distribution Strategies

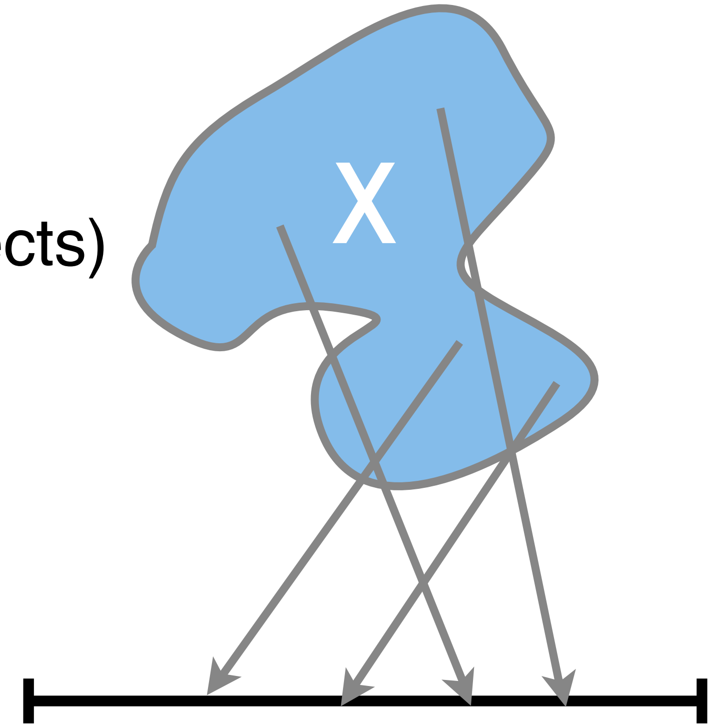
Concepts

- 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

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

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

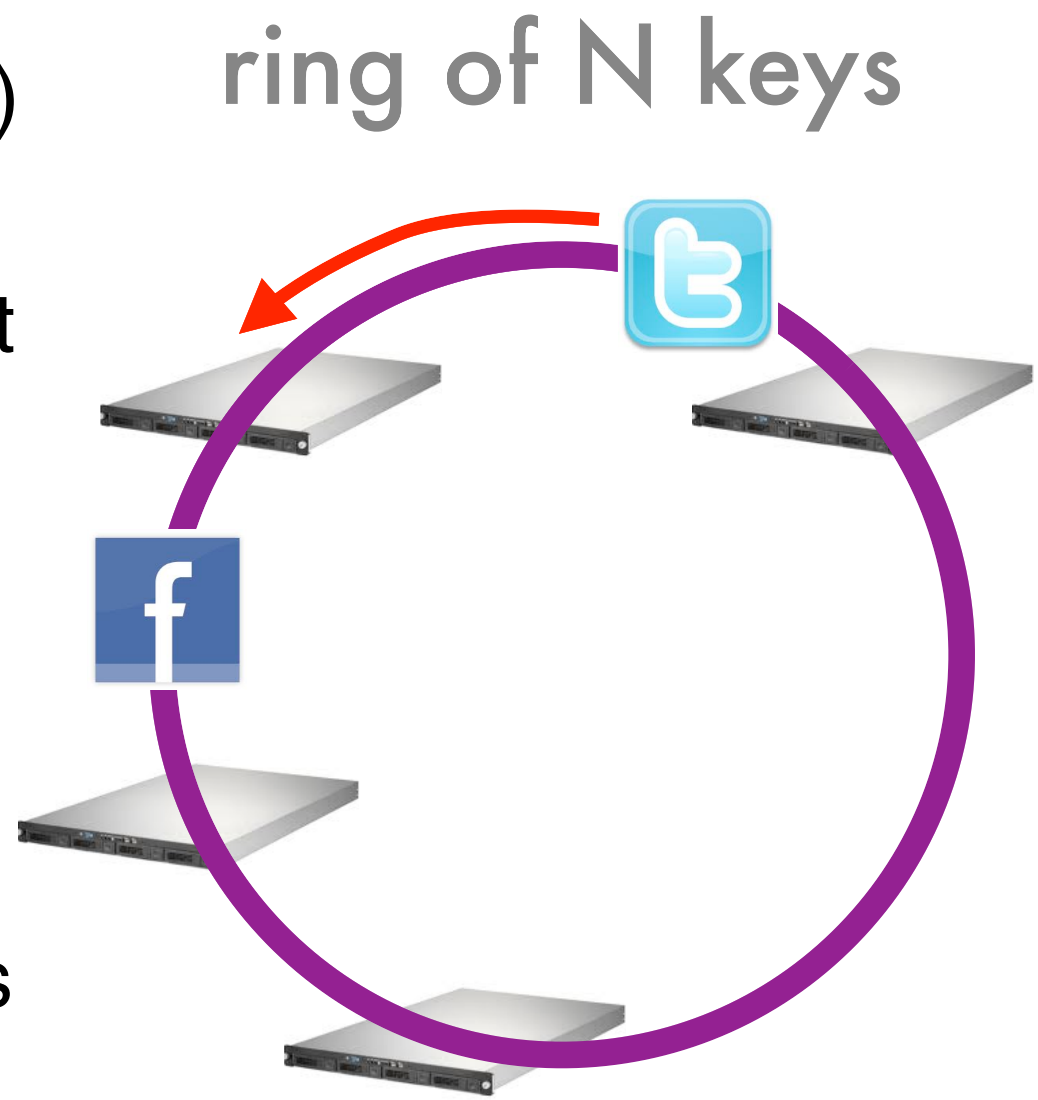
- Consistent hashing with k replications

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

  - 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

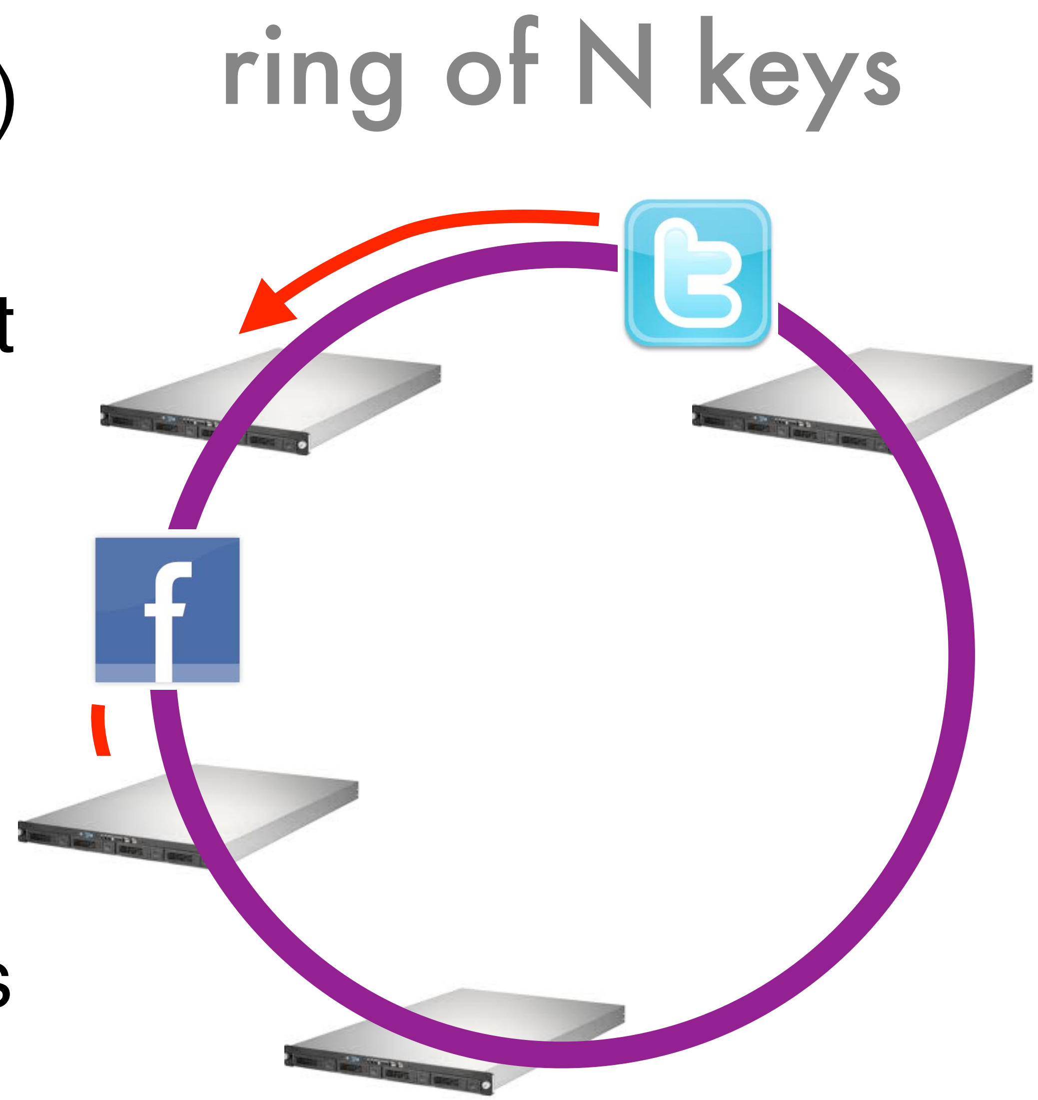
Distributed Hash Table

- 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)



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

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

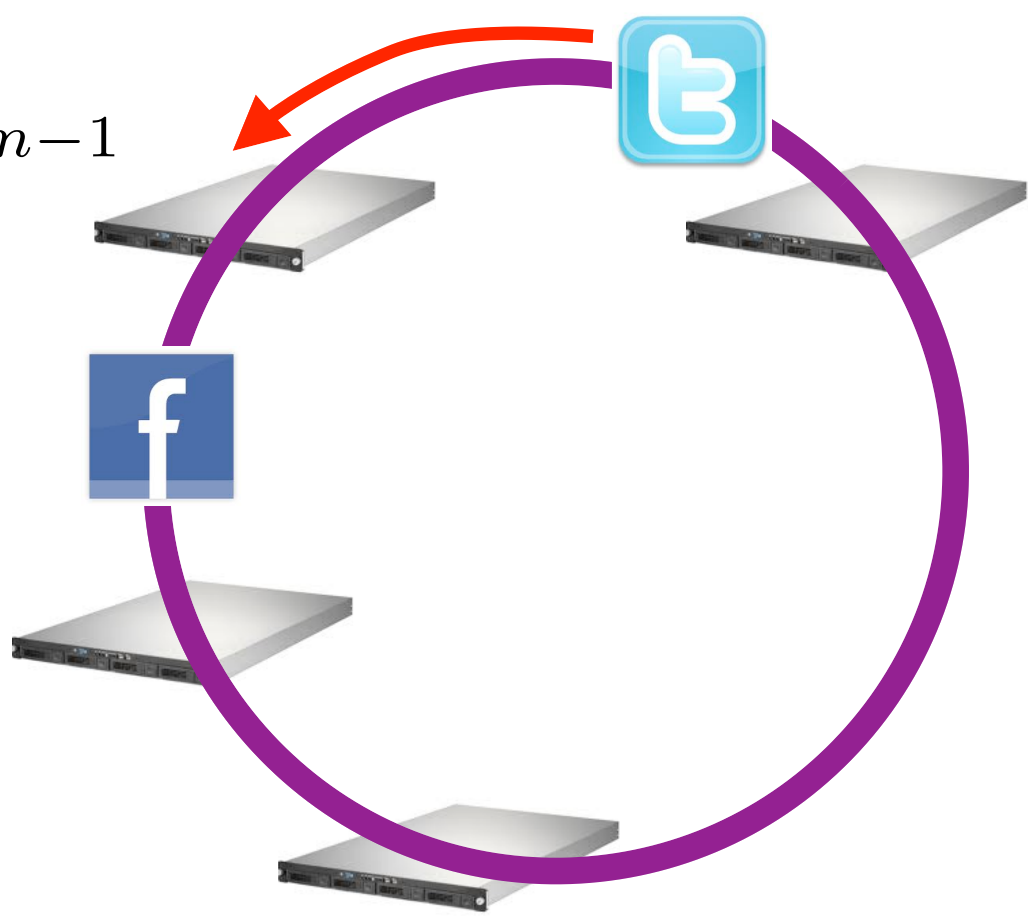
- Density is given by derivative

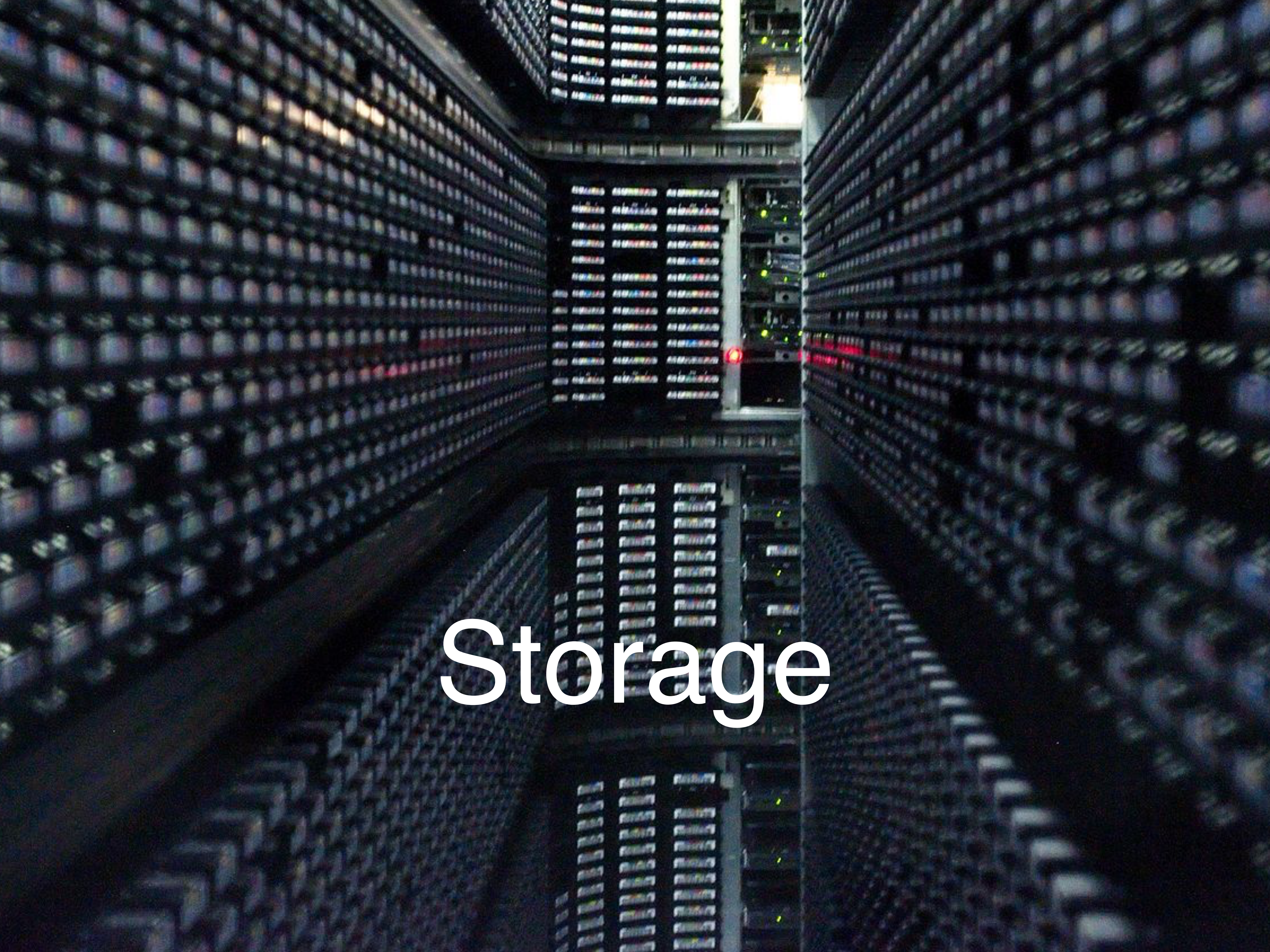
$$p(c) = (m-1)(1-c)^{m-2}$$

- Expected segment length is (follows from symmetry) $c = \frac{1}{m}$
- Probability of exceeding expected segment length (for large m)

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

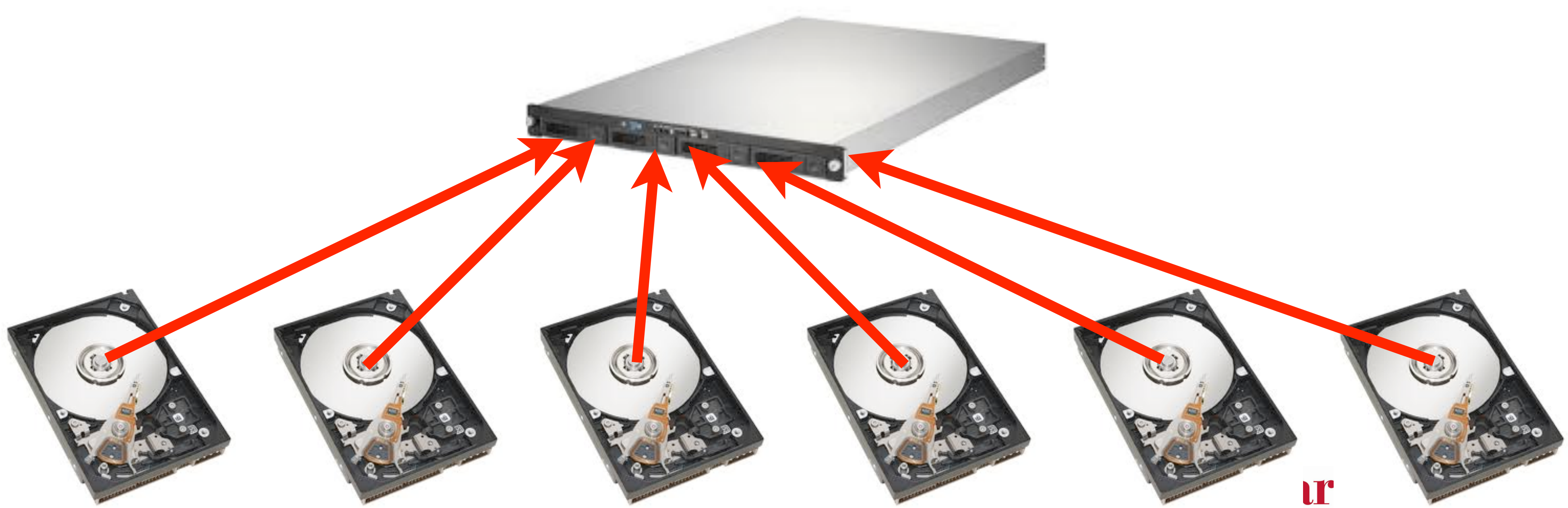
ring of N keys




Storage

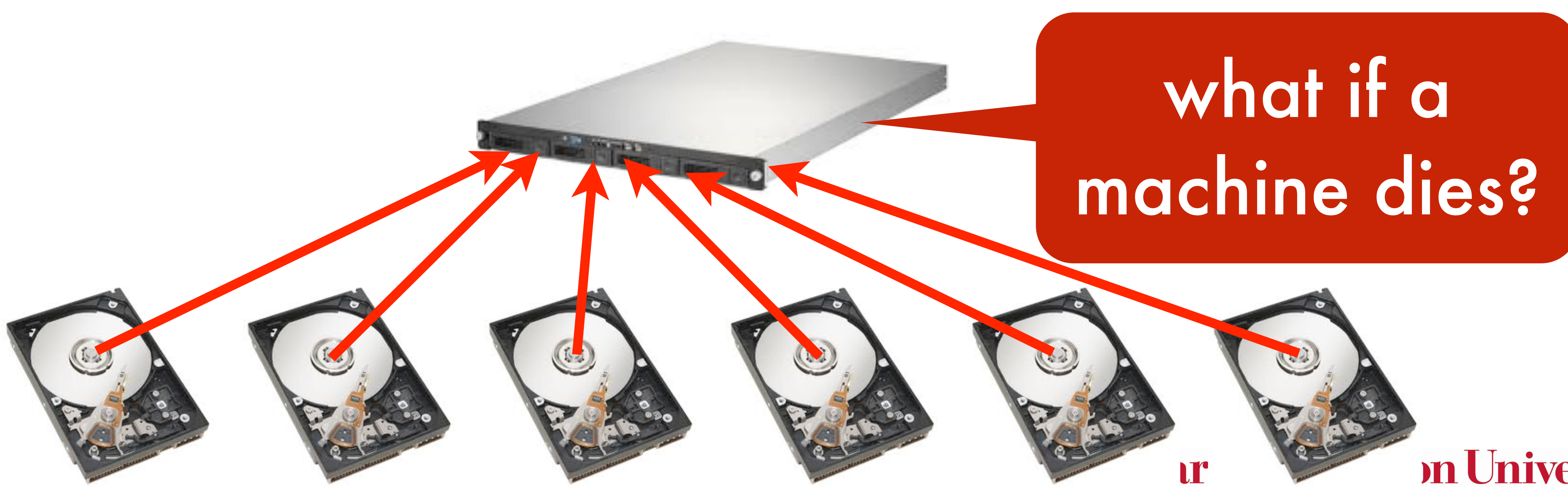
RAID

- Redundant array of inexpensive disks (optional fault tolerance)
  - Aggregate storage of many disks
  - Aggregate bandwidth of many disks
- 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

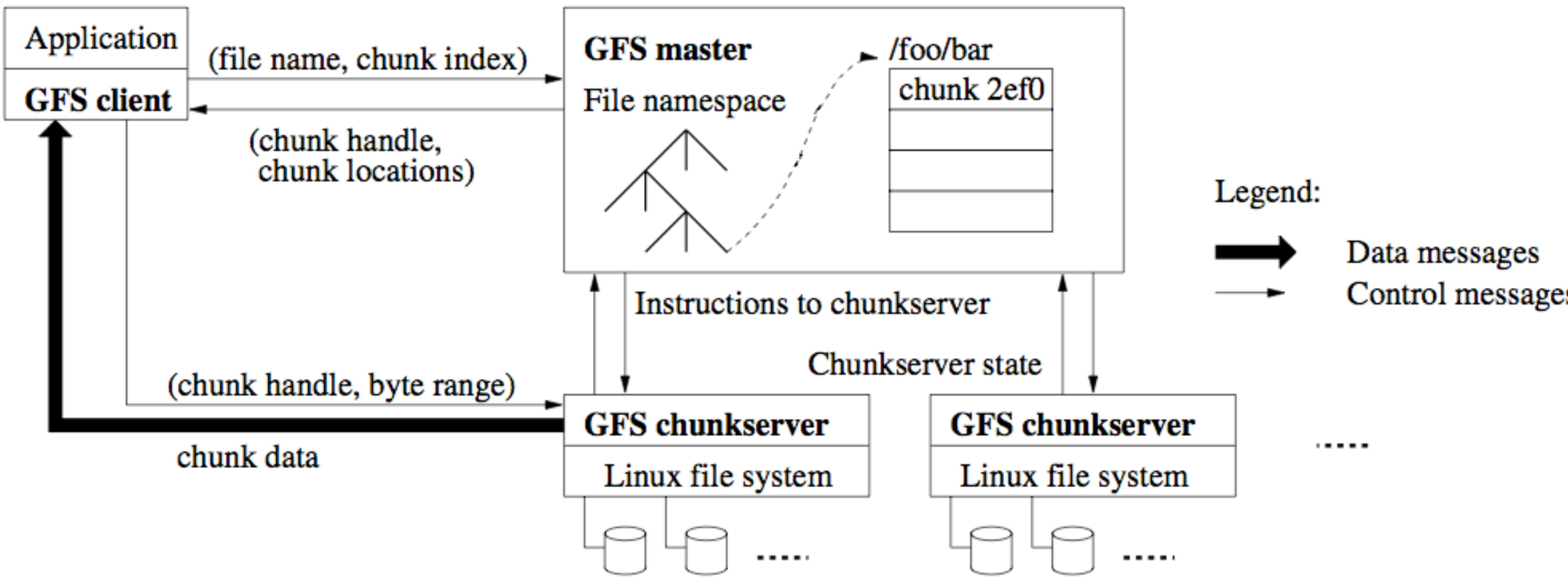
- Redundant array of inexpensive disks (optional fault tolerance)
  - Aggregate storage of many disks
  - Aggregate bandwidth of many disks
- RAID 0 - stripe data over disks (good bandwidth, faulty)
- RAID 1 - mirror disks (mediocre bandwidth, fault tolerance)
- RAID 5 - stripe data with 1 disk for parity (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 / HadoopFS

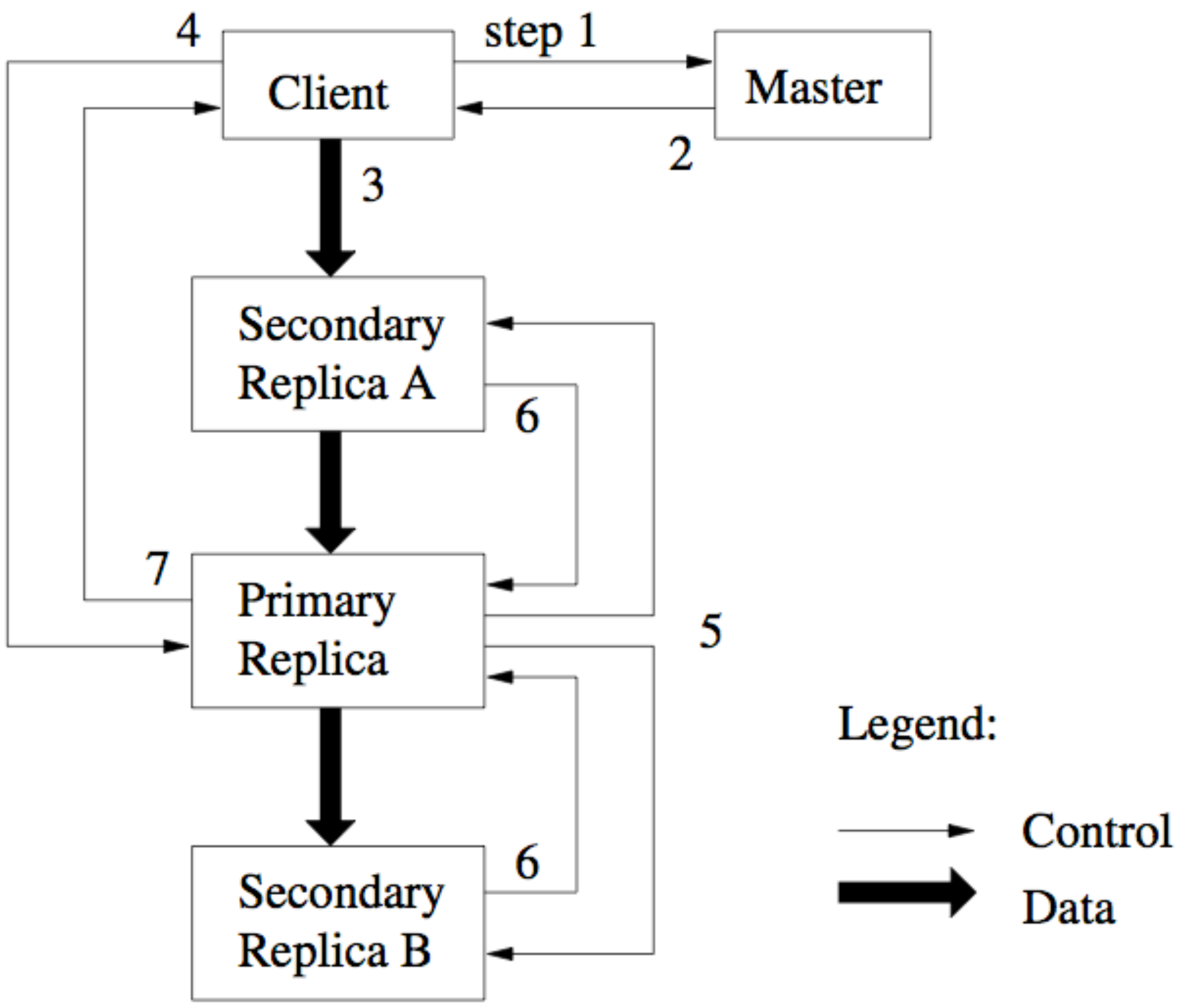


Ghemawat, Gobioff, Leung, 2003

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

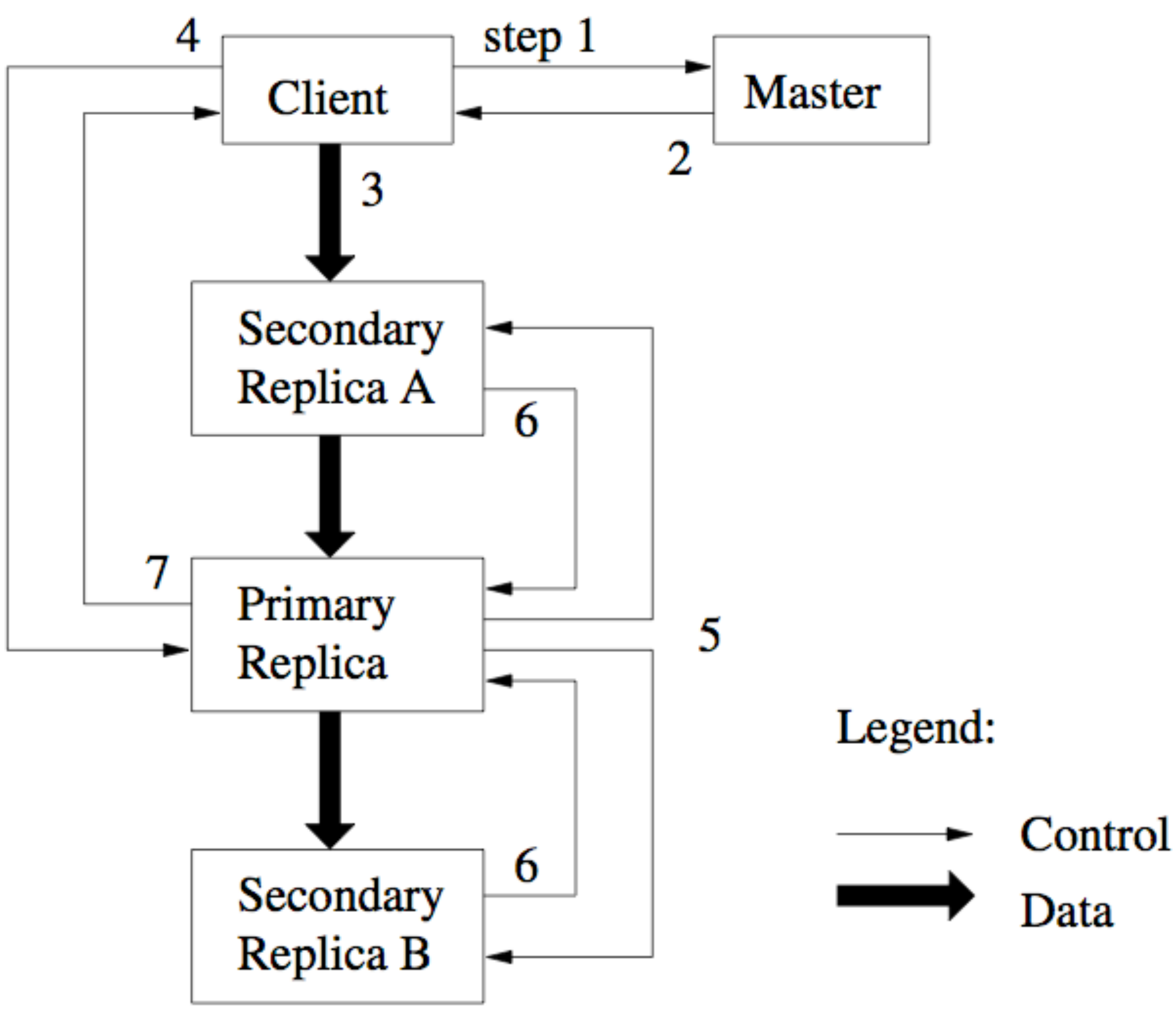
Google File System / HDFS

- Client requests chunk from master
- Master responds with replica location
- Client writes to replica A
- Client notifies primary replica
- Primary replica requests data from replica
- Replica A sends data to Primary replica (s
- Primary replica confirms write to client



Google File System / HDFS

- Client requests chunk from master
- Master responds with replica location
- Client writes to replica A
- Client notifies primary replica
- Primary replica requests data from replica
- Replica A sends data to Primary replica (s
- Primary replica confirms write to client

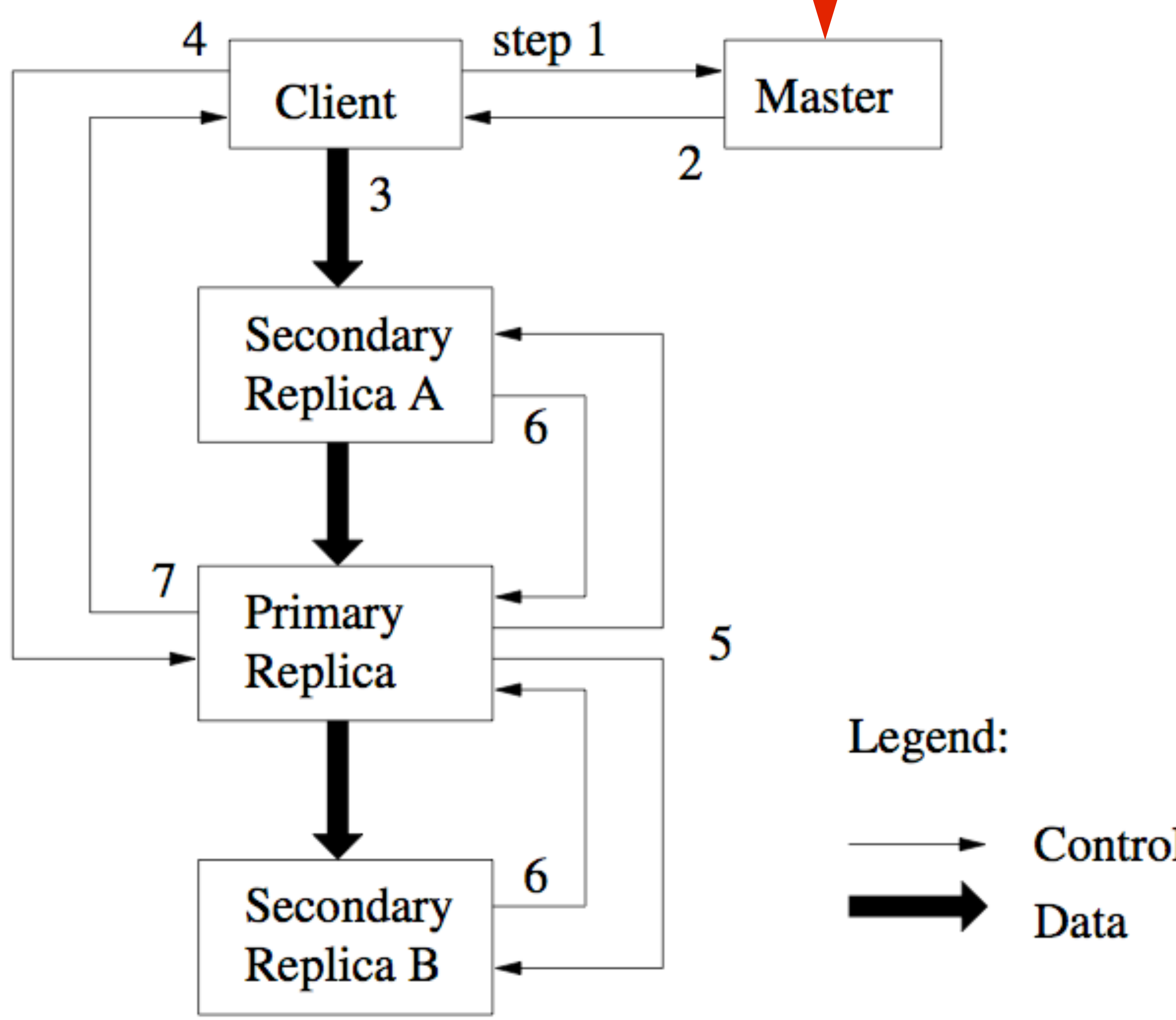


- Master ensures nodes are live
- Chunks are checksummed
- Can control replication factor for hotspots / load balancing
- Deserialize master state by loading data structure as flat file from disk (fast)

Google File System / HDFS

- Client requests chunk from master
- Master responds with replica location
- Client writes to replica A
- Client notifies primary replica
- Primary replica requests data from replica
- Replica A sends data to Primary replica (s
- Primary replica confirms write to client

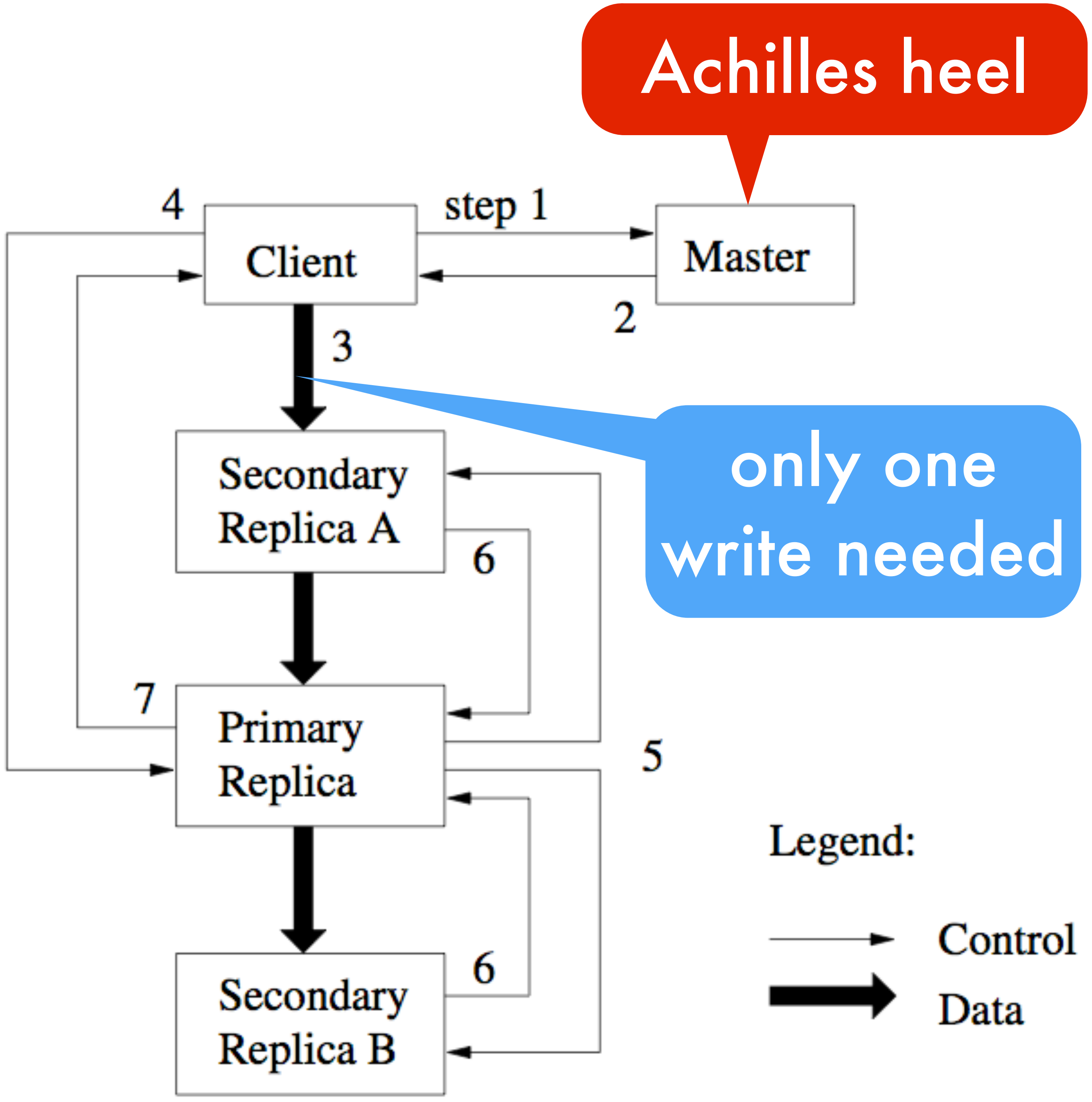
Achilles heel



- 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

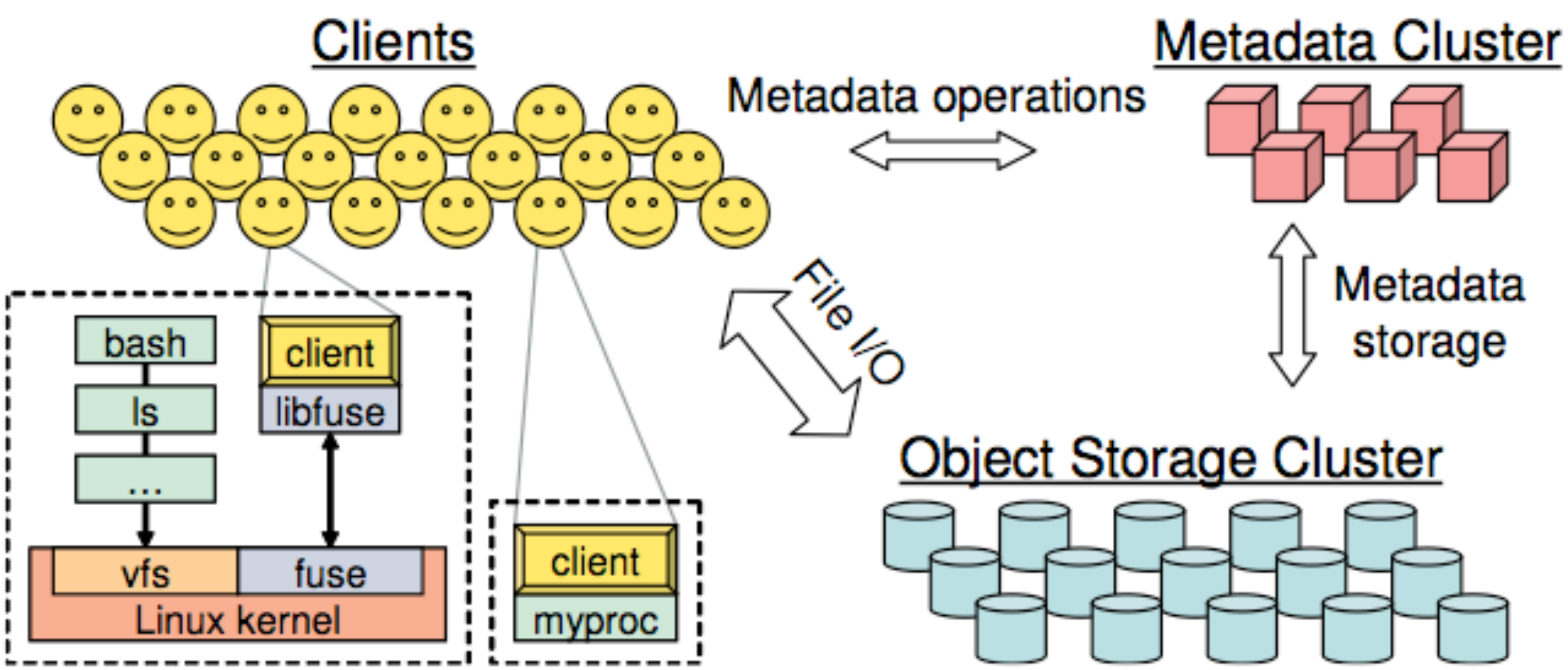
- Client requests chunk from master
- Master responds with replica location
- Client writes to replica A
- Client notifies primary replica
- Primary replica requests data from replica
- Replica A sends data to Primary replica (s
- Primary replica confirms write to client



- 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)

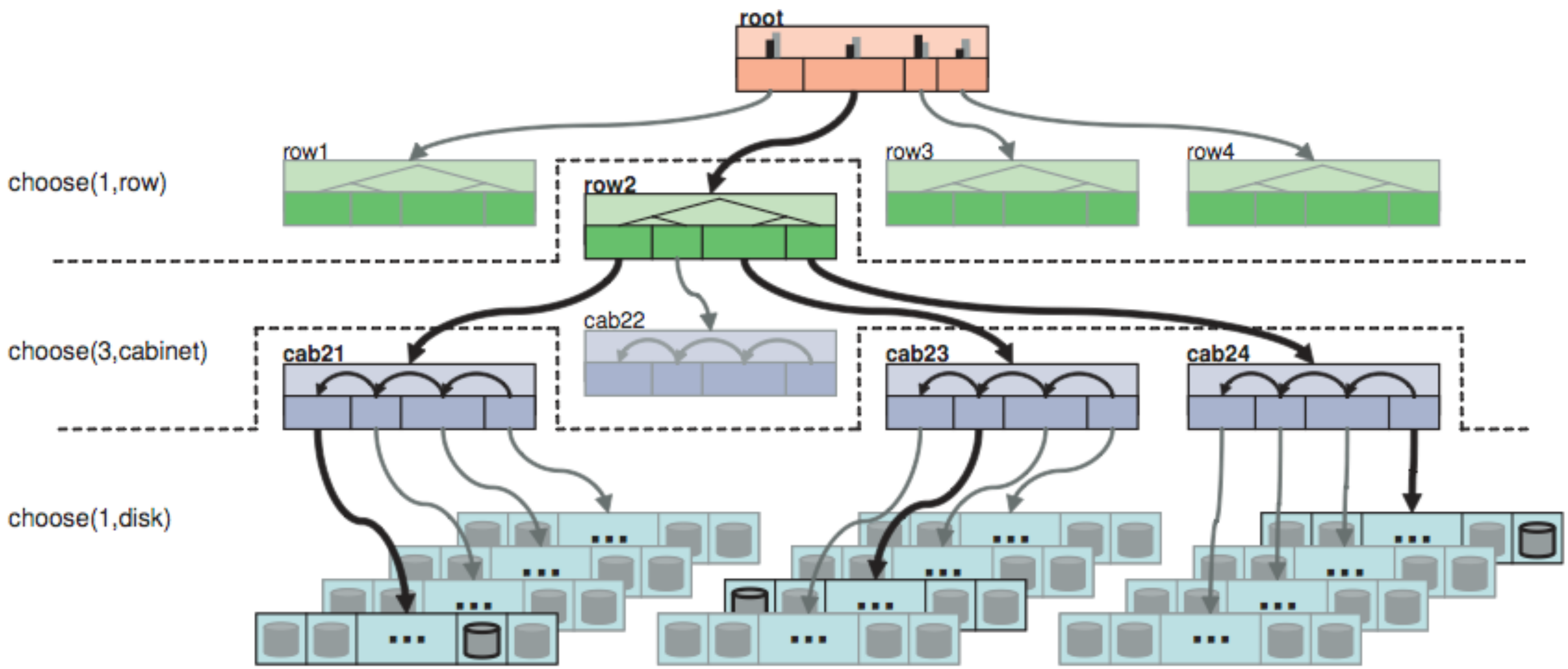
CEPH/CRUSH

- 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?



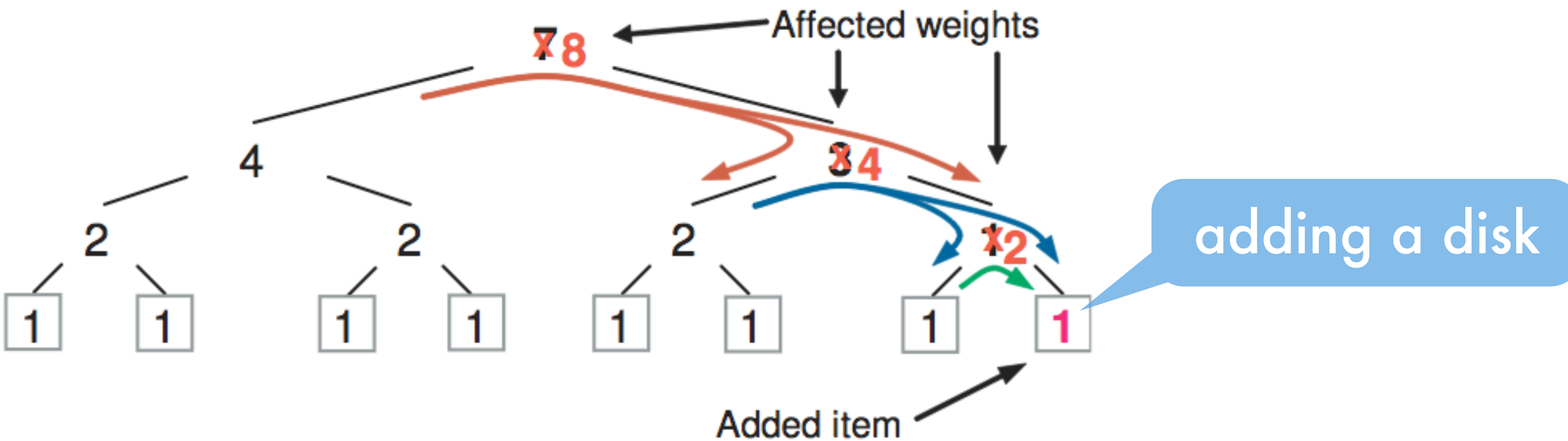
http://ceph.newdream.org (Weil et al., 2006)

CEPH/CRUSH



- Various sampling schemes (ensure that no unnecessary 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

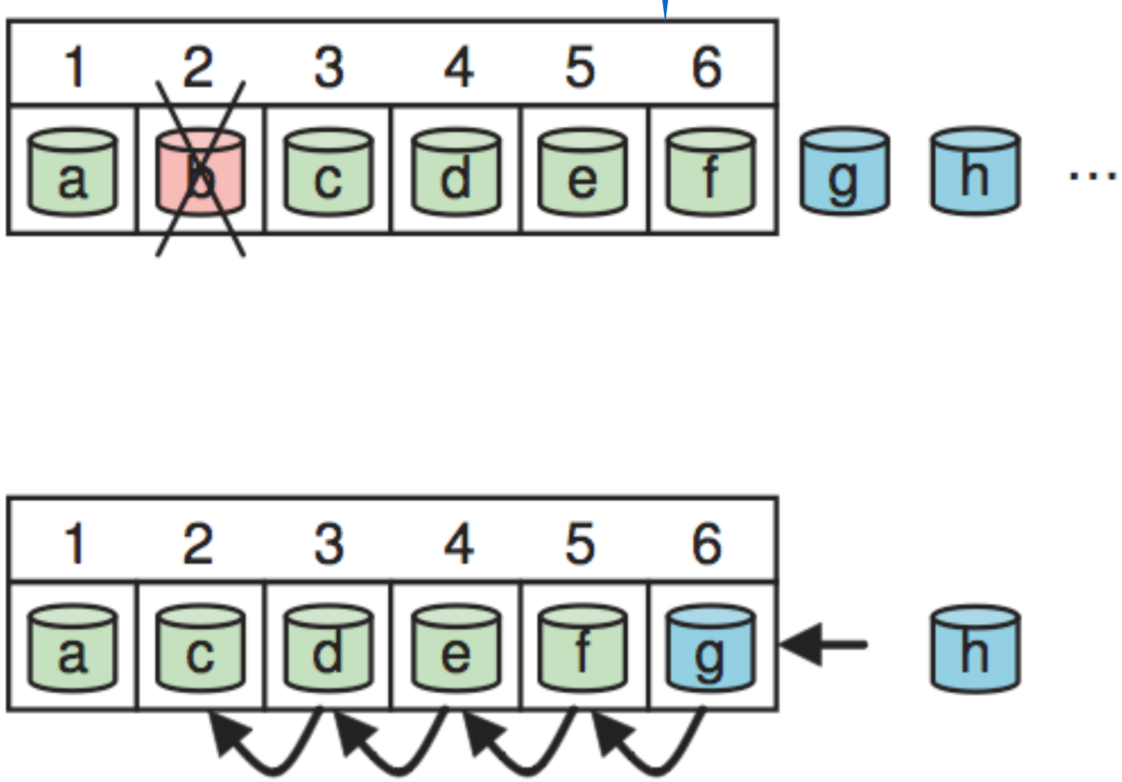


- Various sampling schemes (ensure that no unneccessary data is moved)
- In the simplest case proportional consistent hashing from pool of objects (pick k disks out of n for block with given ID)
- Can incorporate replication/bandwidth scaling like RAID (stripe block over several disks, error correction)

CEPH/CRUSH fault

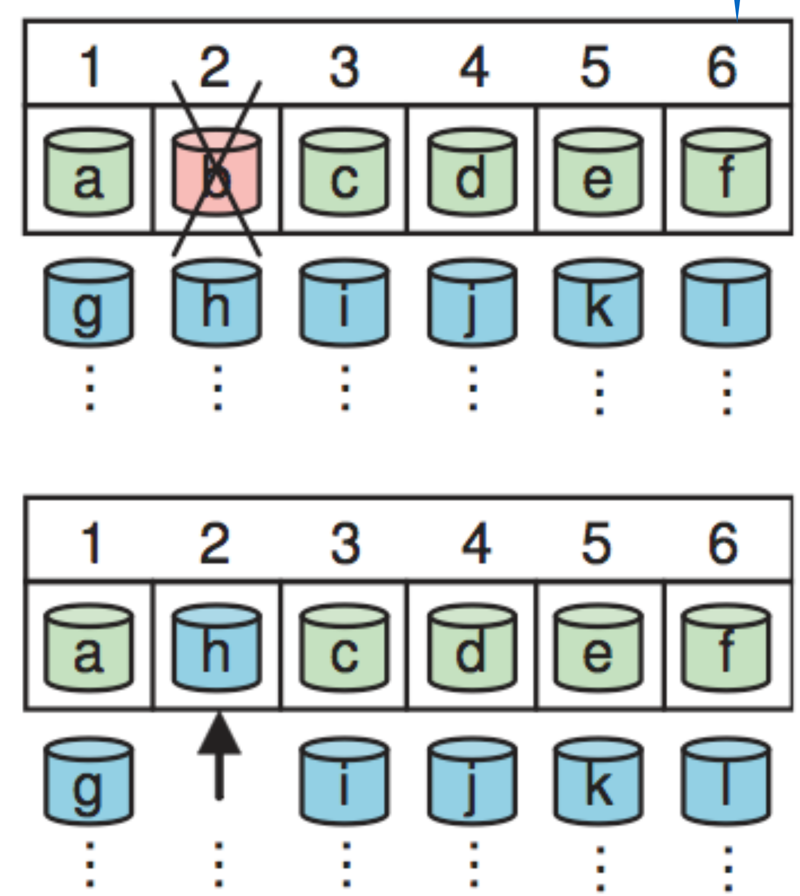
plain replication

$r' = r + f$



striped data

$r' = r + f_r n$



Carnegie Mellon University

MACHINE LEARNING DEPARTMENT

3.2 Processing

3 Systems

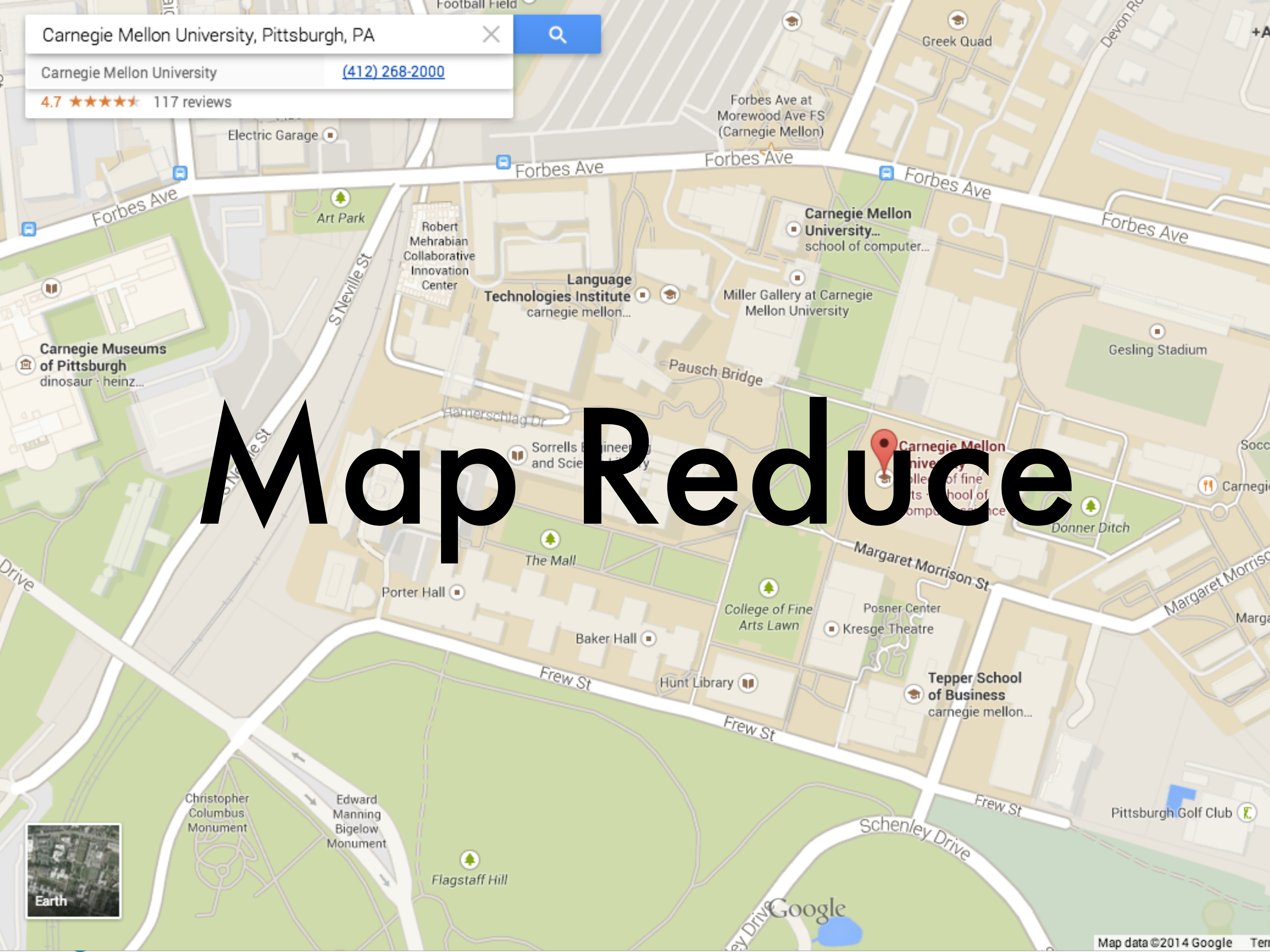
Alexander Smola

Introduction to Machine Learning 10-701

http://alex.smola.org/teaching/10-701-15

Carnegie Mellon University

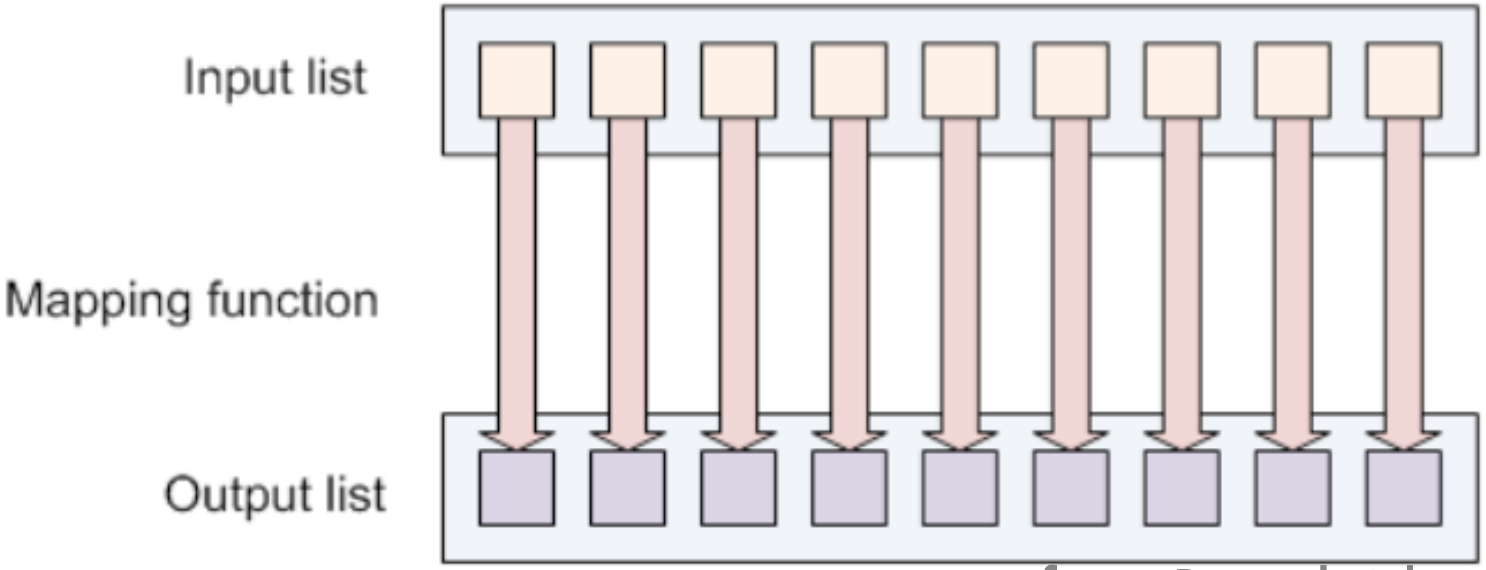
Map Reduce

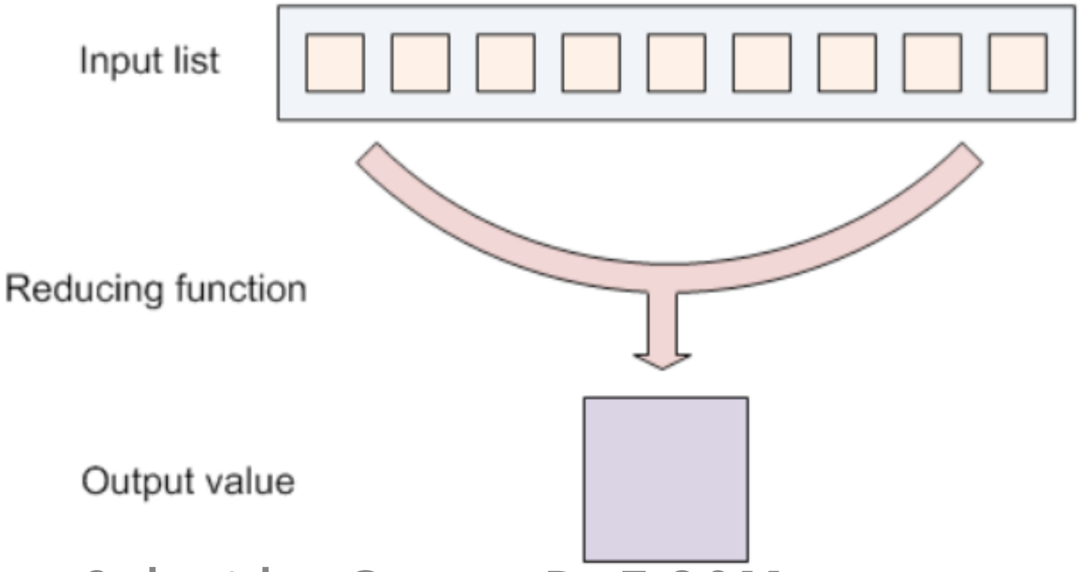


Carnegie Mellon University, Pittsburgh, PA
Carnegie Mellon University
(412) 268-2000
4.7 117 reviews
Football Field
Greek Quad
Electric Garage
Forbes Ave at Morewood Ave FS (Carnegie Mellon)
Forbes Ave
Art Park
Robert Mehrabian Collaborative Innovation Center
Carnegie Mellon University... school of computer...
Language Technologies Institute carnegie mellon...
Miller Gallery at Carnegie Mellon University
S Neville St
Carnegie Museums of Pittsburgh dinosaur · heinz...
Gesling Stadium
Pausch Bridge
Hamerschlag Dr
Donner Ditch
The Mall
Margaret Morrison St
Porter Hall
College of Fine Arts Lawn
Posner Center
Kresge Theatre
Baker Hall
Frew St
Hunt Library
Tepper School of Business carnegie mellon...
Christopher Columbus Monument
Edward Manning Bigelow Monument
Flagstaff Hill
Schenley Drive
Pittsburgh Golf Club
Earth
Google
Map data ©2014 Google

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

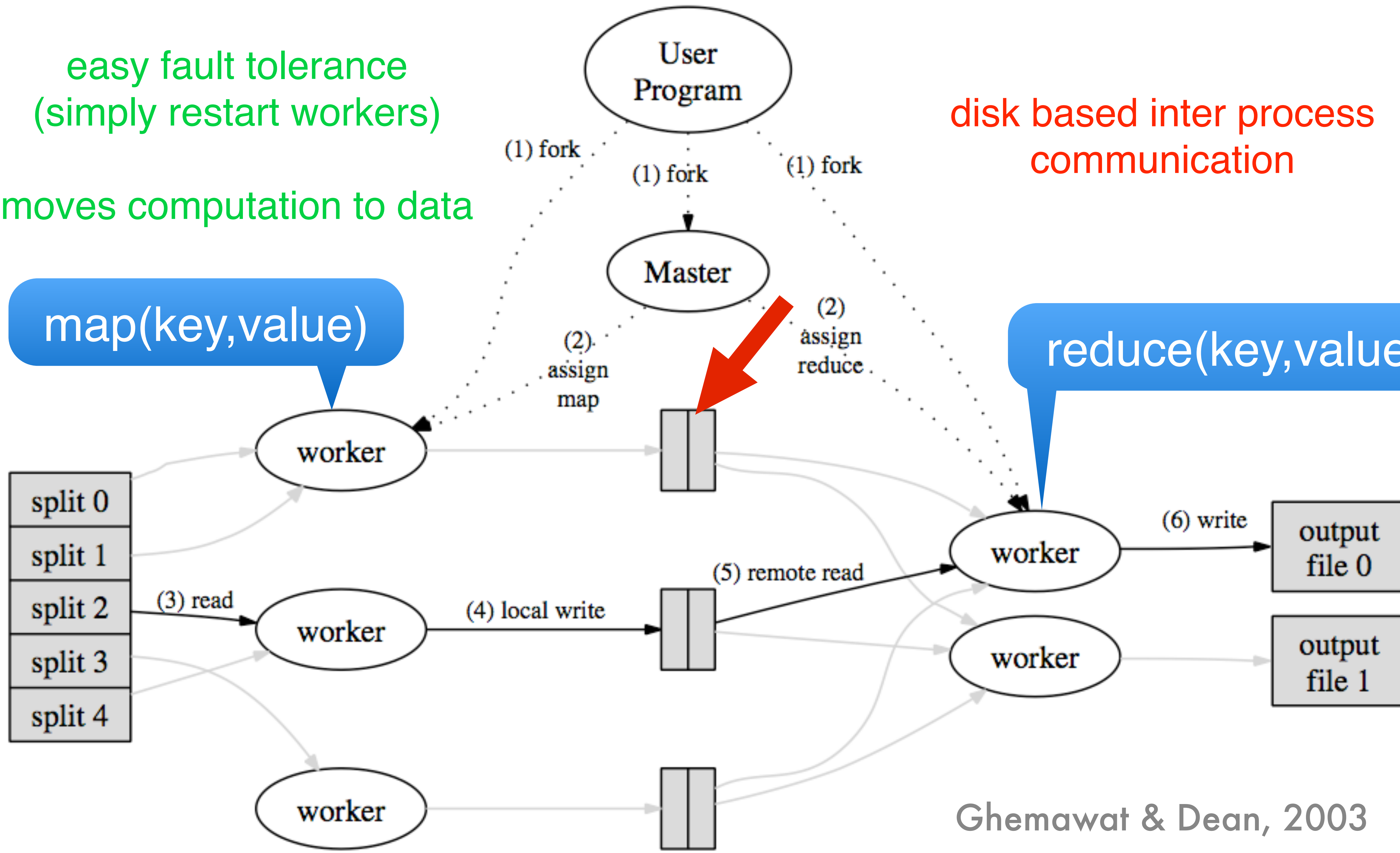
easy fault tolerance
(simply restart workers)

moves computation to data

disk based inter process
communication

map(key,value)

reduce(key,value)



Ghemawat & Dean, 2003

Map Combine Reduce

- Combine aggregates keys before sending to reducer (save 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

Example - Gradient Descent

- Objective

$$\underset{w}{\text{minimize}} \sum_{i=1}^{m} l(x_i, y_i, w) + \frac{\lambda}{2} \|w\|^2$$

- Algorithm
  - compute gradient

$$g := \sum_{i=1}^{m} \partial_w l(x_i, y_i, w)$$

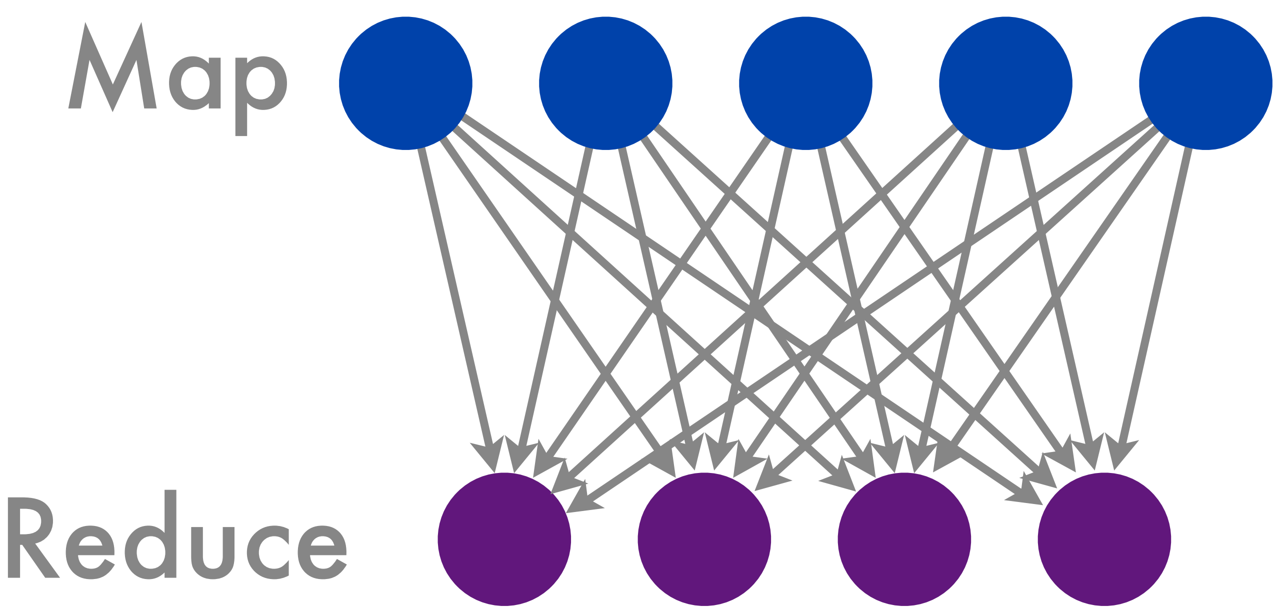
    - On each data point via Map(i,data)
    - Sum gradient via Reduce(coordinate)
  - perform update step (better with line search)

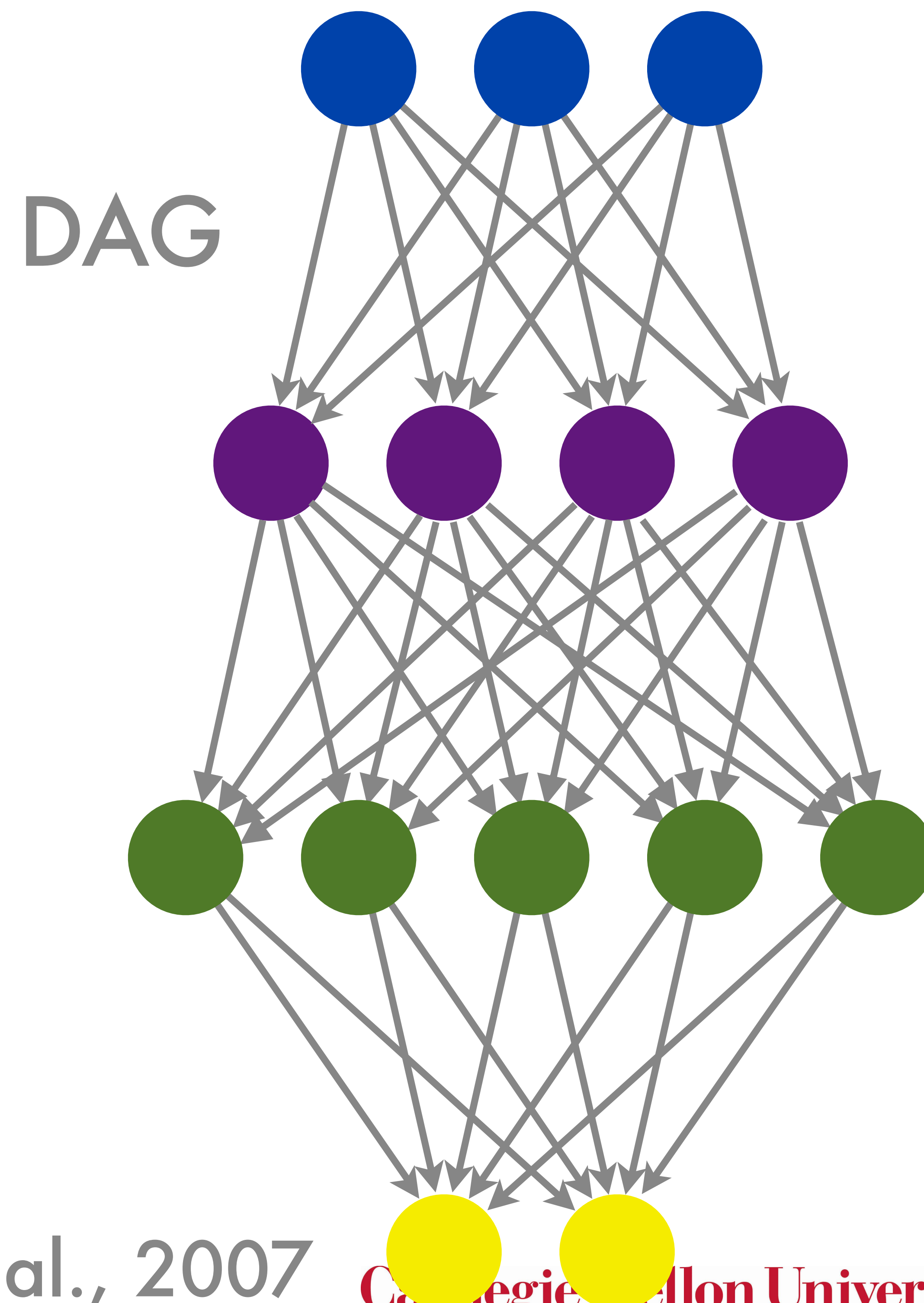
$$w \leftarrow w - \eta(g + \lambda w)$$

  - repeat


Dryad & S4

Dryad

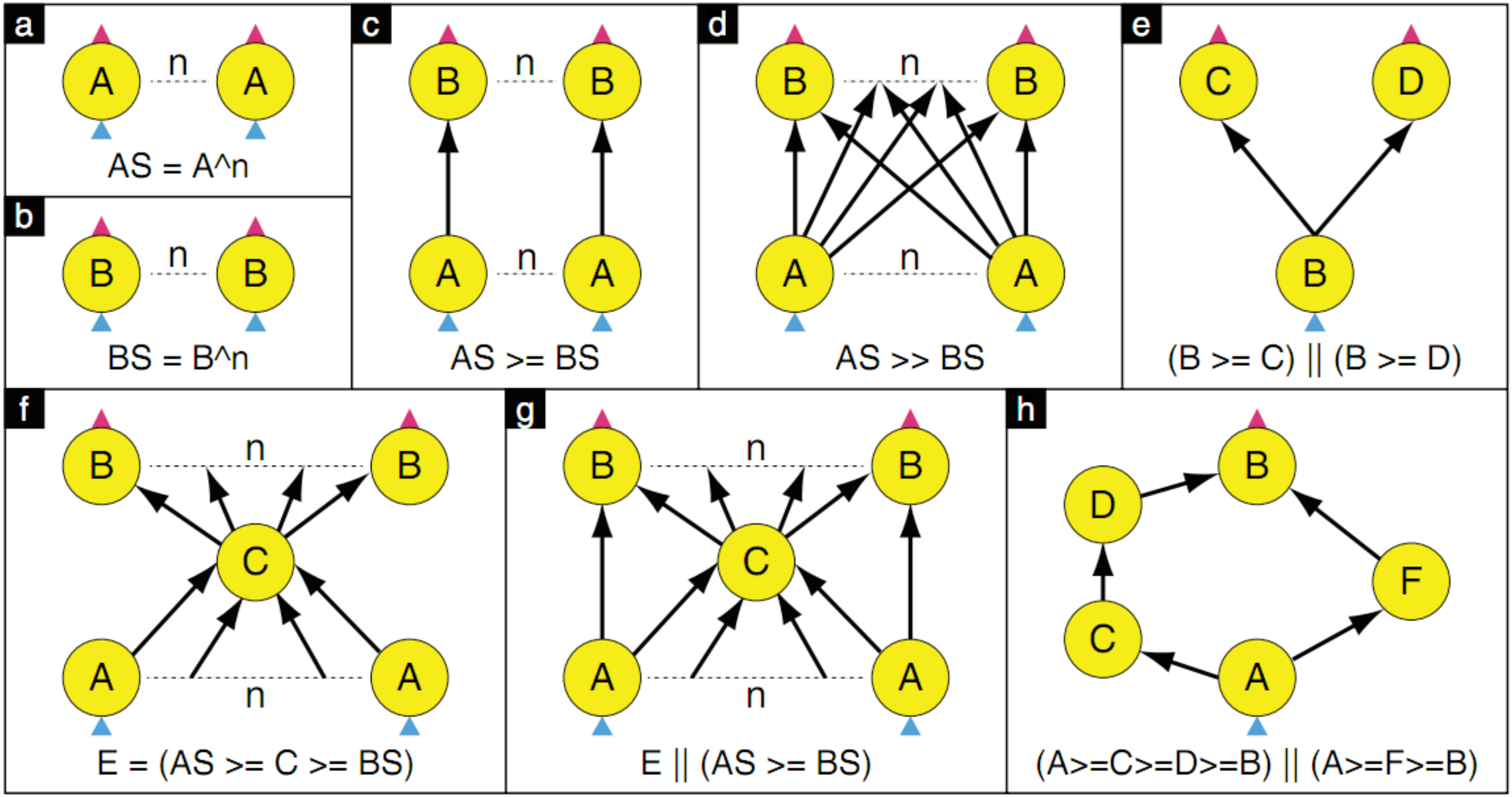




- Directed acyclic graph
- System optimizes parallelism
- Different types of IPC (memory FIFO/network/file)
- Tight integration with .NET (allows easy prototyping)

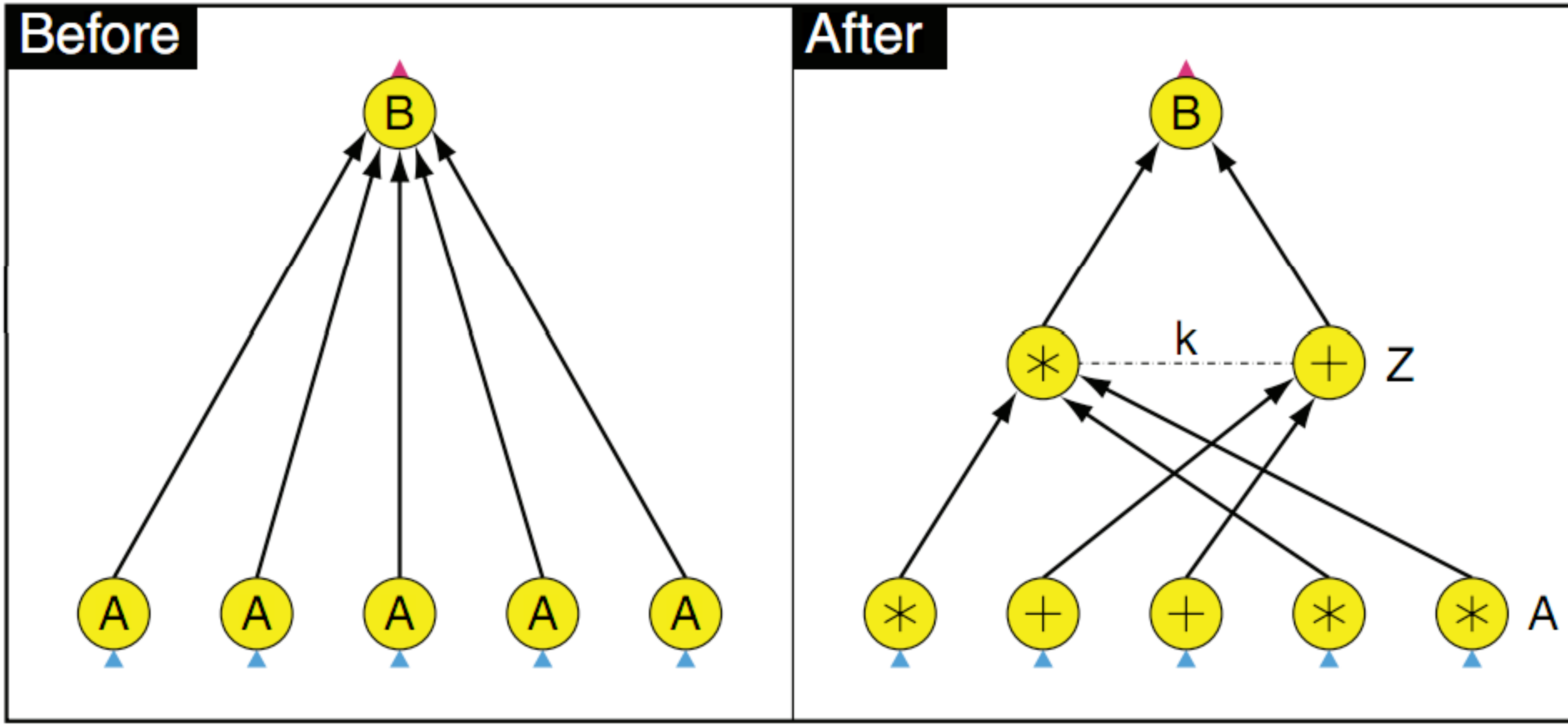
Isard et al., 2007

DRYAD



graph description language

DRYAD



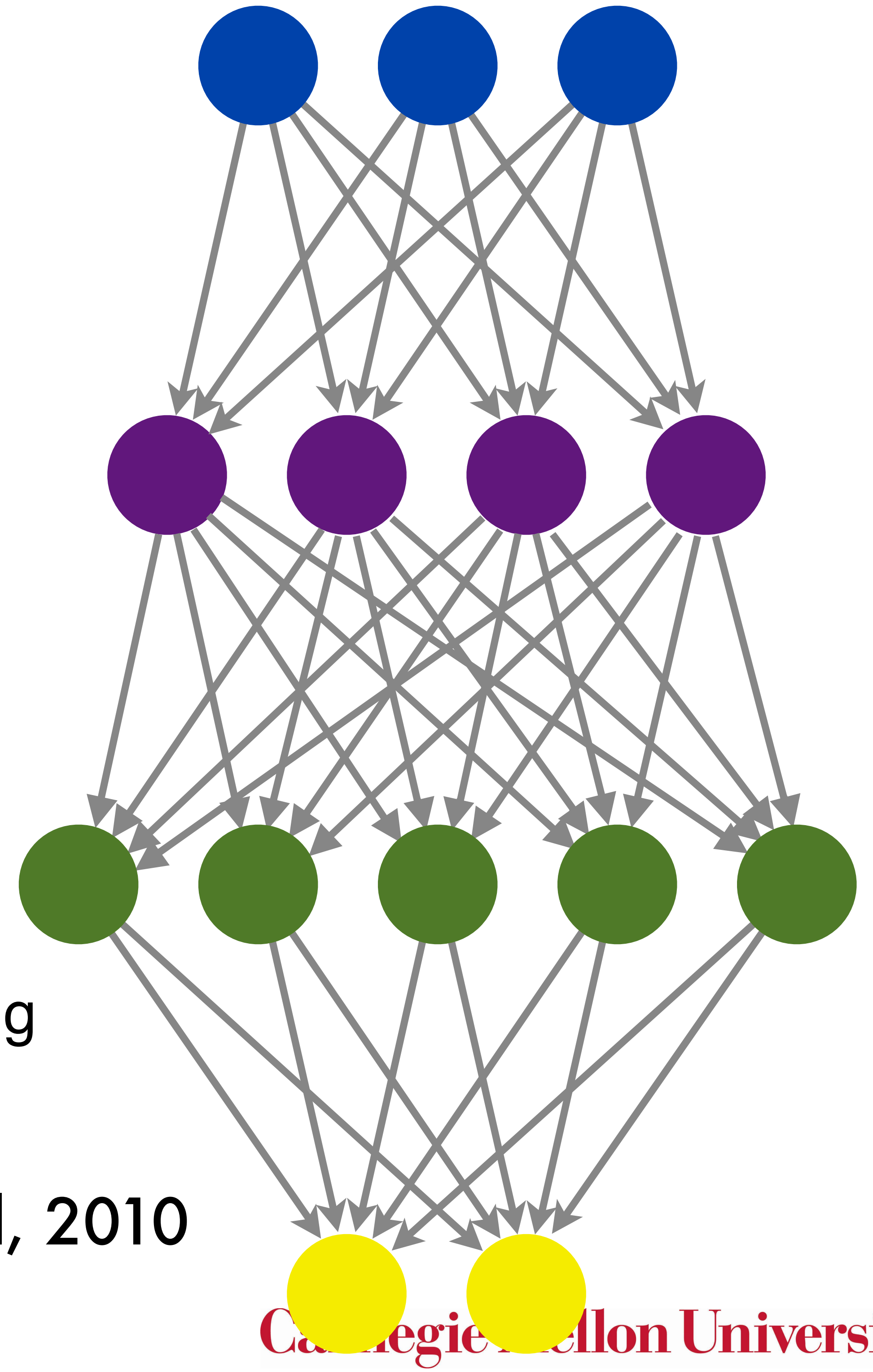
automatic graph refinement

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)

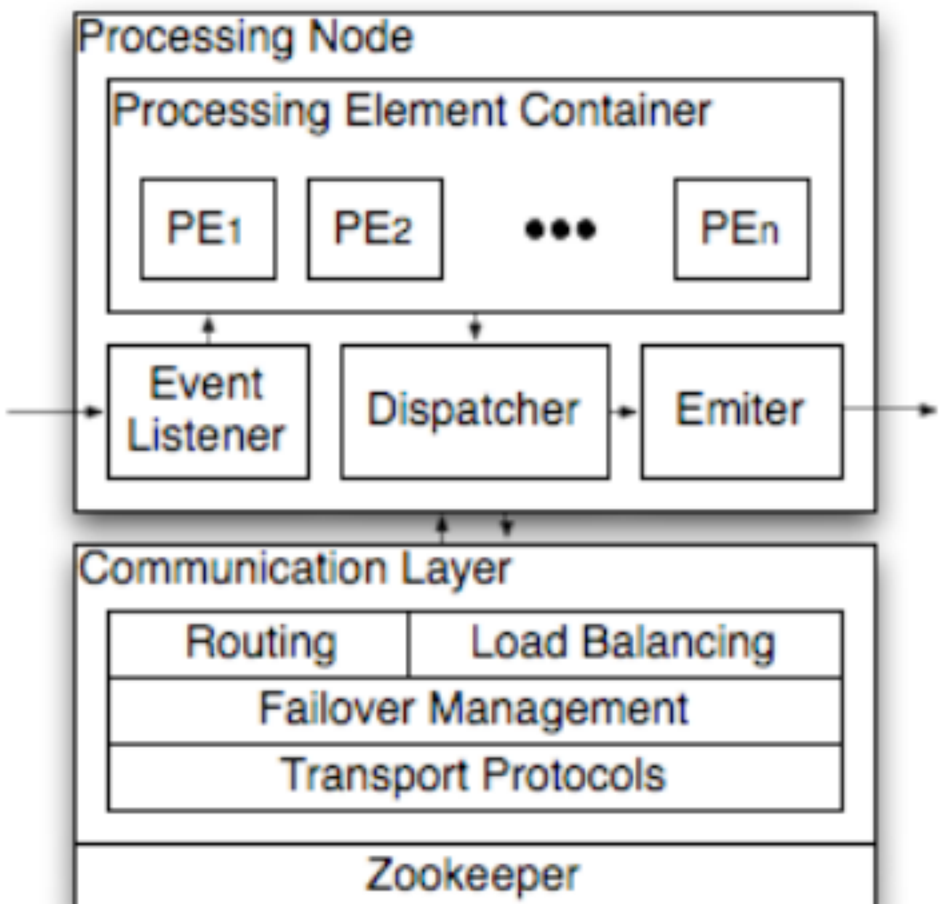
- Simple implementation e.g. via consistent hashing

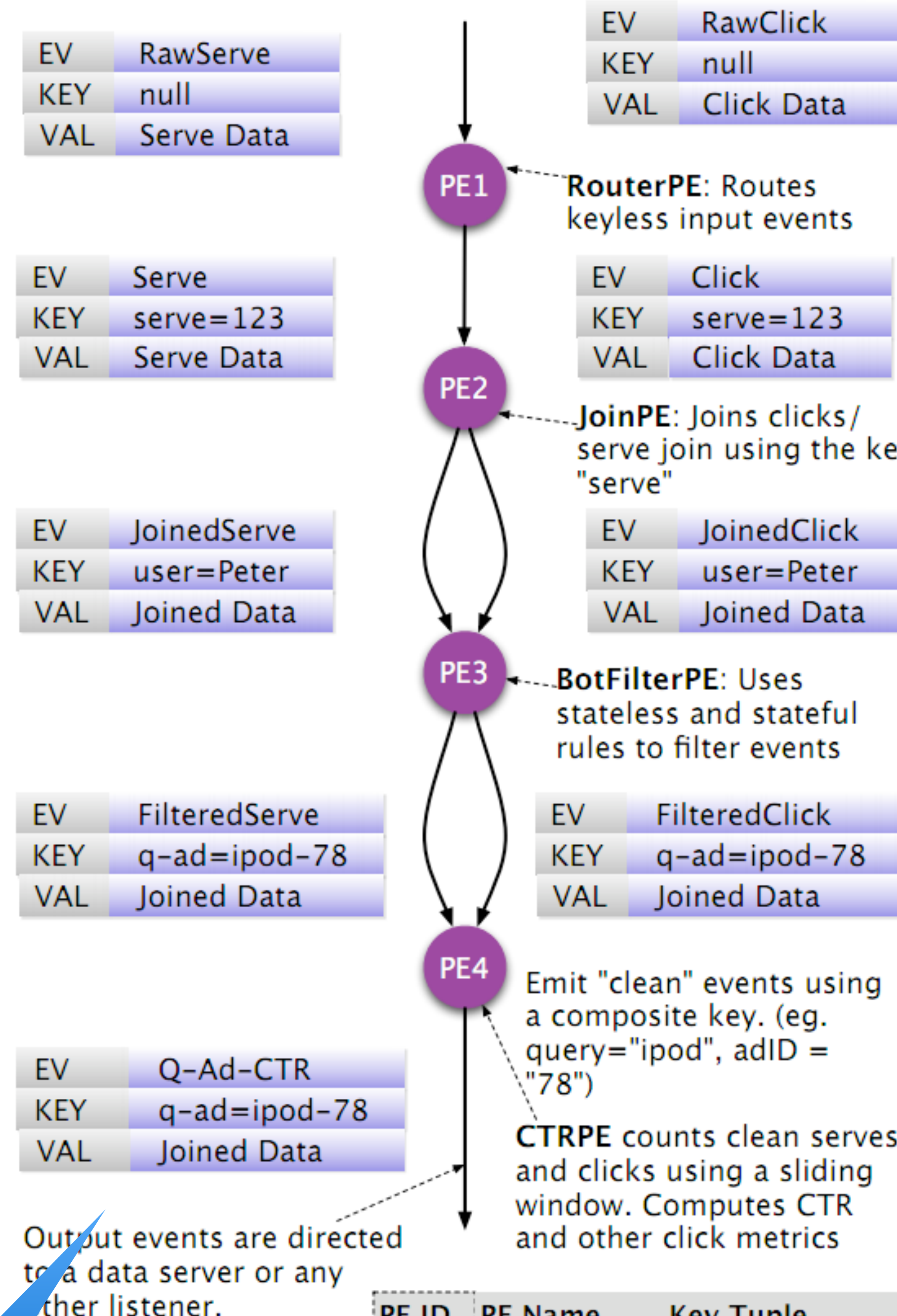


http://incubator.apache.org/s4/ Neumeyer et al, 2010

S4

processing element





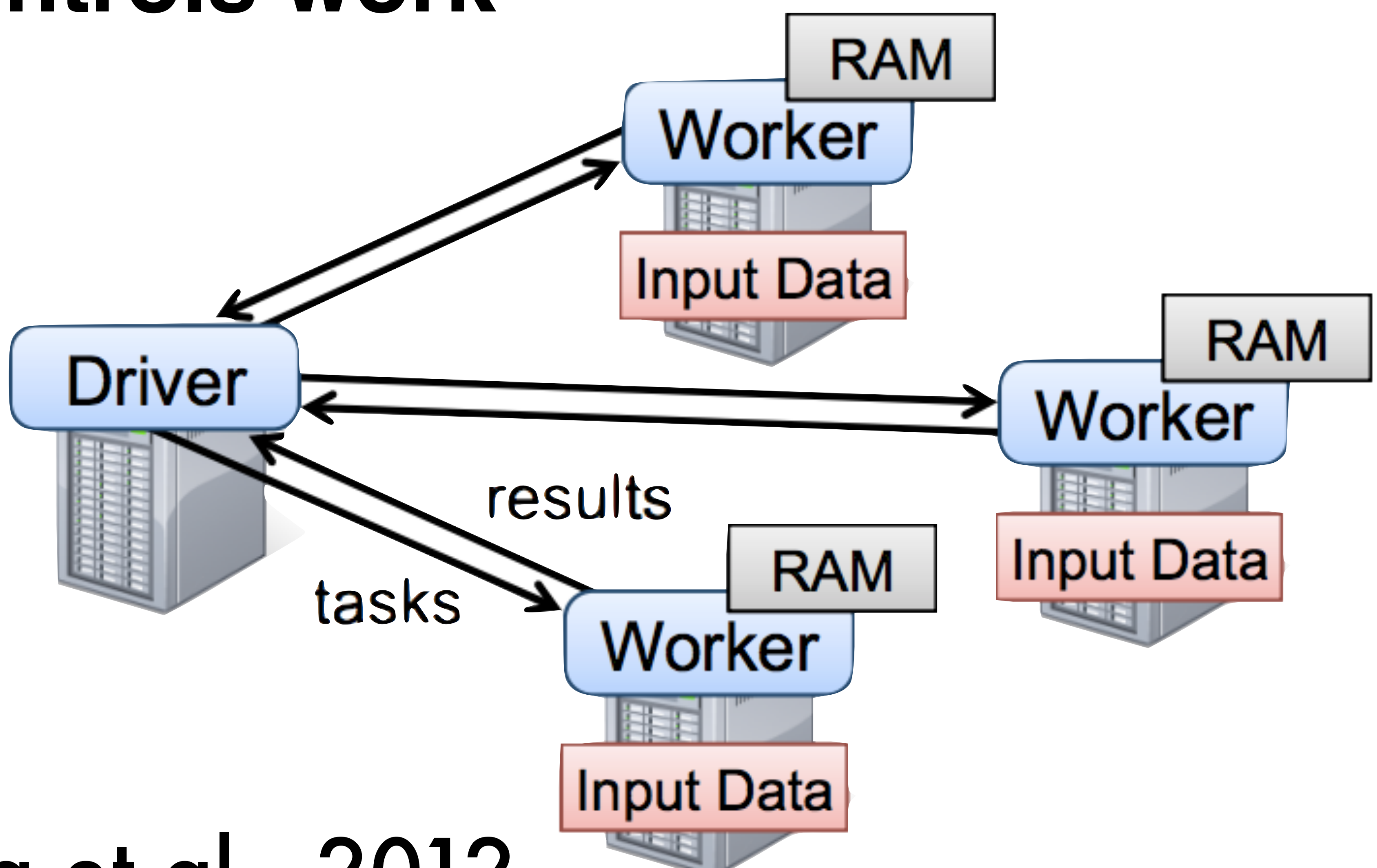
PE ID	PE Name	Key Tuple
PE1	RouterPE	null
PE2	JoinPE	serve=123
PE3	BotFilterPE	user="Peter"
PE4	CTRPE	q-ad=ipod-78

click through rate estimation


Spark
SPARK
MICHIGAN
028M179
FEB
MANUFACTURER
12

Resilient Distributed Datasets

- **Data is transformed by processing**
- **Store intermediate data using lineage**
- **Driver controls work**



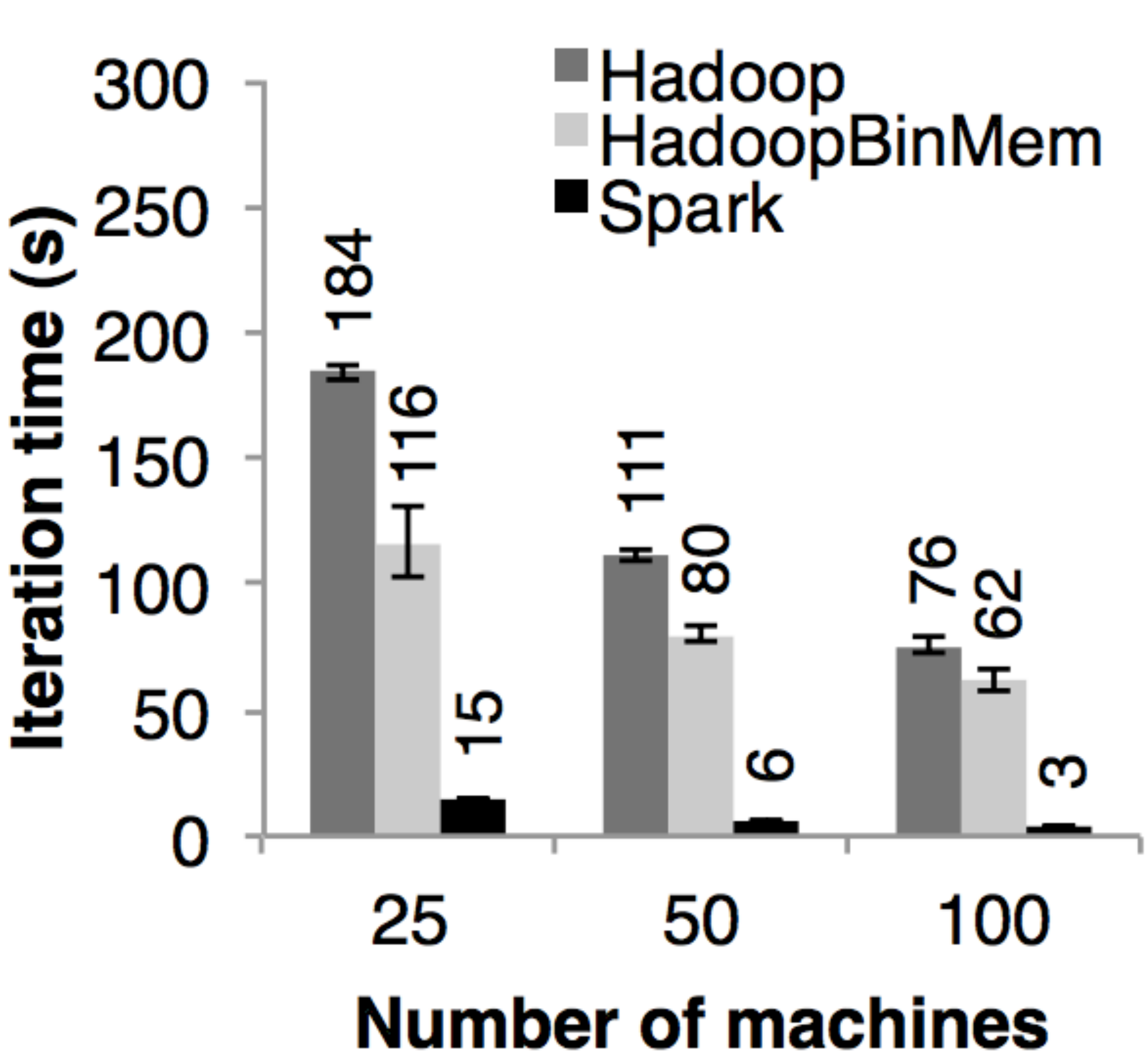
Zaharia et al., 2012

Beyond MapReduce

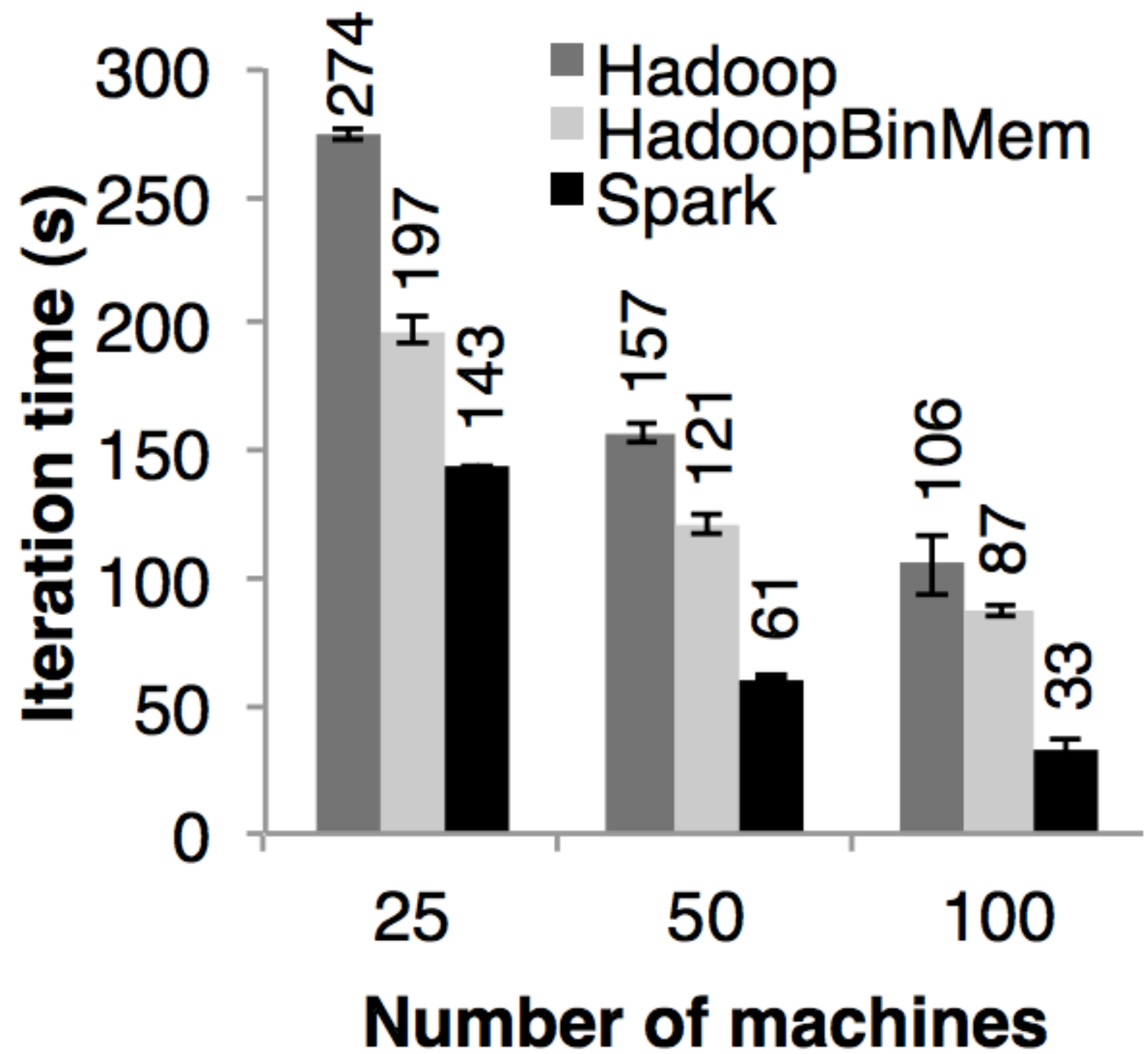
Transformations	$map(f : \mathrm{T} \Rightarrow \mathrm{U})$	: RDD[T] ⇒ RDD[U]
	$filter(f : \mathrm{T} \Rightarrow \mathrm{Bool})$	: RDD[T] ⇒ RDD[T]
	$flatMap(f : \mathrm{T} \Rightarrow \mathrm{Seq[U]})$	: RDD[T] ⇒ RDD[U]
	$sample(fraction : \mathrm{Float})$	: RDD[T] ⇒ RDD[T] (Deterministic sampling)
	$groupByKey()$	: RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	$reduceByKey(f : (\mathrm{V}, \mathrm{V}) \Rightarrow \mathrm{V})$	: RDD[(K, V)] ⇒ RDD[(K, V)]
	$union()$	: (RDD[T], RDD[T]) ⇒ RDD[T]
	$join()$	: (RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	$cogroup()$	: (RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	$crossProduct()$	: (RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	$mapValues(f : \mathrm{V} \Rightarrow \mathrm{W})$	: RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	$sort(c : \mathrm{Comparator[K]})$	: RDD[(K, V)] ⇒ RDD[(K, V)]
	$partitionBy(p : \mathrm{Partitioner[K]})$	: RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	$count()$	: RDD[T] ⇒ Long
	$collect()$	: RDD[T] ⇒ Seq[T]
	$reduce(f : (\mathrm{T}, \mathrm{T}) \Rightarrow \mathrm{T})$	: RDD[T] ⇒ T
	$lookup(k : \mathrm{K})$	: RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	$save(path : \mathrm{String})$	: Outputs RDD to a storage system, *e.g.*, HDFS

rich language & preprocessor

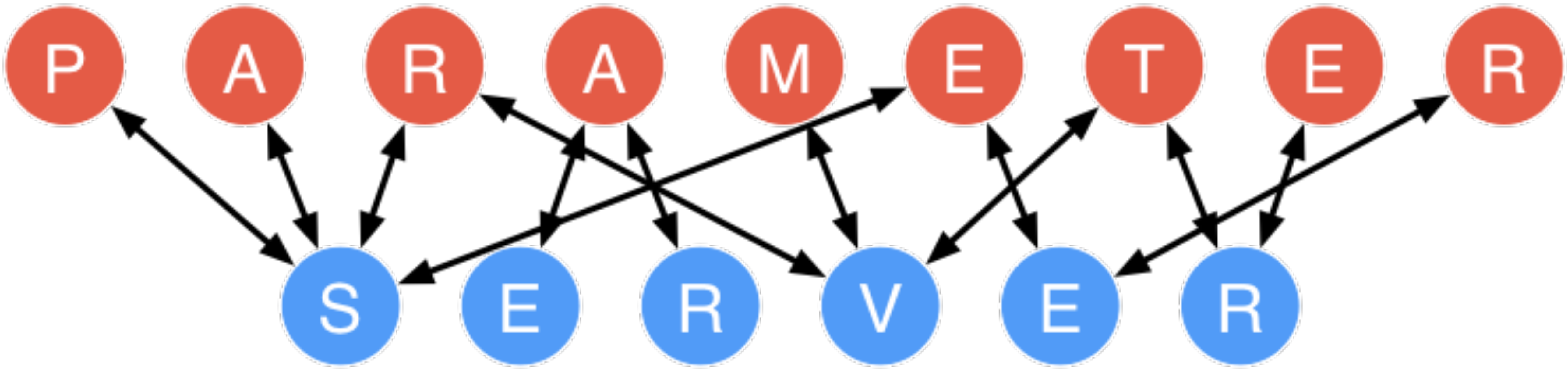
Improvement over MapReduce



logistic regression



k-means


P
A
R
A
M
E
T
E
R
S
E
R
V
E
R

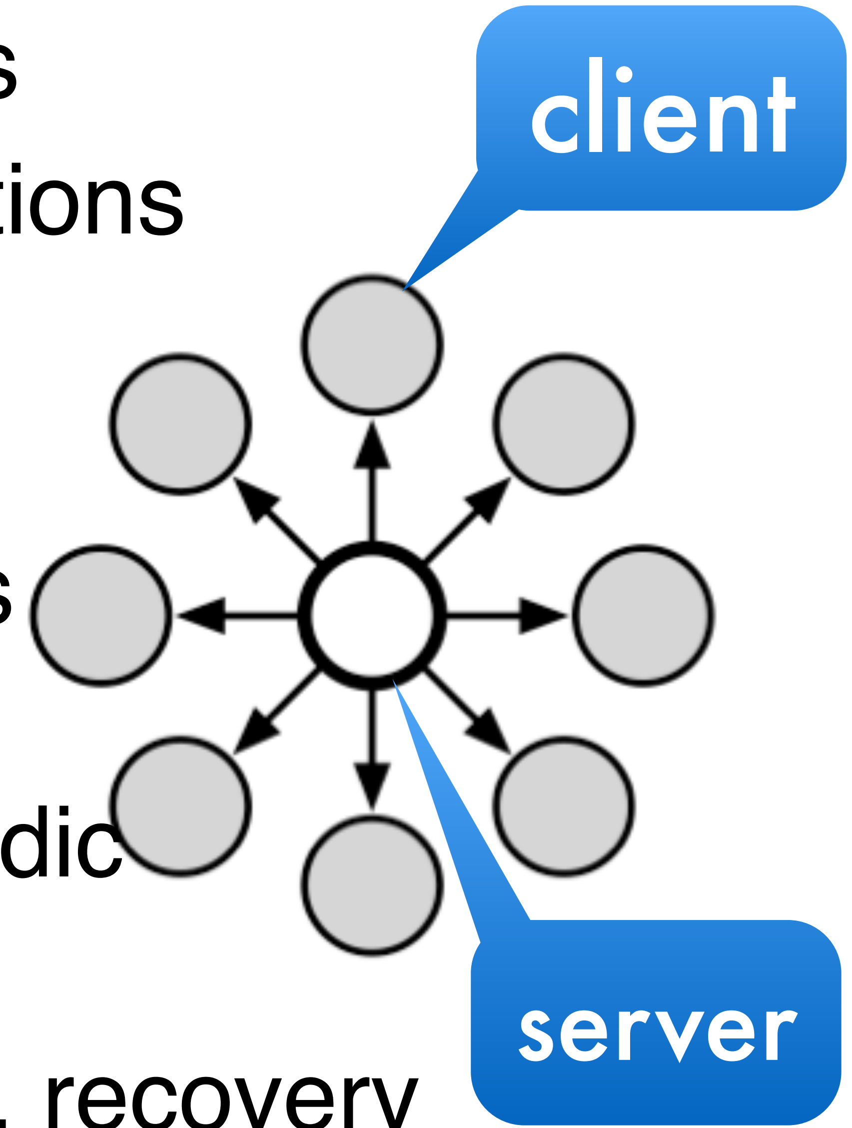
Machine Learning Problems

- Many models have O(1) blocks of O(n) terms (LDA, logistic regression, recommender systems)
- More terms than what fits into RAM (personalized CTR, large inventory, action space)
- Local model typically fits into RAM
- Data needs many disks for distribution
- Decouple data processing from aggregation

- Optimize for the 80% of all ML problems

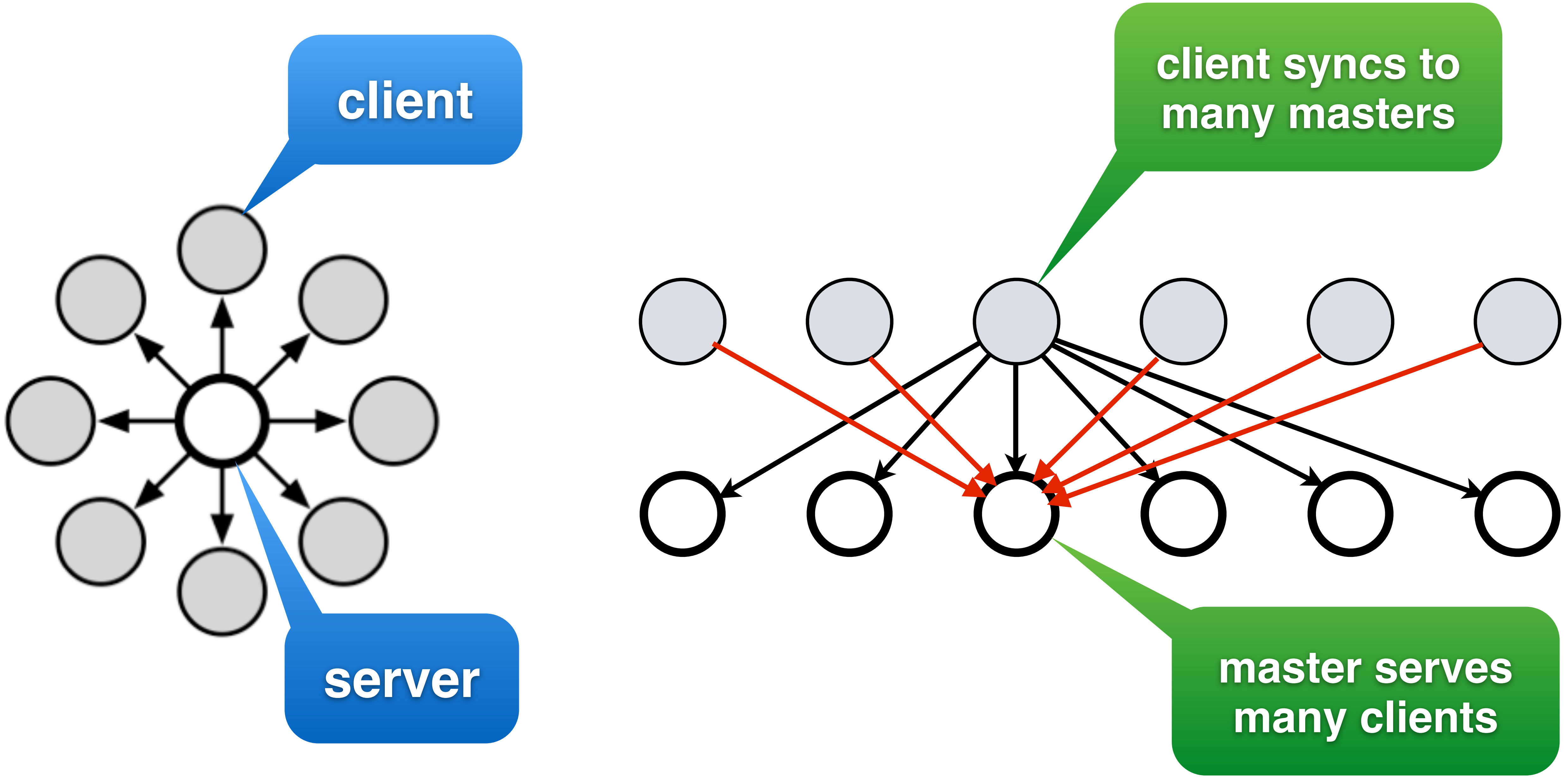
General parallel algorithm template

- Clients have local view of parameters
- P2P is infeasible since $O(n^2)$ connections
- Synchronize with parameter server
  - Reconciliation protocol
    average parameters, lock variables
  - Synchronization schedule
    asynchronous, synchronous, episodic
  - Load distribution algorithm
    uniform distribution, fault tolerance, recovery



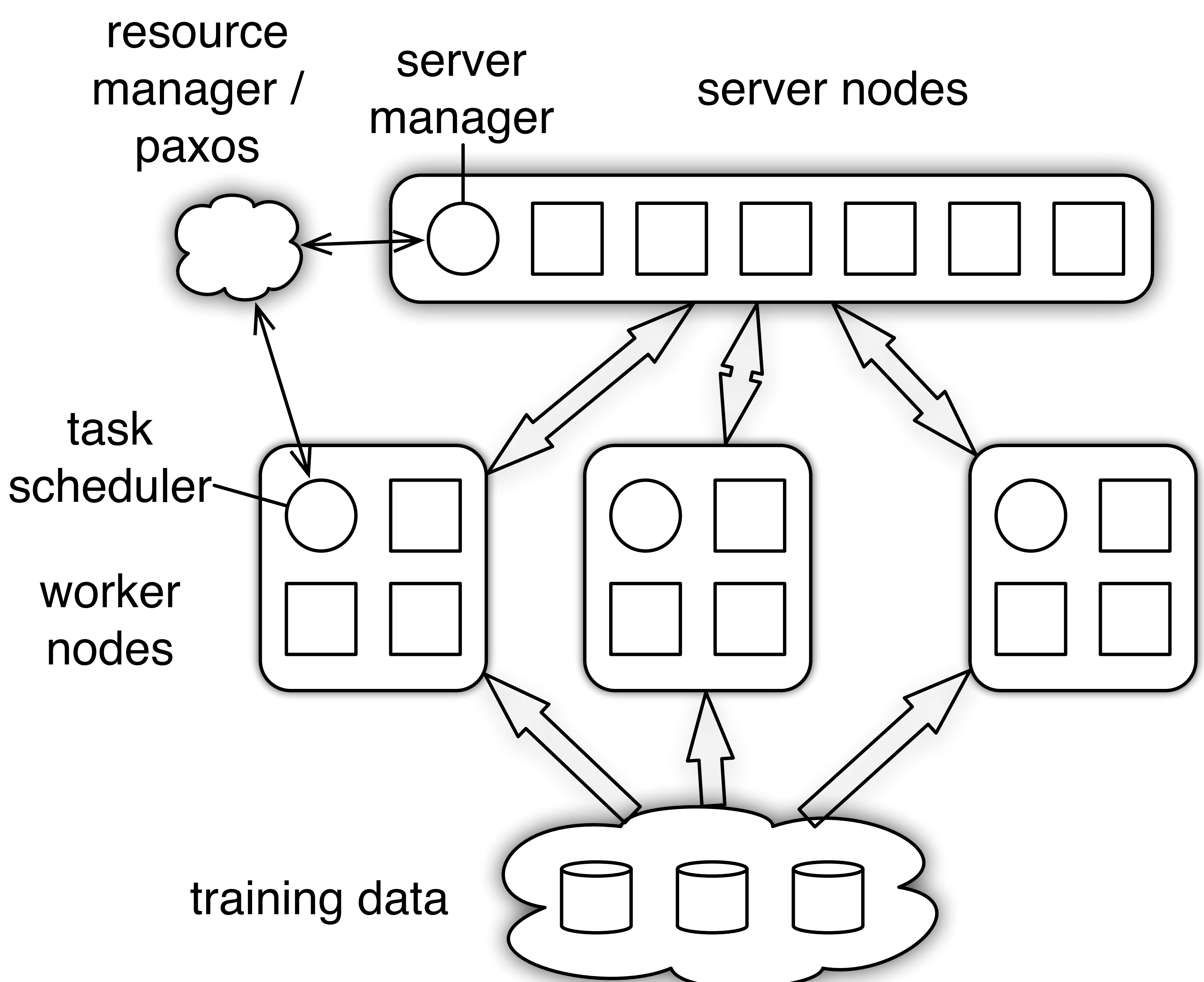
Smola & Narayanamurthy, 2010, VLDB
Gonzalez et al., 2012, WSDM
Shervashidze et al., 2013, WWW

Communication pattern

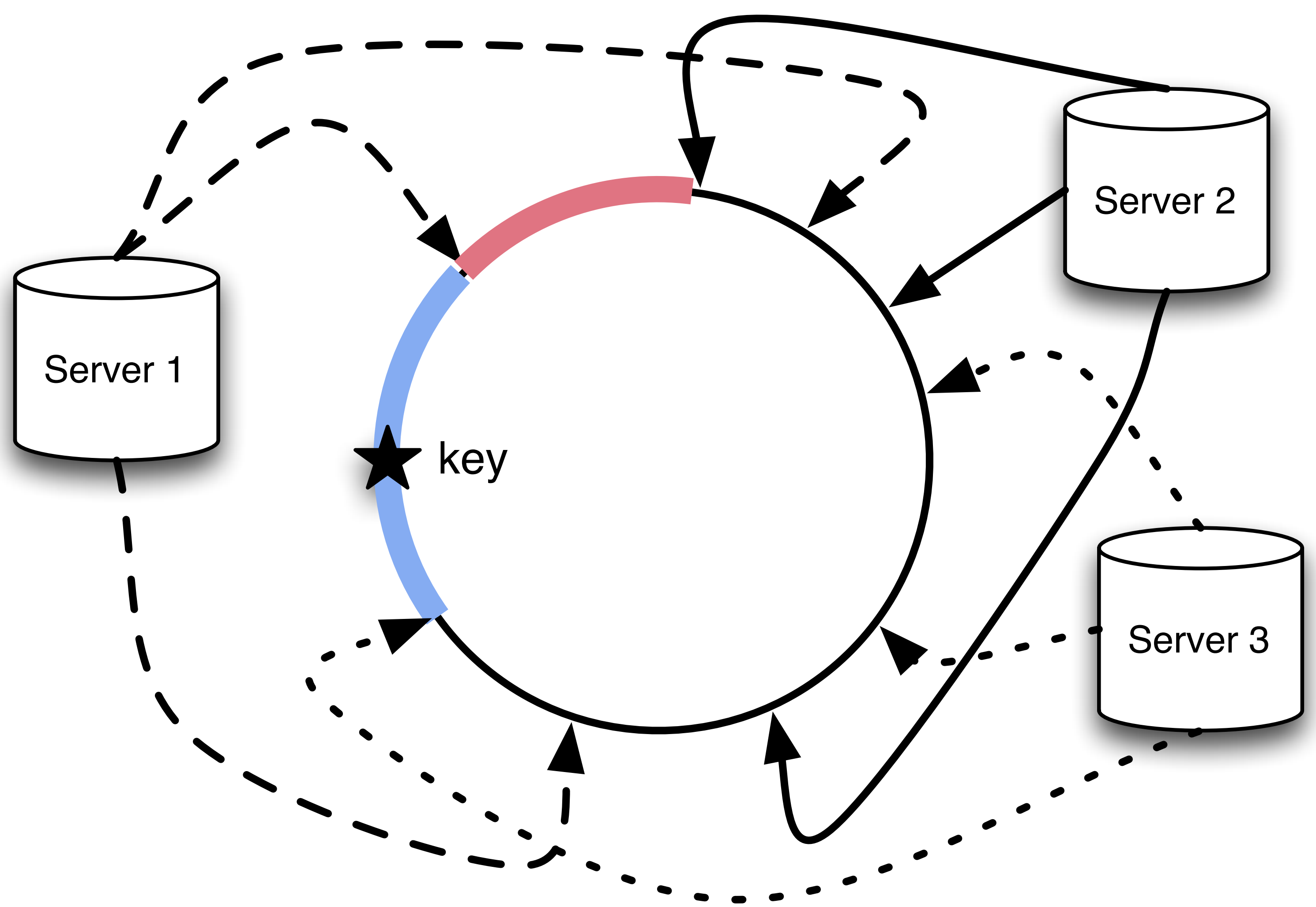


put(keys,values,clock), get(keys,values,clock)

Architecture

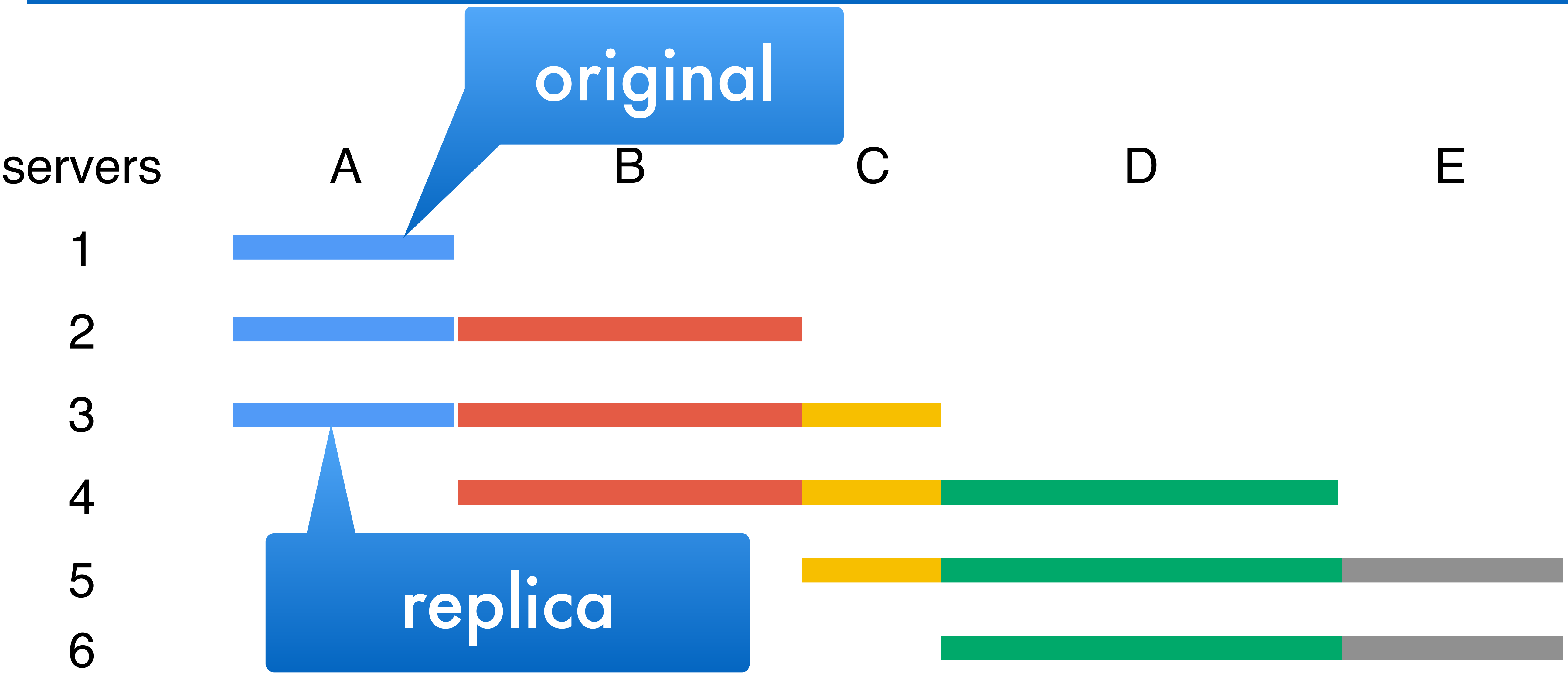


Keys arranged in a DHT

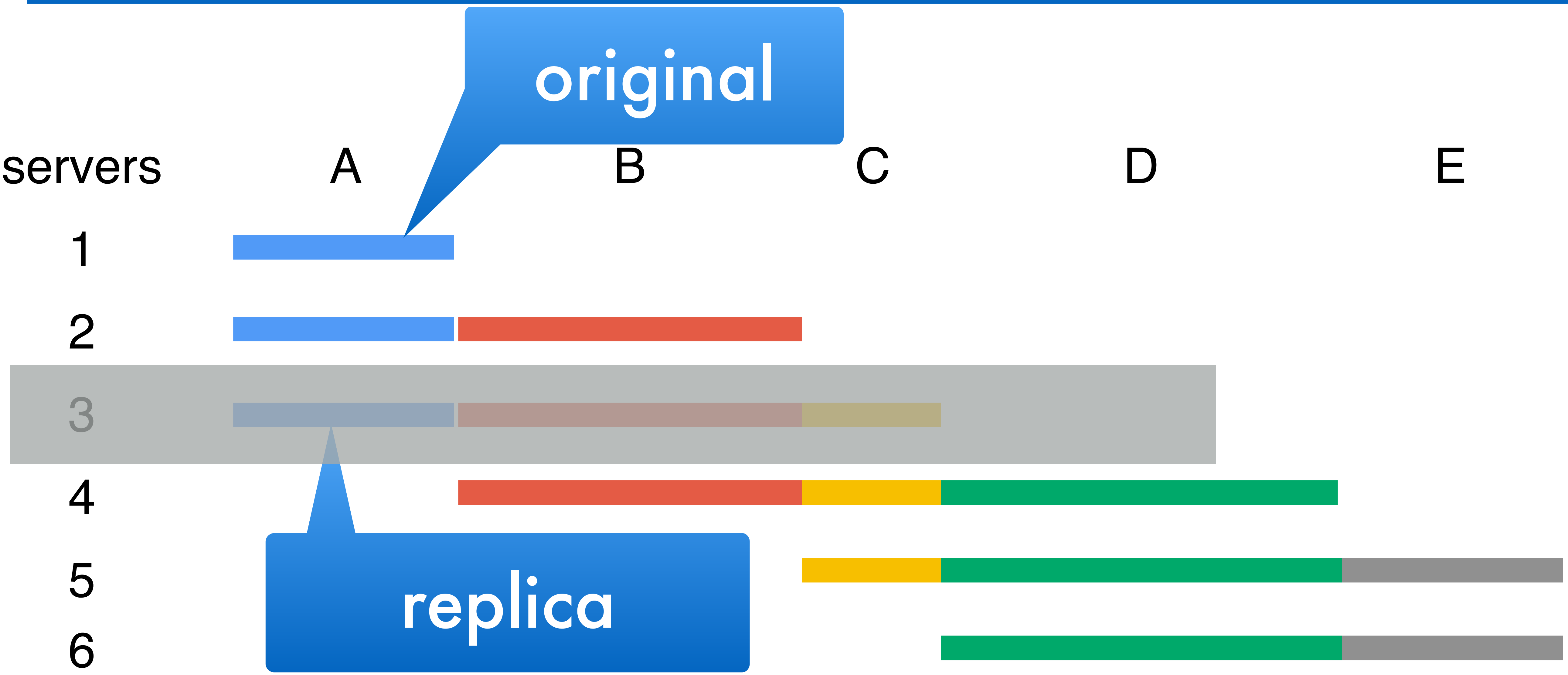


- Virtual servers
  - loadbalancing
  - multithreading
- DHT
  - contiguous key range for clients
  - easy bulk sync
  - easy insertion of servers
- Replication
  - Machines hold replicas
  - Easy fallback
  - Easy insertion / repair

Key layout

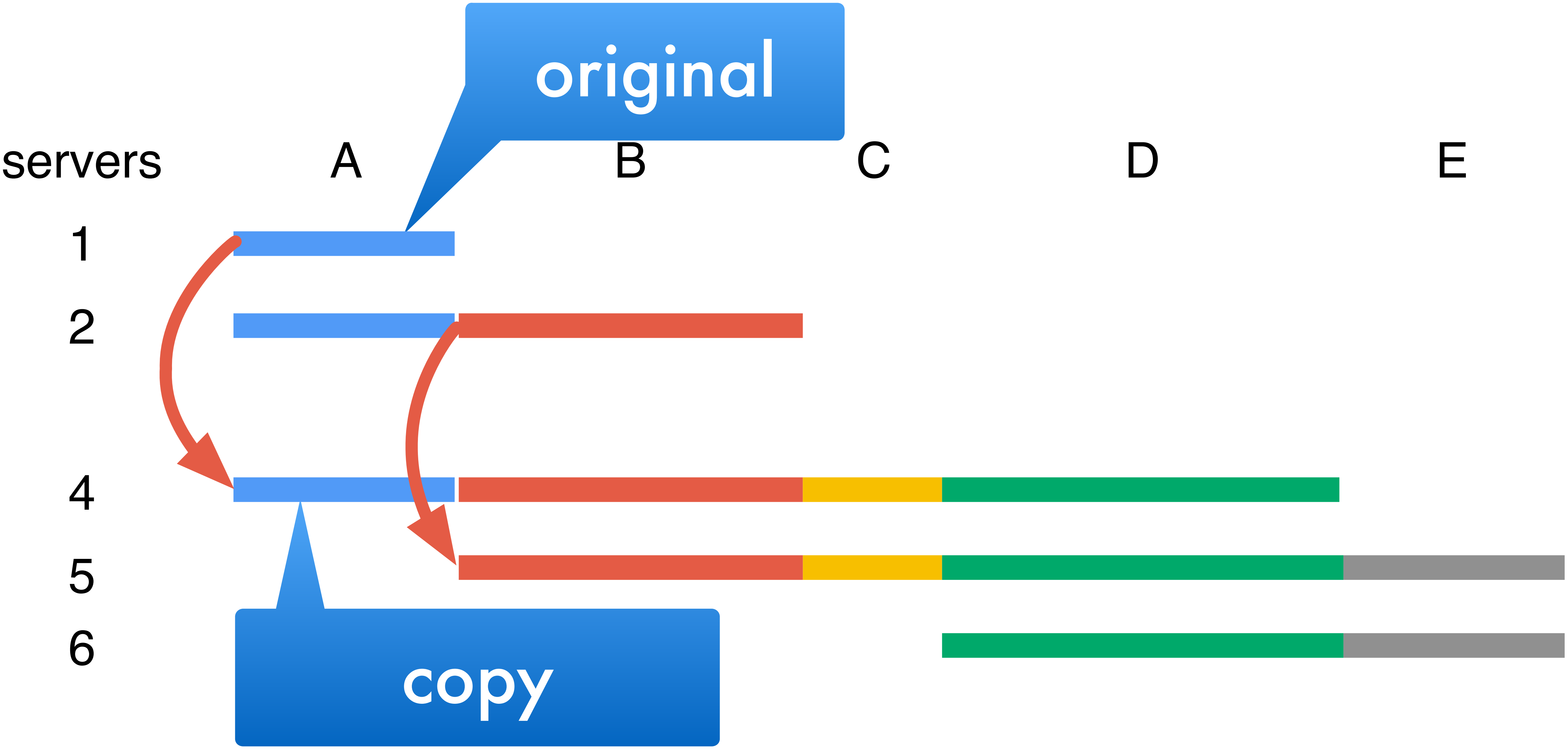


Key layout

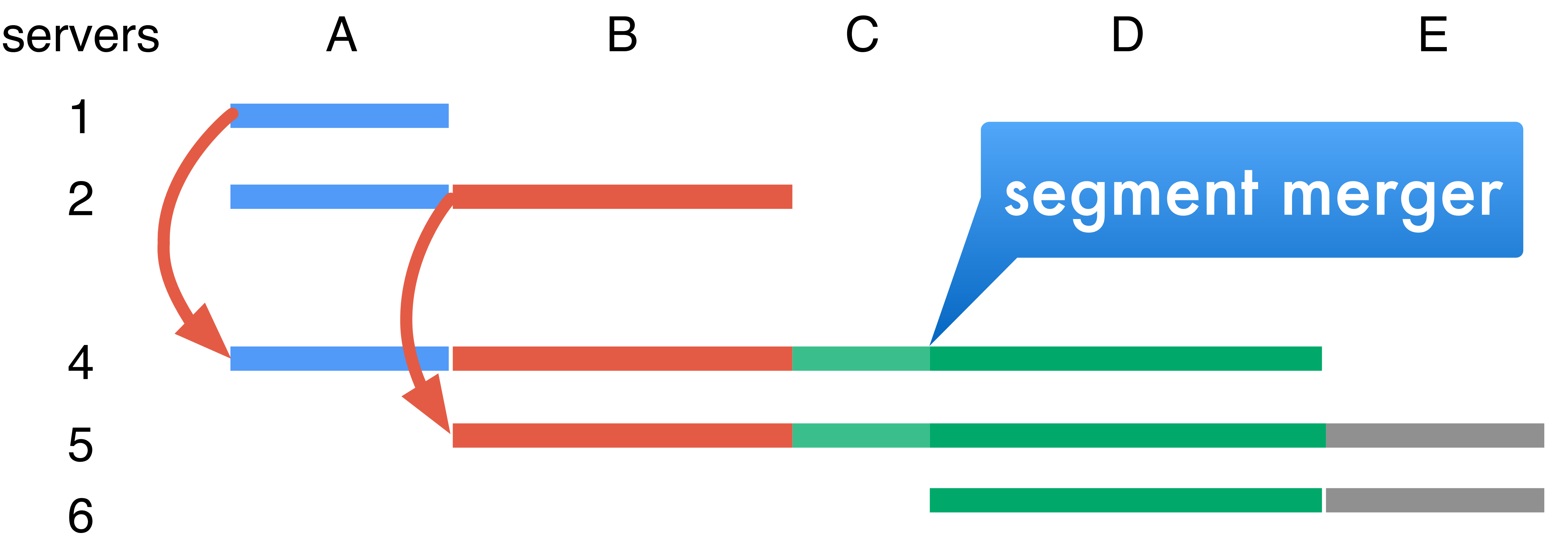


Carnegie Mellon University

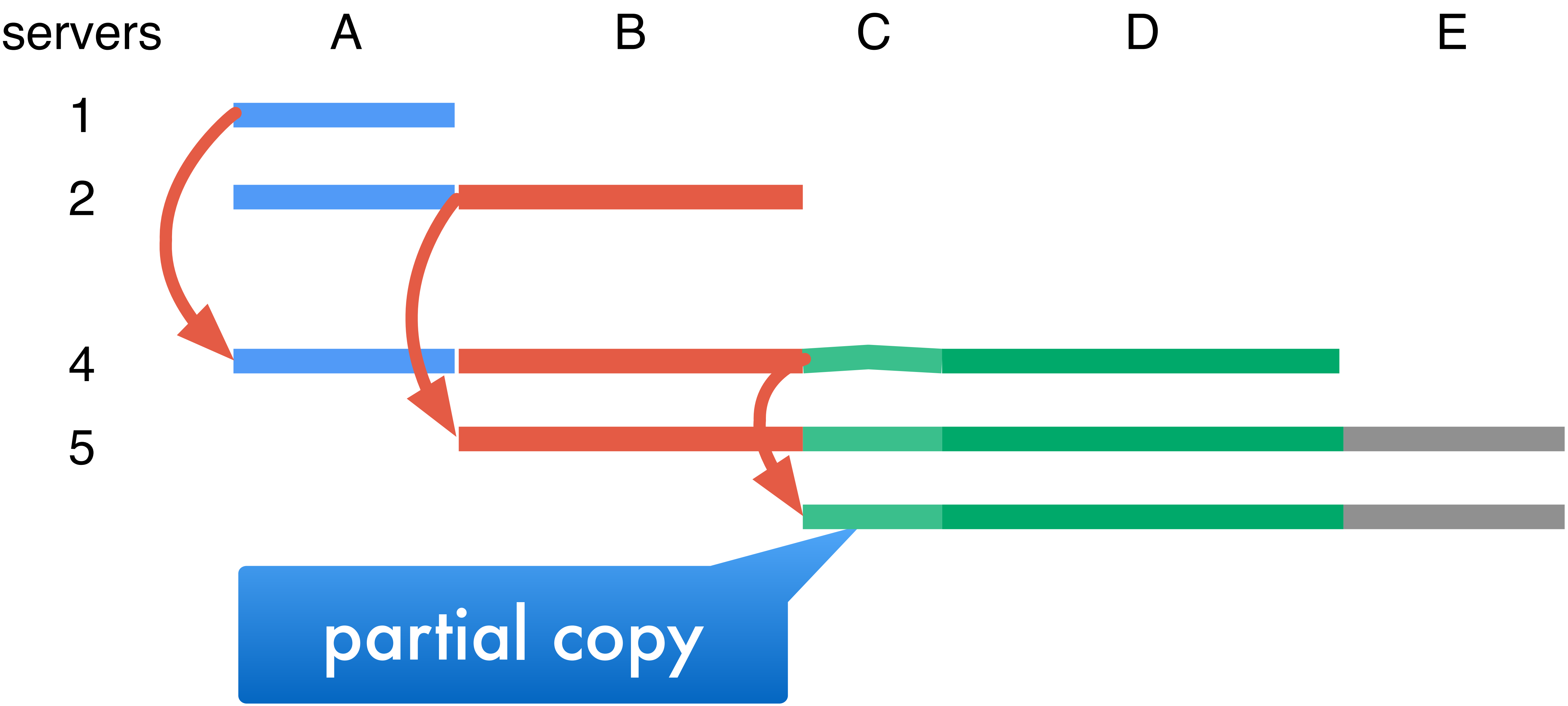
Key layout



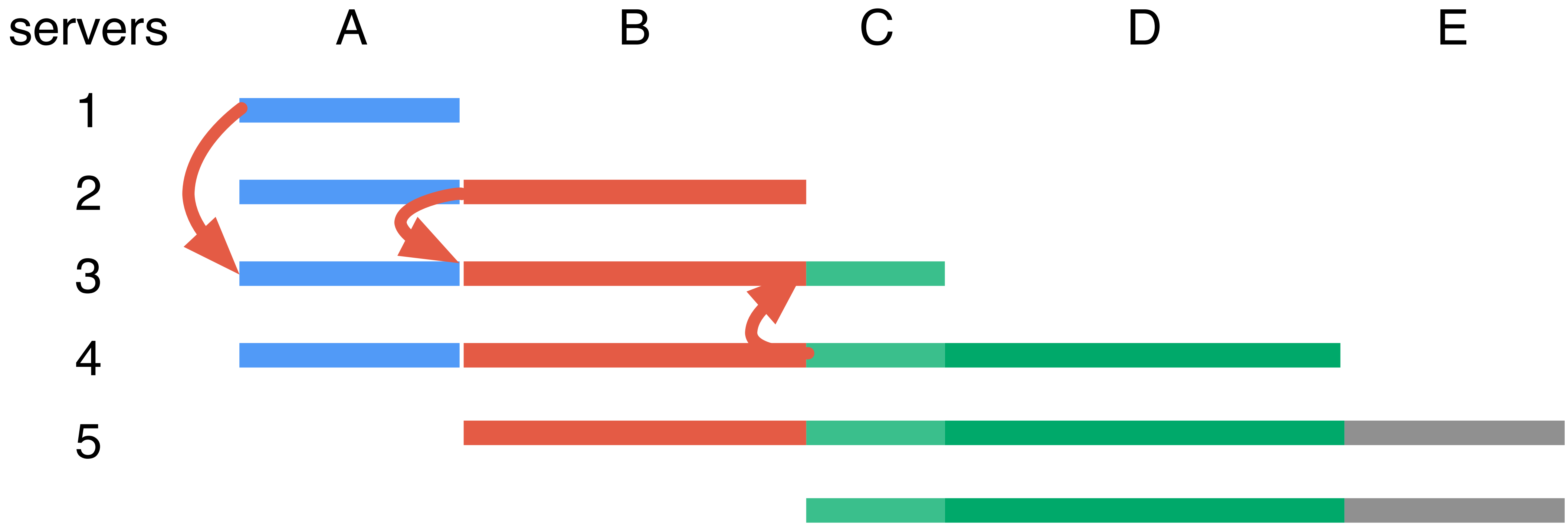
Key layout



Key layout

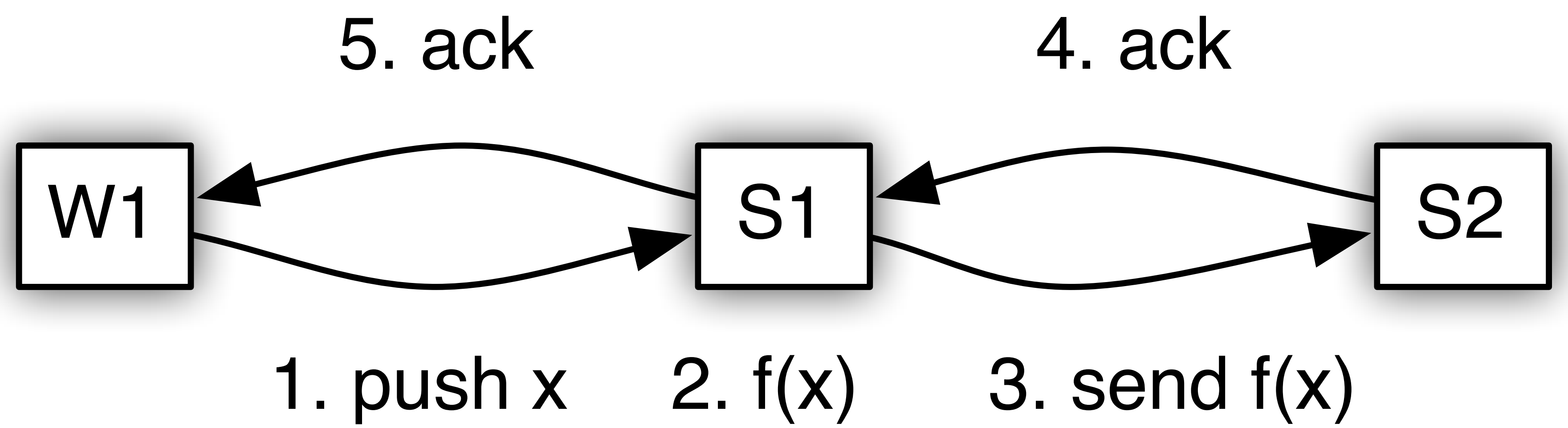


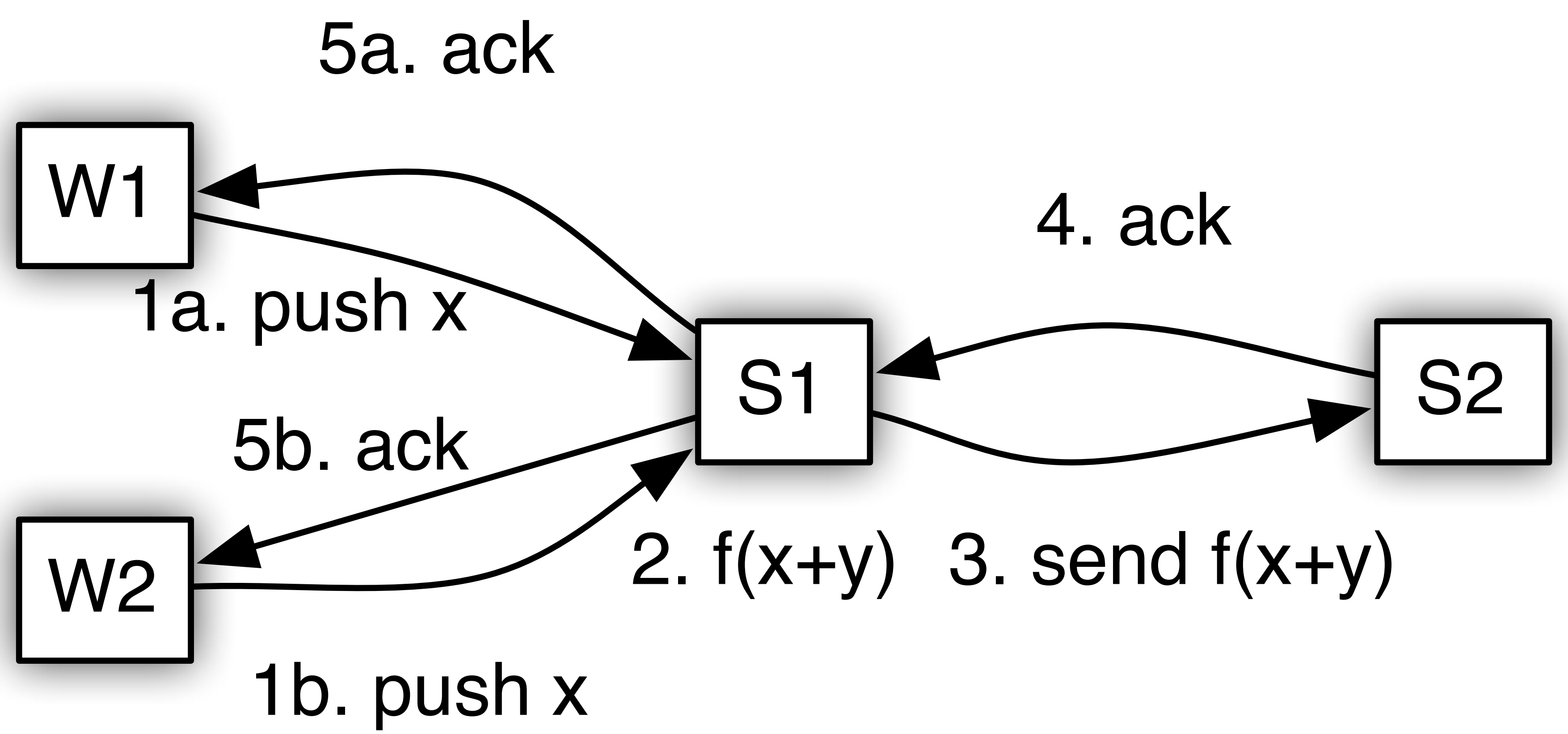
Recovery / server insertion



- Precopy server content to new candidate (3)
- After precopy ended, send log
- For k virtual servers this causes $O(k^{-2})$ delay
- Consistency using vector clocks

Message Aggregation on Server



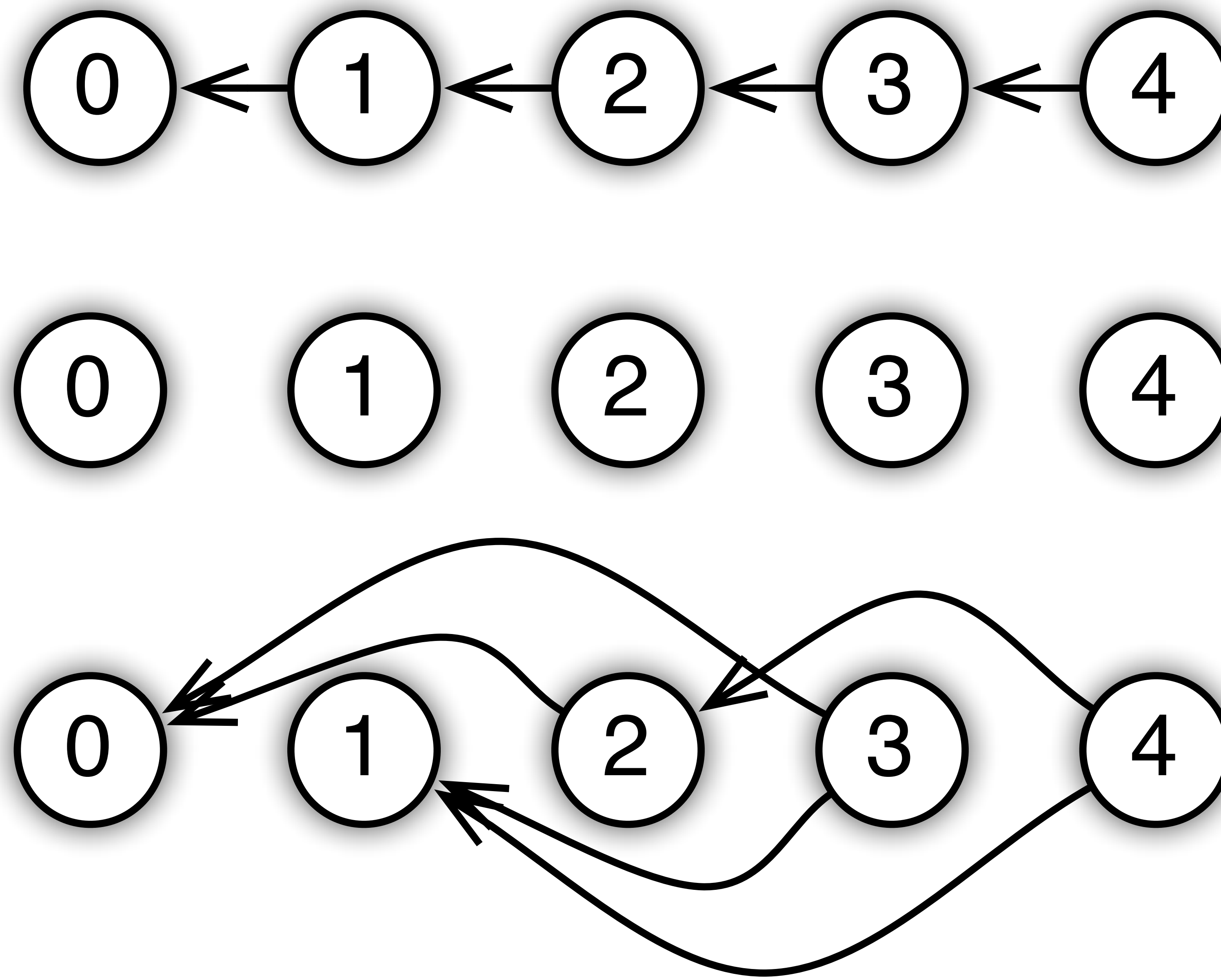


Consistency models

(a) Sequential

(b) Eventual

(c) Bounded delay



via task processing engine on client/controller


Models

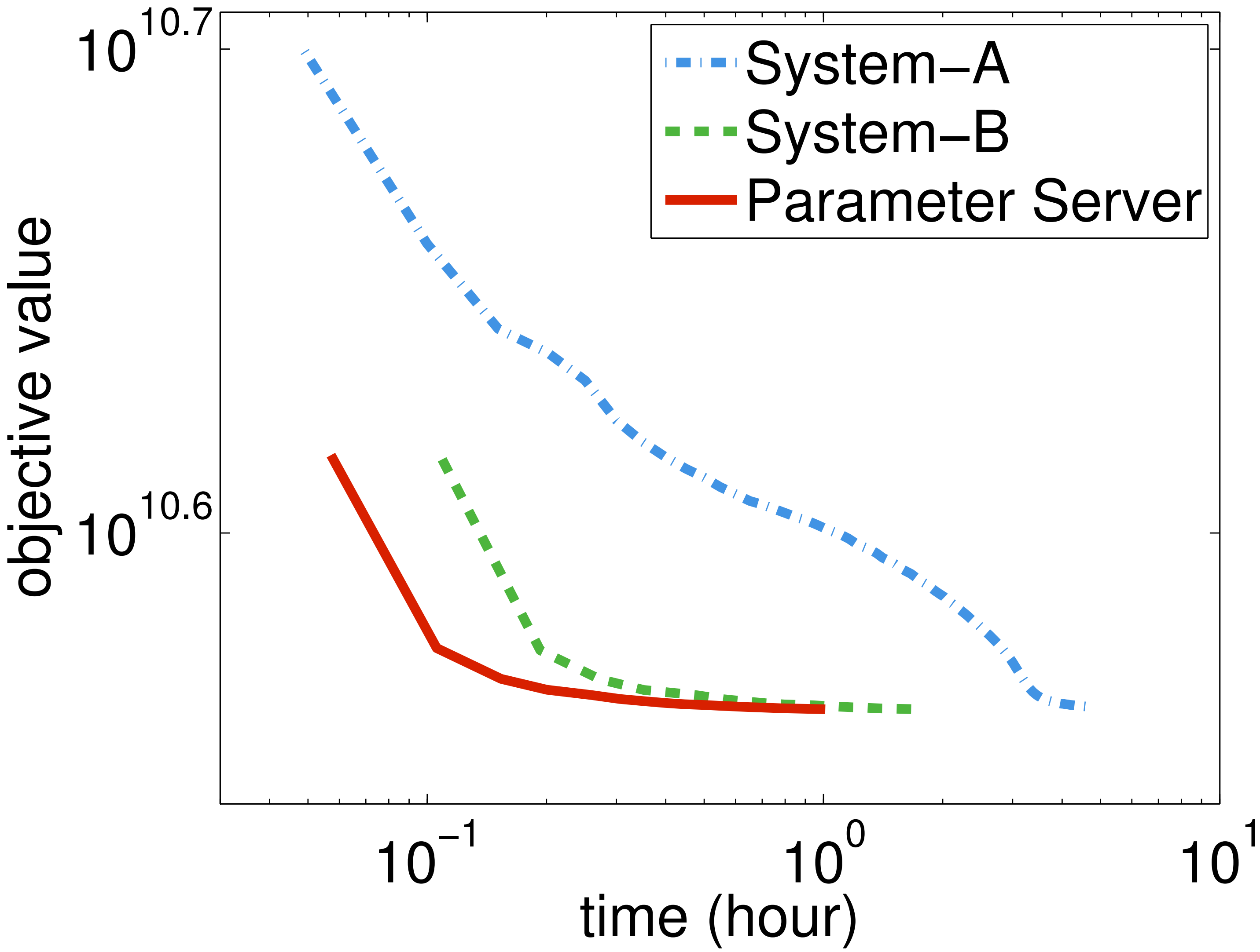
Guinea pig - logistic regression

$$\min_{w \in \mathbb{R}^p} \sum_{i=1}^{n} \log(1 + \exp(-y_i \langle x_i, w \rangle)) + \lambda \|w\|_1$$

- Implementation on Parameter Server

	Method	Consistency	LOC
System-A System-B	L-BFGS Block PG	Sequential Sequential	10,000 30,000
Parameter Server	Block PG	Bounded Delay KKT Filter	300

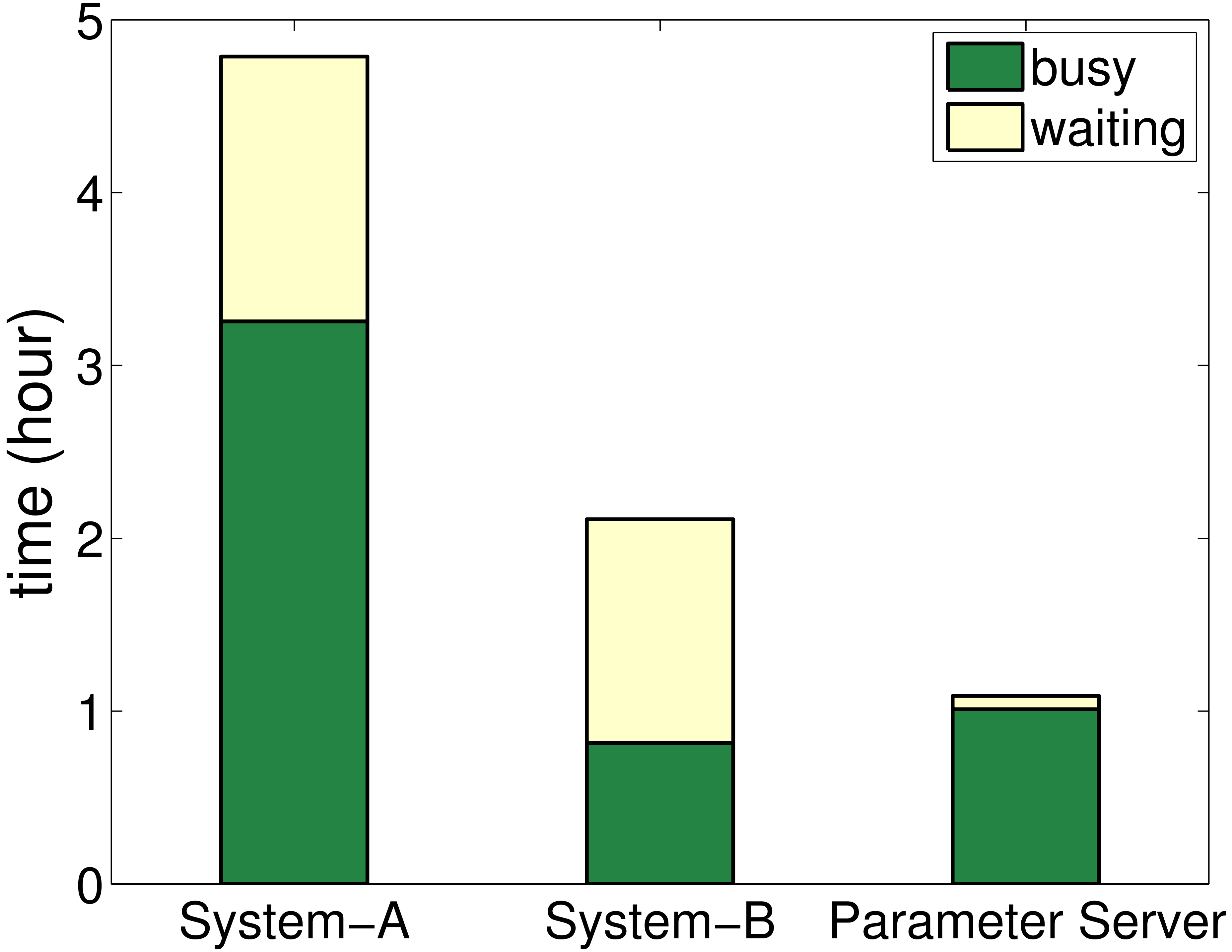
Convergence speed



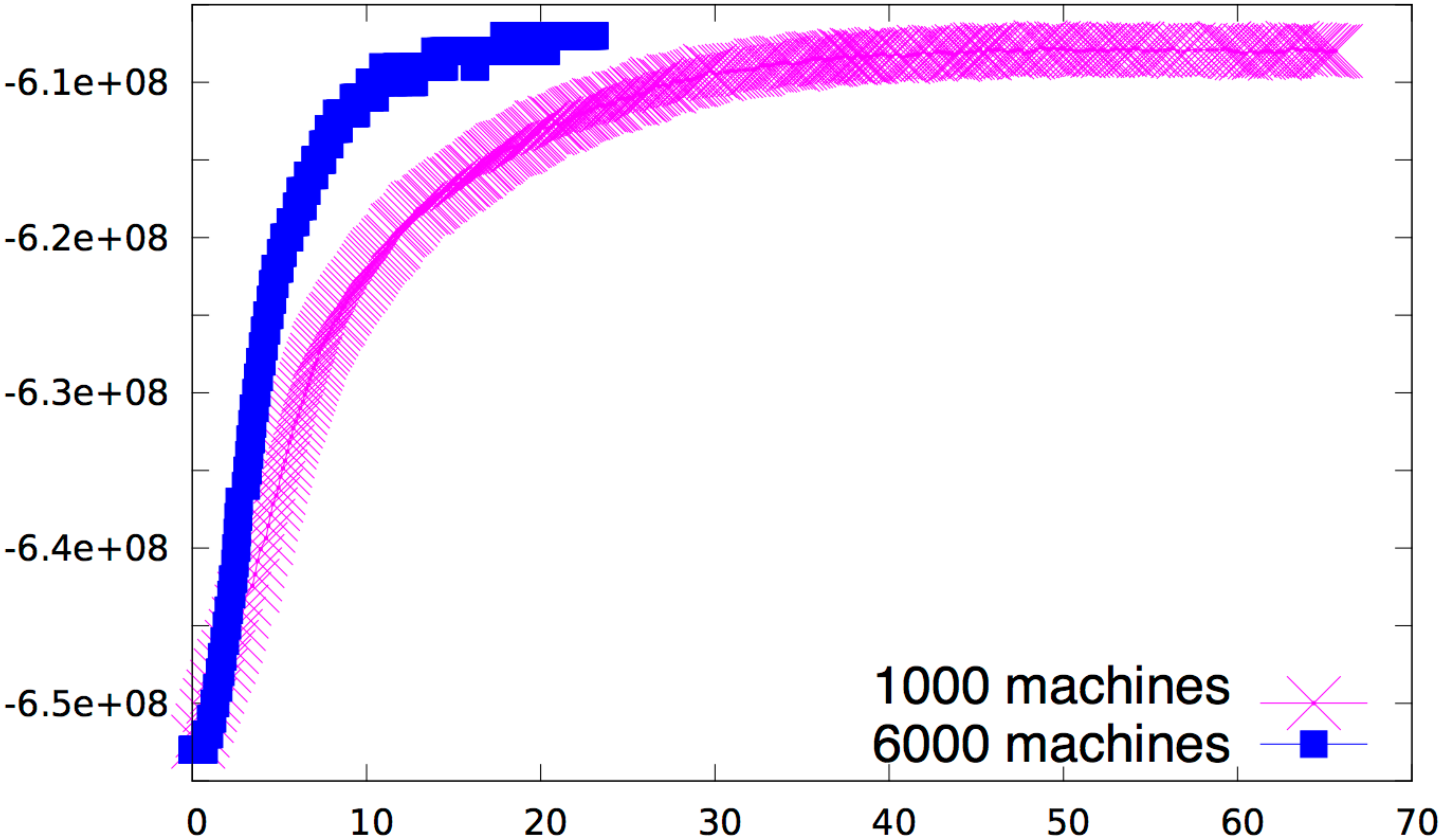
500TB CTR data
100B variables
1000 machines

- System A and B are production systems at a very large internet company …

Scheduling Efficiency



Topic models ...



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