# CLOUD
## scale storage

Anwitaman DATTA
SCE, NTU Singapore

# NIST definition: Cloud Computing

Cloud computing is a model for enabling *ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources* (e.g., networks, servers, storage, applications, and services) that can be *rapidly provisioned and released with minimal management* effort or service provider interaction

# Cloud: Inside Out

⌘ Outside view
  - ubiquitous, convenient, **on-demand**

⌘ A single/exclusive entity
  - Access through a "demilitarized zone"
  - API based
  - Agnostic to multi-tenancy

⌘ Elastic/infinite resources
  - Pay as you use

⌘ Often web based
  - Any time any where any device

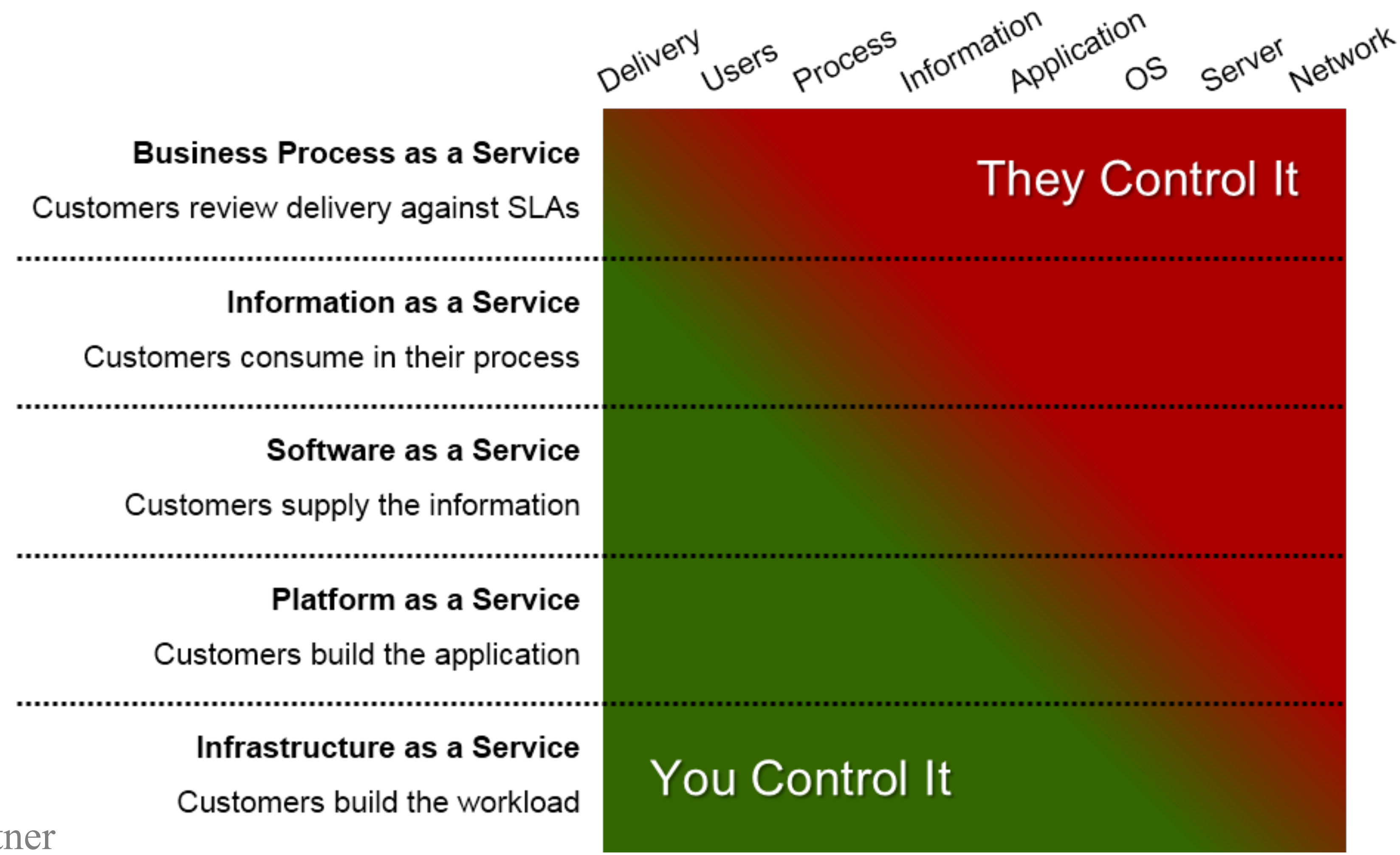
⌘ Inside view
  - **shared pool** of configurable computing resources

⌘ Rapid provisioning, minimal management
  - New compute units joining, old ones retiring
  - Adaptive: loads, faults, …

⌘ Scaled-out infrastructure
  - e.g., GFS, Dynamo, Azure …

⌘ Multi-tenancy
  - Virtualization, migration, …

# Cloud: In many flavors




Source: Gartner

# A new stack

Applications

SQL Implementations
e.g., PIG (relational algebra), HIVE, …

NoSQL
e.g., Map-Reduce, Hadoop, DynamoDB, BigTable, Hbase, Cassandra …

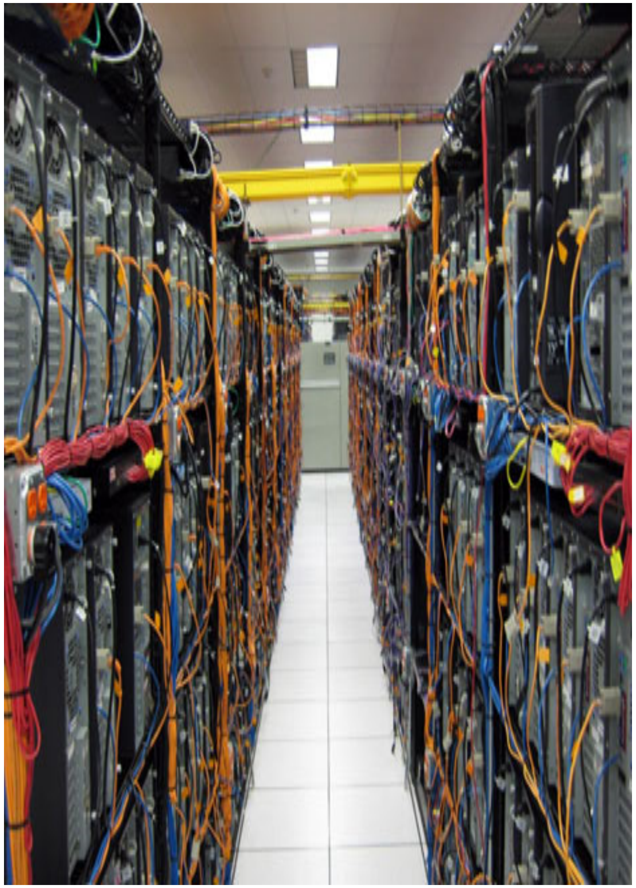
Distributed file systems

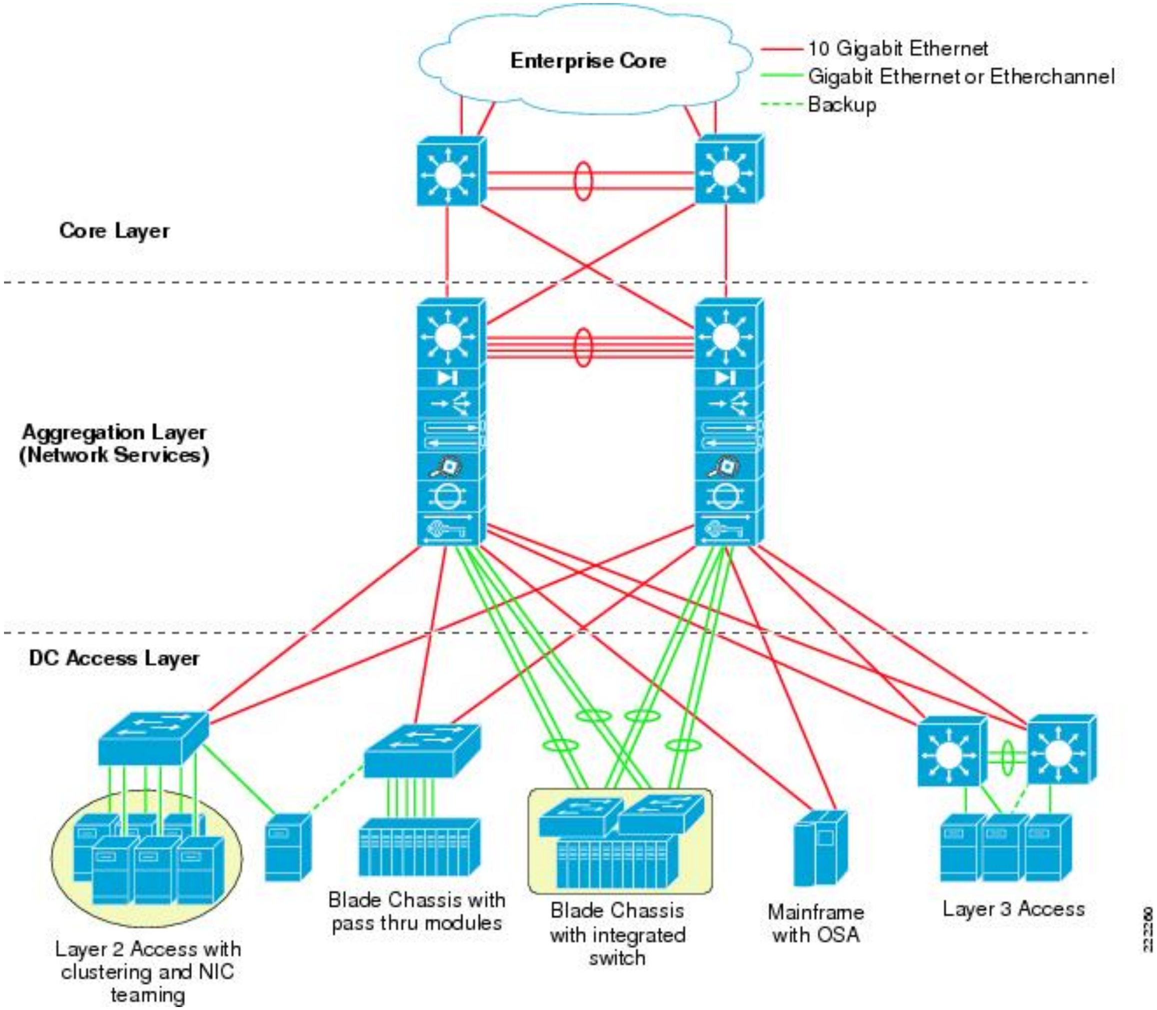
Reliable storage service

Distributed Physical Infrastructure
Storage & Compute Nodes, Interconnect, …

The boundaries are often blurry

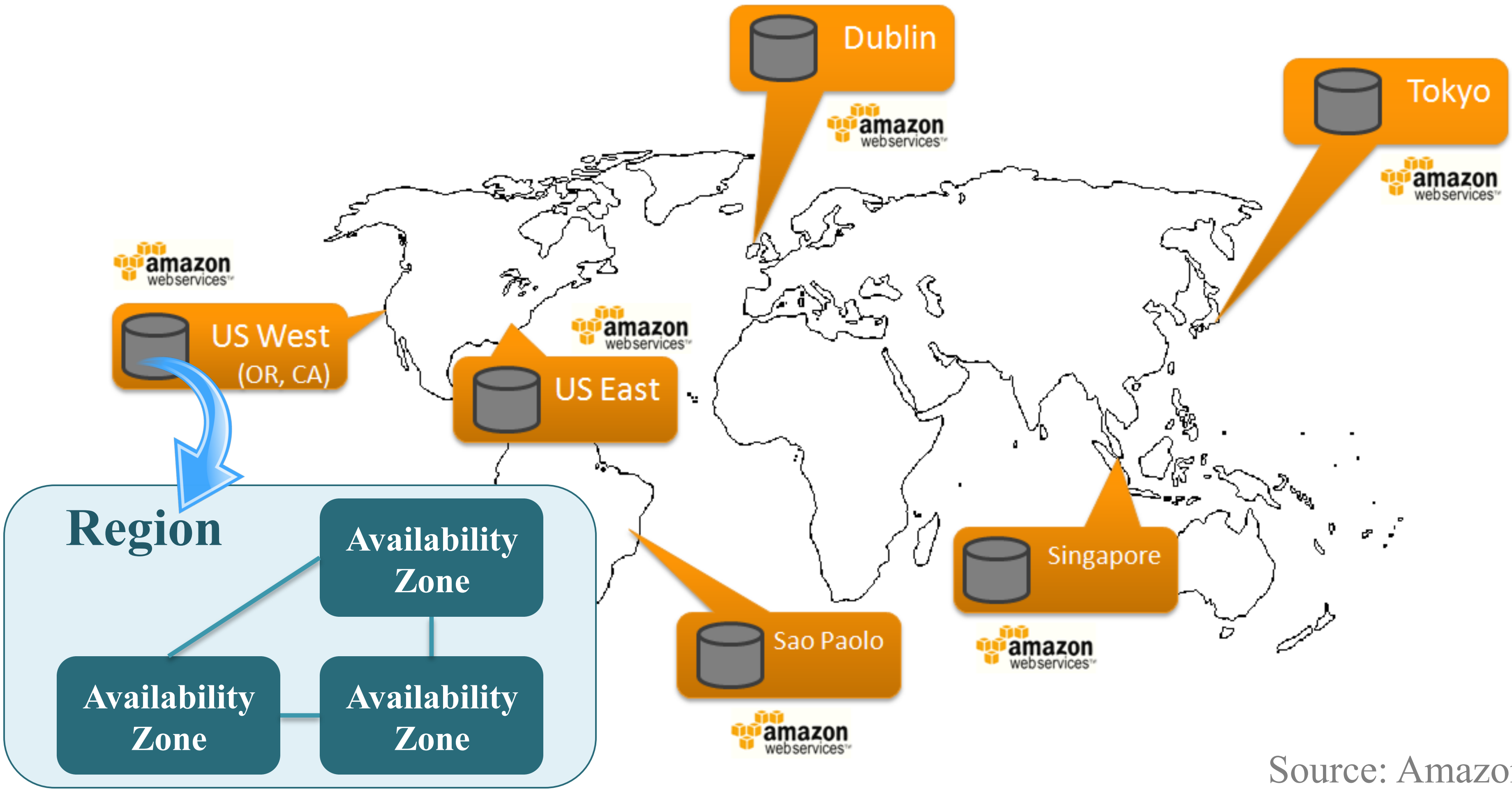
# Distributed physical infrastructure






Source: Cisco

# Many levels of fault-tolerance




Source: Amazon

# Huge physical infrastructure

- Requires matching software solutions
  - **data storage and management**
  (among many other aspects)

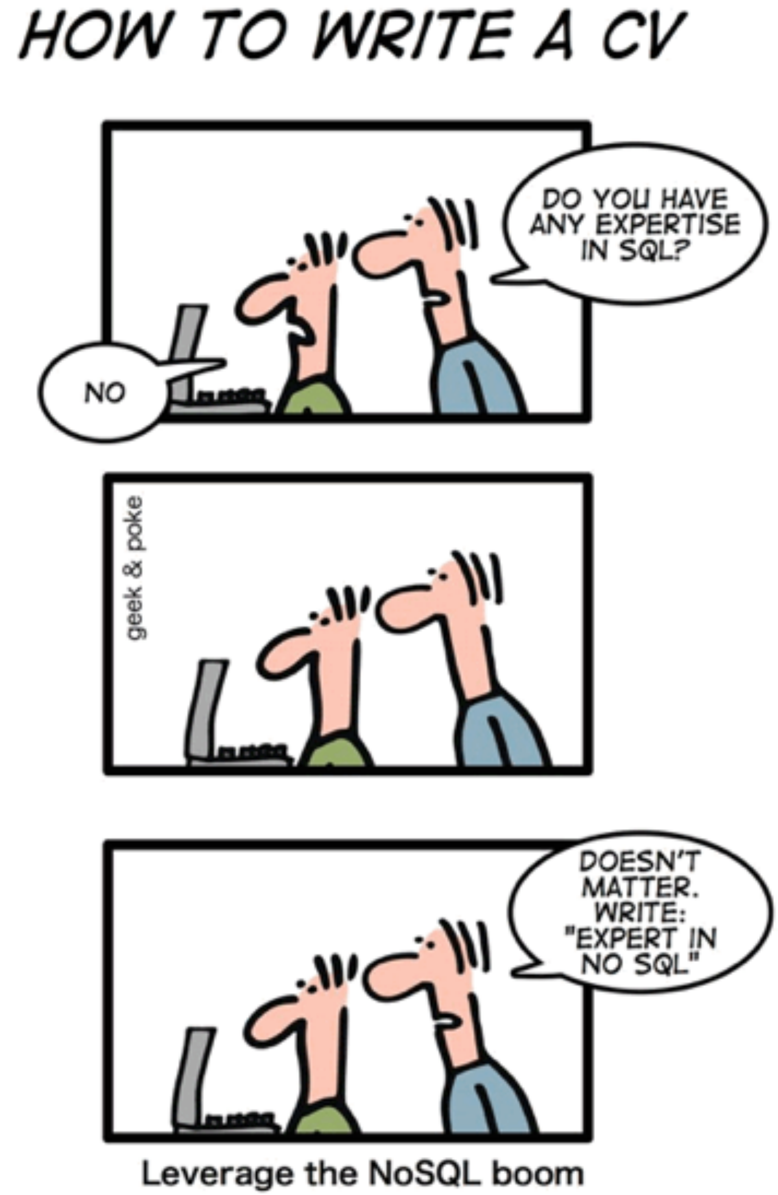
# One size does not fit all

- ⌘ Workload based designs
  The why decides
  the how and the what!

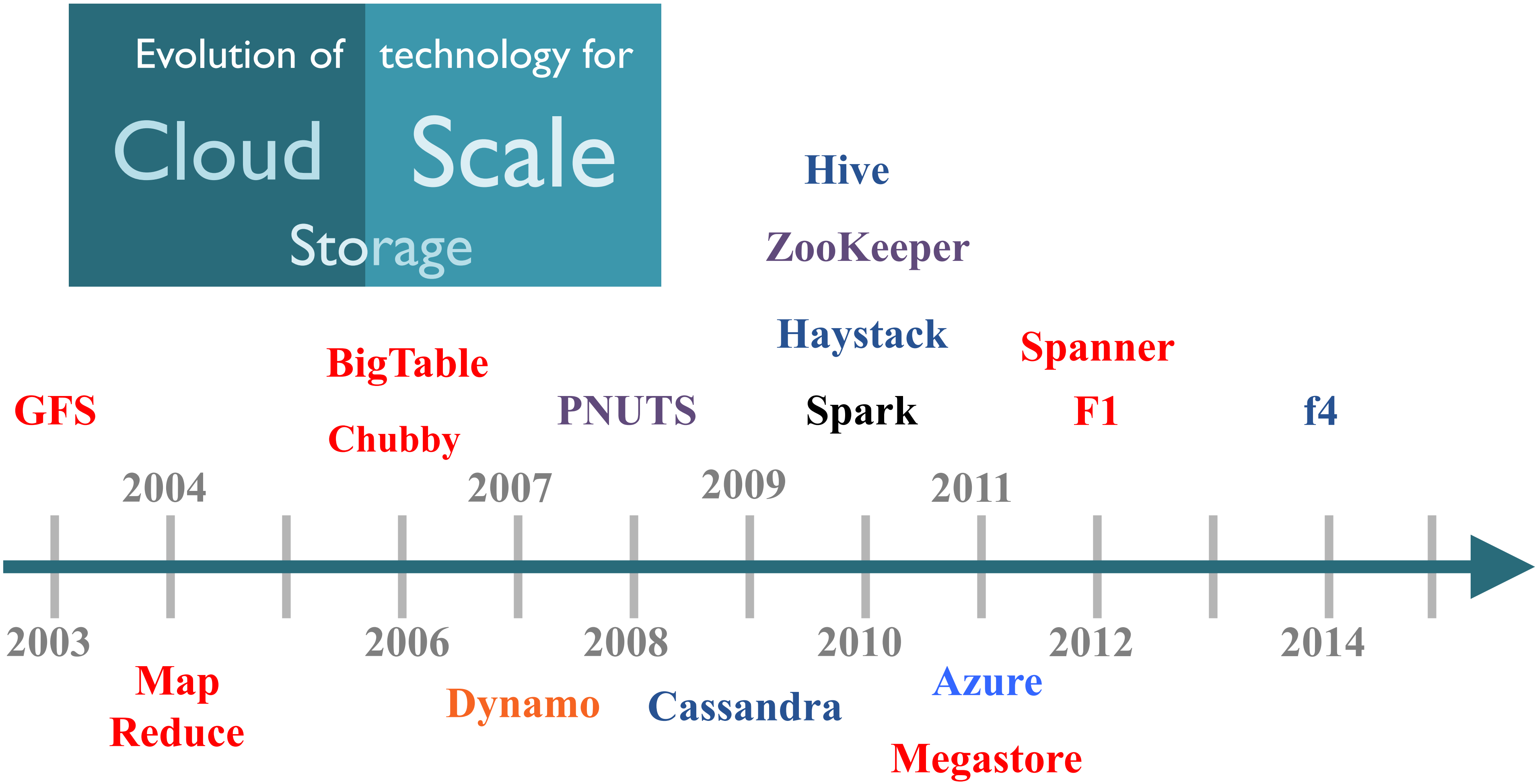
# Not only SQL

⌘ RDBMS may not be suitable
- overkill for a purpose
- scale(out) issues

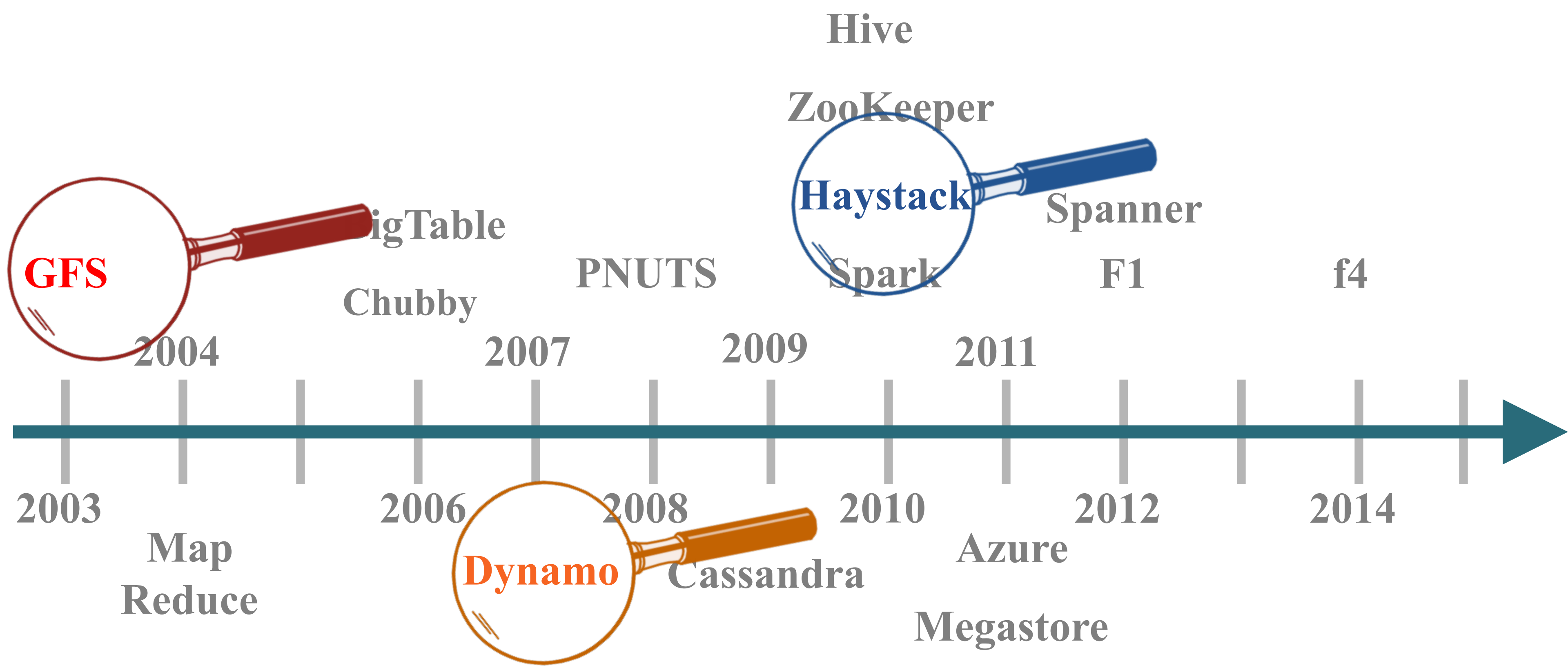
⌘ Application & workload specific custom solutions for storage and data management
- object stores, tuple stores, key-value stores, graph data bases, …

# Some influential papers

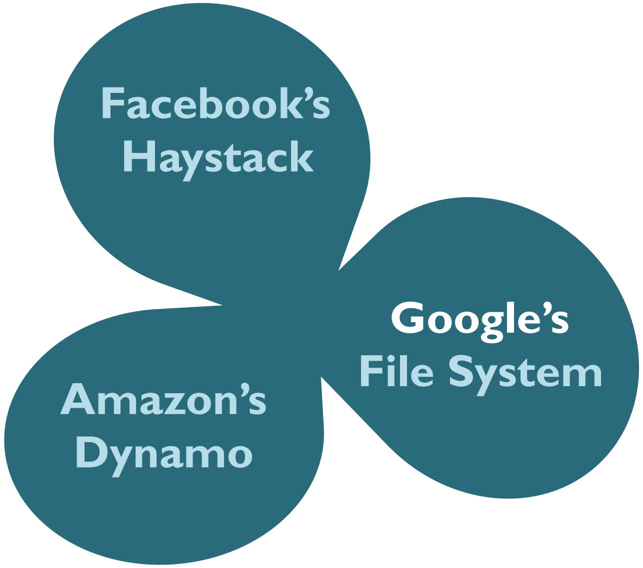


# A closer look



# GFS

## GOOGLE FILE SYSTEM



*Reference*

**The Google File System**

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung

**SOSP 2003**

# Workload characteristics

- Bulk data processing in batches
  - **Throughput** (high *sustained bandwidth*) is more important than *latency*
- Few million files
  - Mostly **> 100 MB**
  - Multi-GB files very common
- Support for small files needed
  - But no optimization required

**GFS** was designed (circa 2002-3) primarily for processing the crawled web pages, when Google *(almost) exclusively* worked on web search.

Over time, Google's workload characteristics have changed drastically with the evolution of their business and product offerings, which in turn has led to many further innovations and changes in the underlying infrastructure, e.g. Megastore, F1, Spanner, BigTable, Colossus, etc.

# Read/Write characteristics

⌘ **Two kinds of reads**
  Large streaming reads
  - 100s of KBs, > 1MB
  Small random reads

⌘ Successive operations from a client **read contiguous region** of a file

⌘ Applications can **sort small reads** to advance steadily through a file
  Avoid going back & forth

⌘ **Writes**
  Mainly when a file is being created
  - Once written, seldom modified
  Small writes at arbitrary position
  - needs to be supported
  - do **NOT** have to be efficient

⌘ **Multiple clients**
  Append concurrently
  - Producer-consumer queues
  - many-way merging
    e.g., map-reduce operations
  - needs atomicity

# KISS: Keep it simple, stupid

- Not standard (POSIX) compliant
  Some basic operations adequate
  - create, delete, open, close, read, write files
  - snapshot, record append
- Single master (centralized)
  Simplifies system design
  - Can carry out sophisticated data placement and replication decisions using global knowledge
  - But what about fault-tolerance, bottleneck?
- Multiple chunk servers and multiple clients
  Could be running on the same machine

**HDFS** (Hadoop Distributed File System) follows a very similar architecture.

# GFS files and chunks

- ⌘ Files divided into fixed sized **chunks**
  Each chunk is identified by an immutable and globally unique 64 bit chunk handle

- ⌘ Chunk servers store chunks on local disks
  - As *Linux* files
  - Chunks are replicated across multiple servers (Default: Three replicas)

Data ***reliability*** is maximized by storing replicas across different racks.

Such a placement also helps *aggregate bandwidth* of racks during *read operations*.

However, it causes *multi-rack write traffic*.

# Clients and master

- Clients interact with master for **metadata**
  - Interacts with *chunk servers* directly for *actual data* manipulation

- Caching
  - Clients cache metadata
  - Chunk servers have automatic Linux caching
  - Any other caching not meaningful nor feasible (for the specific workloads)

# The single master

⌘ Maintains all **system metadata in main memory**
  - **Namespace**
  - Mapping from files to chunks
  - Current locations of chunks

Advantages of holding all system metadata in main memory include:
- Performance
- Easy and efficient scans for
  * Garbage collection
  * Re-replication
  * Migration for rebalance

# The single master

⌘ Maintains all **system metadata in main memory**
- Namespace
- **Mapping from files to chunks**
- Current locations of chunks

If the mapping information is lost, then, even if the chunks survive, the file system is useless.

This information is thus **stored persistently**, logging mutations in an operations log stored in a local disk (of the master) and at *master replicas*.

# The single master

⌘ Maintains all **system metadata in main memory**
- Namespace
- Mapping from files to chunks
- **Current locations of chunks**

Periodic **HeartBeat** with *chunk servers* to *keep track of status, chunk locations*, etc.

It is very hard to keep all the information synched persistently, and it is thus best for the chunk servers to claim ownership, and when a (new) Master reboots, it needs to gather this information from *chunk servers* before starting regular operations.

# Chunk size choice

64MB Chunks

- ⌘ Reduces **size of metadata stored at master**
  - Fits in memory ➔ Significant performance boost (<64 bytes metadata per chunk at Master)
- ⌘ Reduces **clients' need to interact with master**
  - Many *operations are contiguous*/sequential on a file, and *involves the same chunk server*
  - Client can cache chunk locations for even a multi-TB working set
- ⌘ Clients likely to carry out **more operations on same chunk**
  - Amortizes network connection costs (persistent TCP connection with chunk servers)

# Large chunk implications

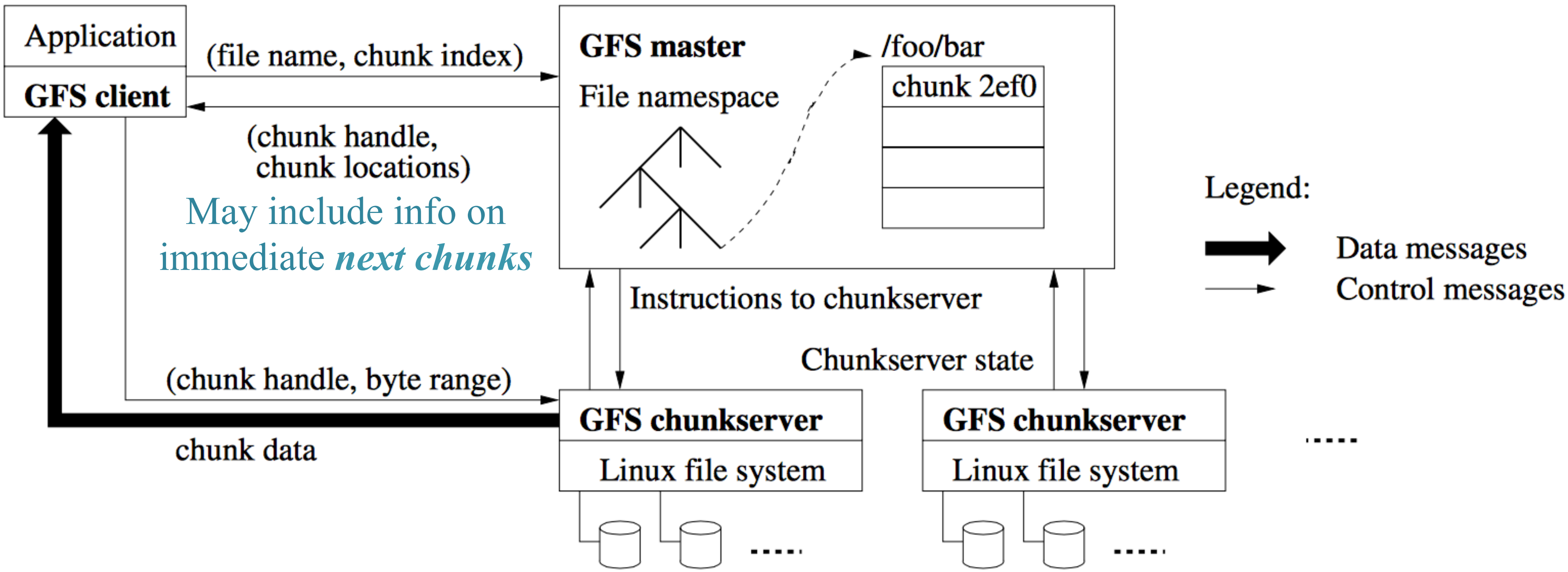
There is the chance of **fragmentation**, and **poor utilization of space**. This however happens infrequently for the specific workloads (large file sizes, so only last chunk wastes space). Lazy space allocation further mitigates wastage of space.

There are also possibilities of **hotspots** (due to small but popular file).

64MB Chunks

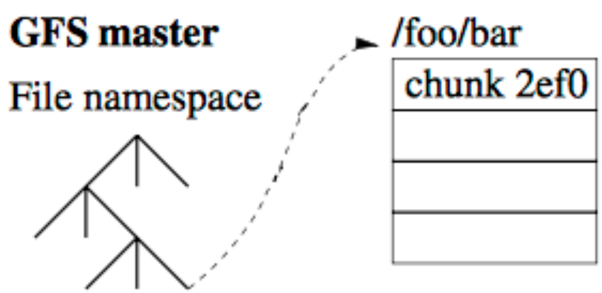
# GFS Architecture

- Client determines **chunk index** based on fixed chunk size and byte offset from application



# Namespace and Locks

- GFS **Namespace**
  - No per directory data structure
  - Lookup table:
    * Maps full path name to metadata
    * Prefix compression




- Each Namespace node has associated **read/write lock**

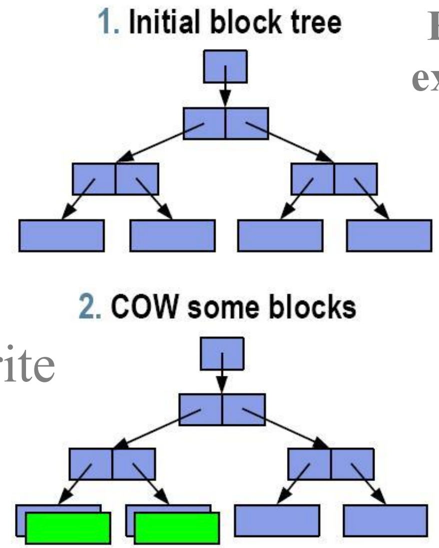
  To manipulate /d1/d2/…/dn/leaf
  - Obtain *read* locks for /d1, /d1/d2, … /d1/d2/…/dn
  - Obtain *read or write lock* for /d1/d2/…/dn/leaf

Since it does NOT use any inode-like data structure:

File creation does not need write lock on parent directory, and allows concurrent mutations in same directory.

# Snapshot

⌘ Uses **copy-on-write (COW)**
- Upon receiving snapshot request Master revokes outstanding leases
- Future write requests through Master who then creates new copy just before write
- New chunk copy is created locally at each affected chunk server




Excerpt of COW example from **ZFS**

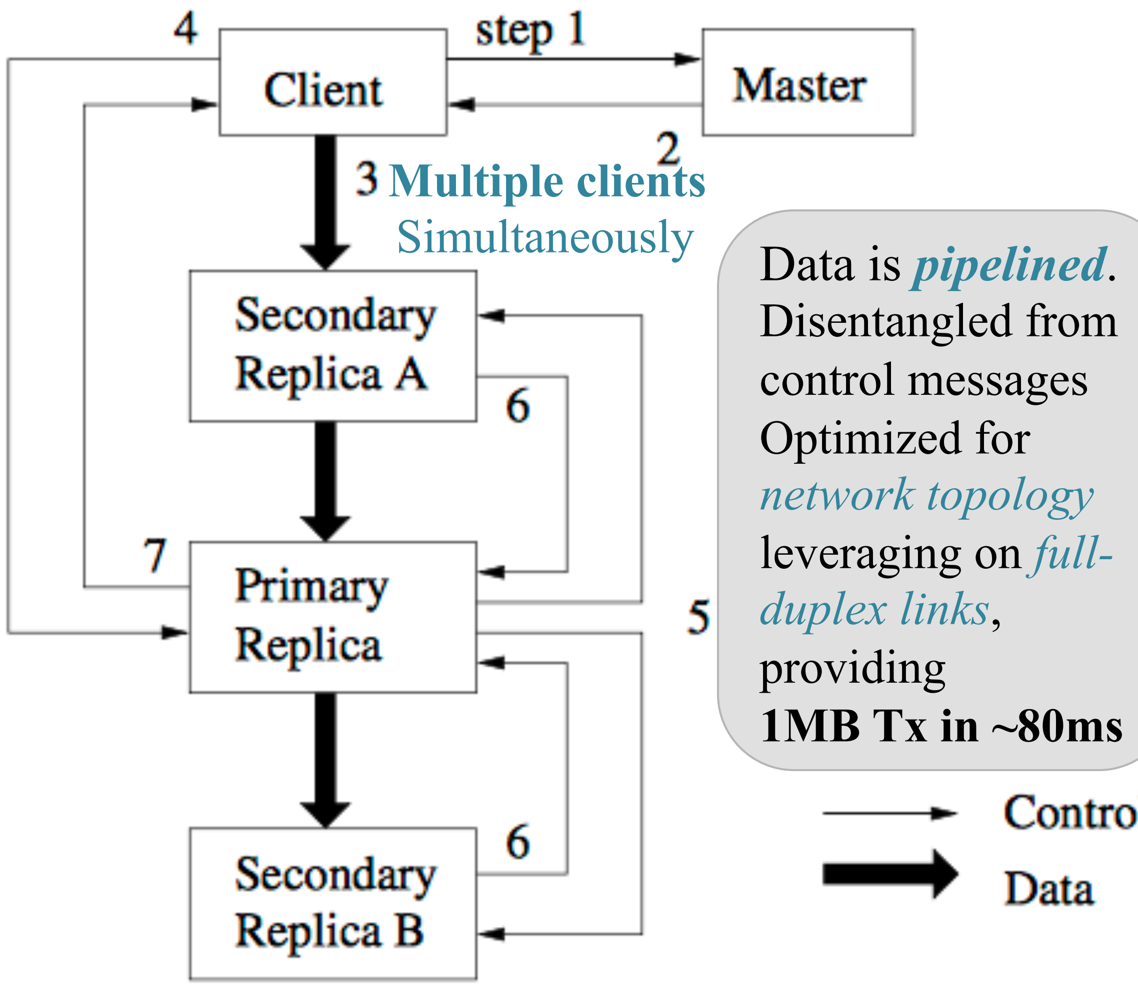
The locking mechanism prevents a file **/home/user/foo** from being created while /**home/user** is being snapshotted to **/save/user**. The snapshot operation acquires read locks on **/home** and **/save**, and write locks on **/home/user** and **/save/user**.

The file creation acquires read locks on **/home** and **/home/user**, and a write lock on **/home/user/foo**. The two operations will be serialized properly because they try to obtain **conflicting locks** on **/home/user**.

# Leases (for mutation)

- Chunk replica given lease to act as **primary replica**
  - Typically for 60sec
    - * renewable
    - * over *HeartBeat*
  - Master can revoke lease

- Global mutation order
  - Lease grant order
  - Serial order inside a lease (determined by *primary*)




**Multiple clients**
Simultaneously

Data is ***pipelined***. Disentangled from control messages Optimized for *network topology* leveraging on *full-duplex links*, providing **1MB Tx in ~80ms**

# Several other considerations

- Atomic record appends
  Special/different from writes, since
  Google had append heavy workload
- Consistency model
- Data integrity, stale data
- Garbage collection
- Replica (re)creation, re-balancing

# Drastic change of landscape

⌘ Early Google
- Latency insensitive batch processes
- Applications seeking document "snippets"

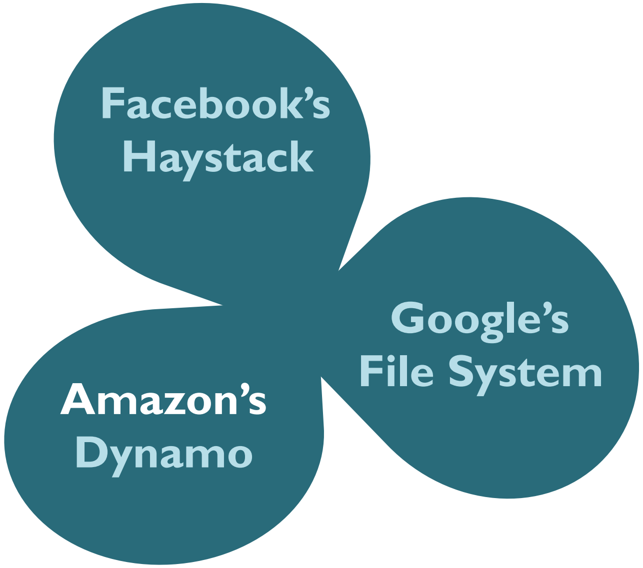
⌘ Google on its way to Alphabet
- Many kinds of workload, e.g.
  Gmail: Seek heavy, latency sensitive
  Docs: Live collaboration in small groups
  Google Cloud Platform: Cloud service

⌘ GFSv2
- **Colossus**: Uses Reed Solomon codes (1.5x)
- Sharded metadata layer

The work on GFS has been followed by many subsequent systems developed at Google, such as replacement GFSv2, a.k.a. **Colossus**, as well as functionalities built on top of (or in addition to) the file system – e.g., BigTable (structured storage), Spanner (database), Megastore (strong consistency w/ ACID), F1 (distributed SQL DB) etc.

# DYNAMO

## KEY-VALUE STORE



*Reference*

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

DeCandia et al.

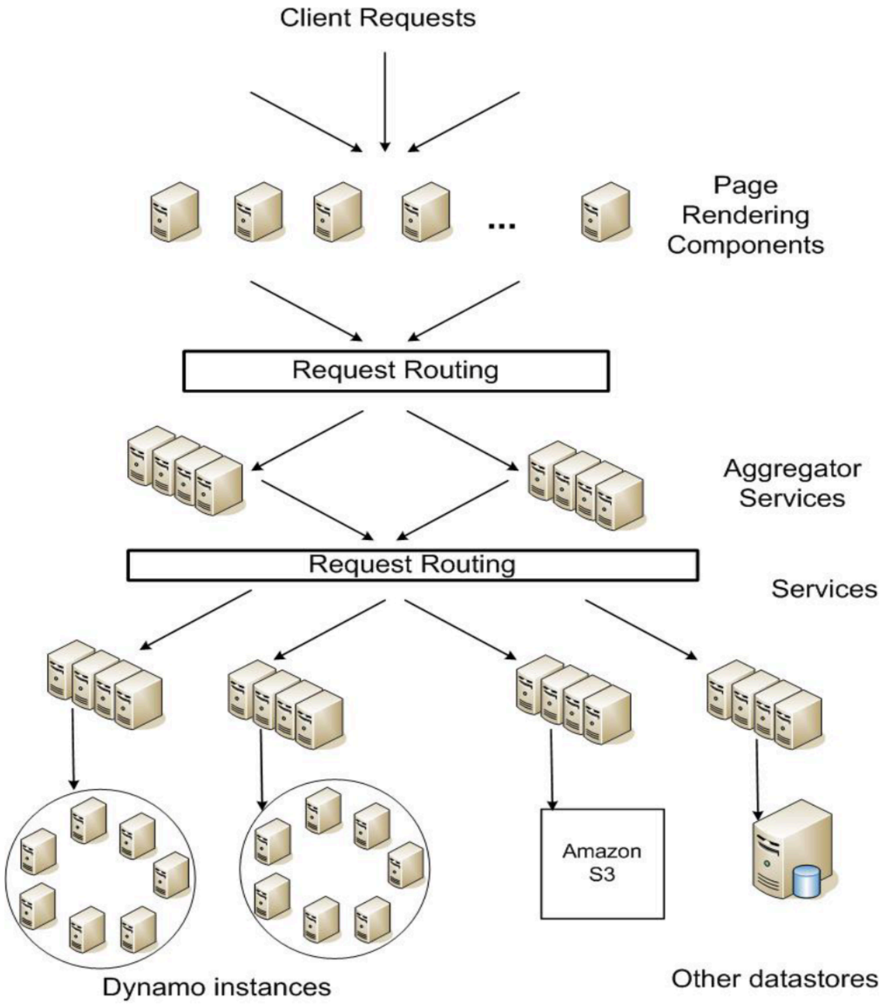
**SOSP 2007**

# Need for speed

- Latency sensitive
  e.g., Shopping cart service
  - 10s of millions of requests per day
  - Millions of checkouts each day
  - Hundreds of thousands of concurrent activities

- A single page request
  100s of services
  - Multiple service **dependencies**

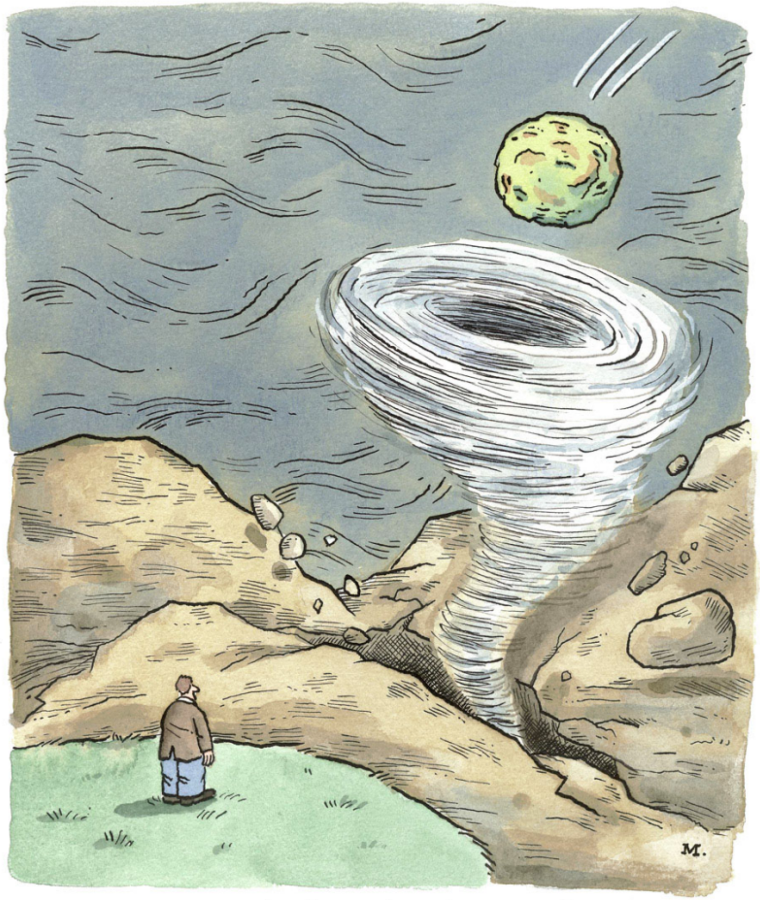
- Stringent **SLAs** for each service
  $99.9^{th}$ percentile < 300 ms
  - mean/std. deviation inadequate



# Need for extremely high availability

- The show must go on
  Downtime ➔ Lost business

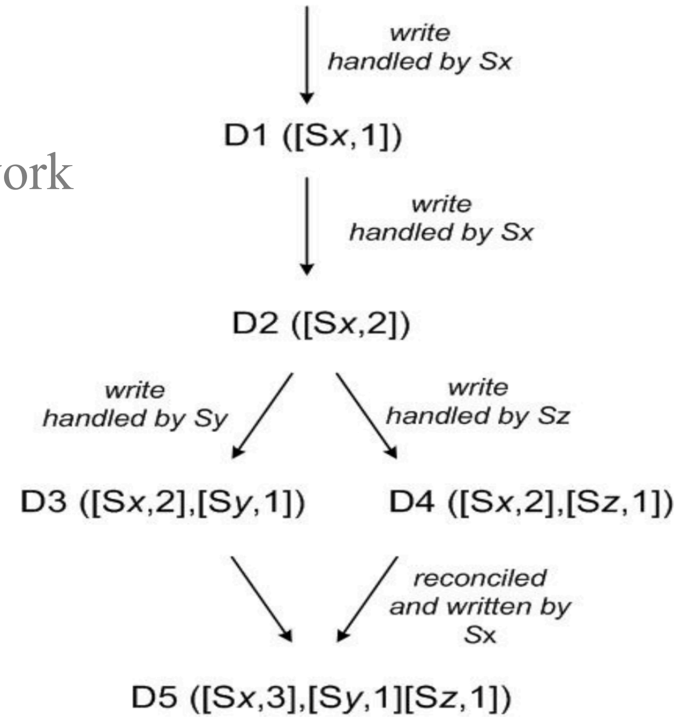
*"customers should be able to view and add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornados"*




The dreaded earthquake – enhanced tornado being hit by a radioactive meteor event.

# The show must go on

- Infrastructure comprises of **millions of components**
  - tens of thousands of servers located across many data centers world-wide
  - a small but **significant number** of server and network components that are failing **at any given time**
- Redundancy for **fault-tolerance**
  CAP theorem
  - **Trade-off** choice: Availability over consistency
- An **always writable** data store
  **Conflict resolution** complexity at reads
  - Unlike most traditional DBs
  - Typically handled at clients (based on application logic)
  - default fall-back option “last write wins”

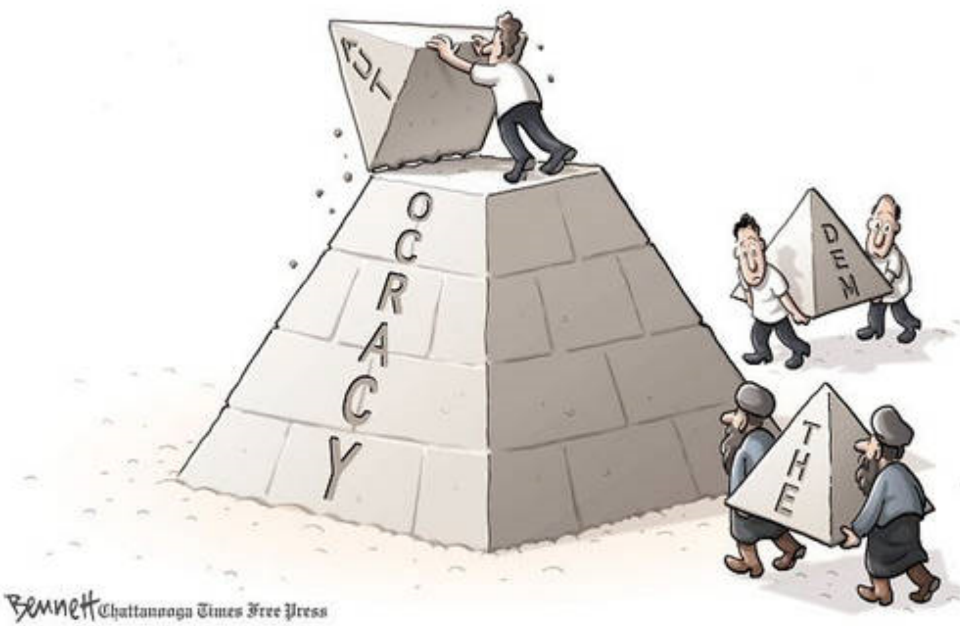


# KISS: Keep it simple, stupid!

- Both **stateful** (needing persistent storage) and **stateless** services
- Most data manipulation using primary keys
  - No need for complex queries
  - Individual operations don't span multiple data items
  - Relatively **small objects** (<1MB)
- RDBMS is an **overkill**
  - Much more expensive & complex
  - Hard to scale (out)

# Dynamo design considerations

- **Incremental** scalability
  Partitioning load using consistent hashing
- Symmetric role of constituent machines
  Simpler system design, provisioning & maintenance
- **Decentralized**
  No single point of failure/bottleneck
  Self-organizing
- Heterogeneity friendly




Changes need to be well thought out!

# Dynamo: DHT Interface

- **get(key)**
  Locate object replicas associated with the key
  Return object/list of objects, with context
- **put(key, context, object)**
  Context encodes system meta-information, e.g., version
- MD5[Caller key] → 128 bit identifier
  (to determine storage nodes)

# Dynamo: Architecture

⌘ **Zero-hop DHT**

Consistent hashing based data partitioning

All nodes know all other nodes

⌘ Multiple `tokens' per node

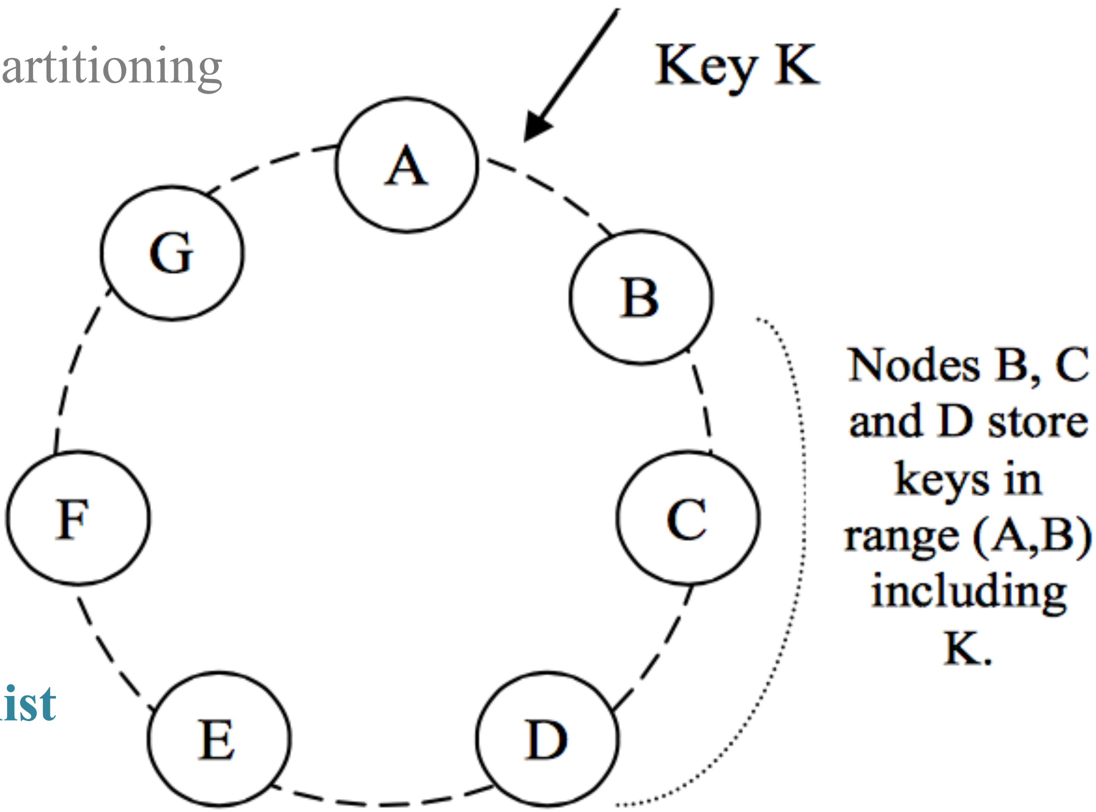
Virtual node instances

- Easy to handle heterogeneity
- Better load-distribution when nodes arrive/depart

⌘ Configurable replication

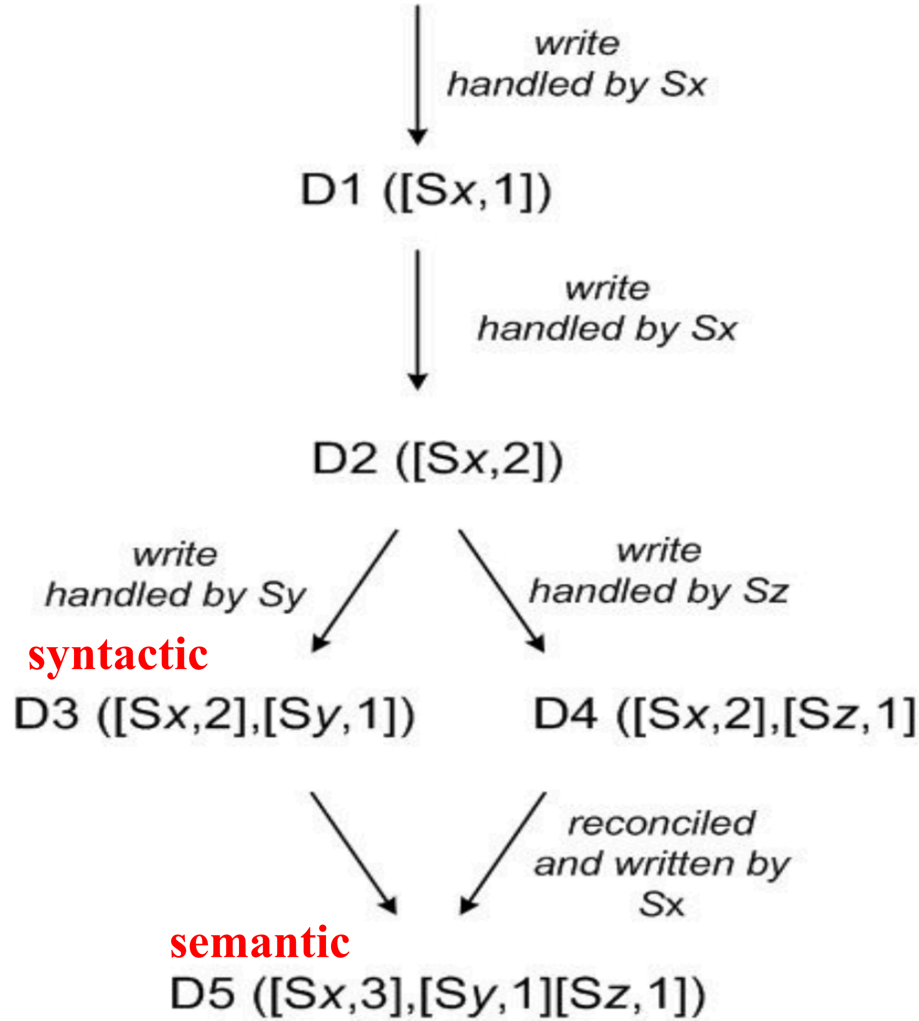
For any given key: **preference list**

- Ensure distinct physical nodes
- Choice to choose across multiple data centers



# Dynamo: Data versioning

- ⌘ Many potential coordinators per key
  Coordinators:
  Nodes handling reads/writes
- ⌘ Versions: Vector clocks
- ⌘ Reconciliation
  - syntactic
  - semantic



# Dynamo: Executing get(), put()

- Symmetry: Client sends get/put request for any key to any Dynamo node

- Sloppy quorum: First N healthy nodes from the preference list

**R+W >N**

Object availability, consistency & durability **trade-offs**

- Upon receiving **put()** request
  - coordinator generates vector clock & writes locally
  - sends to N highest-ranked reachable nodes
  - **W-1 acks** implies a successful write

# Dynamo: Executing get(), put()

- Symmetry: Client sends get/put request for any key to any Dynamo node

- Sloppy quorum: First N healthy nodes from the preference list

**R+W >N**

- Upon receiving get() request
  - coordinator requests for all existing versions from **N highest-ranked nodes**
  - waits for **R responses**, gathers all versions, and sends (to client) all causally unrelated versions

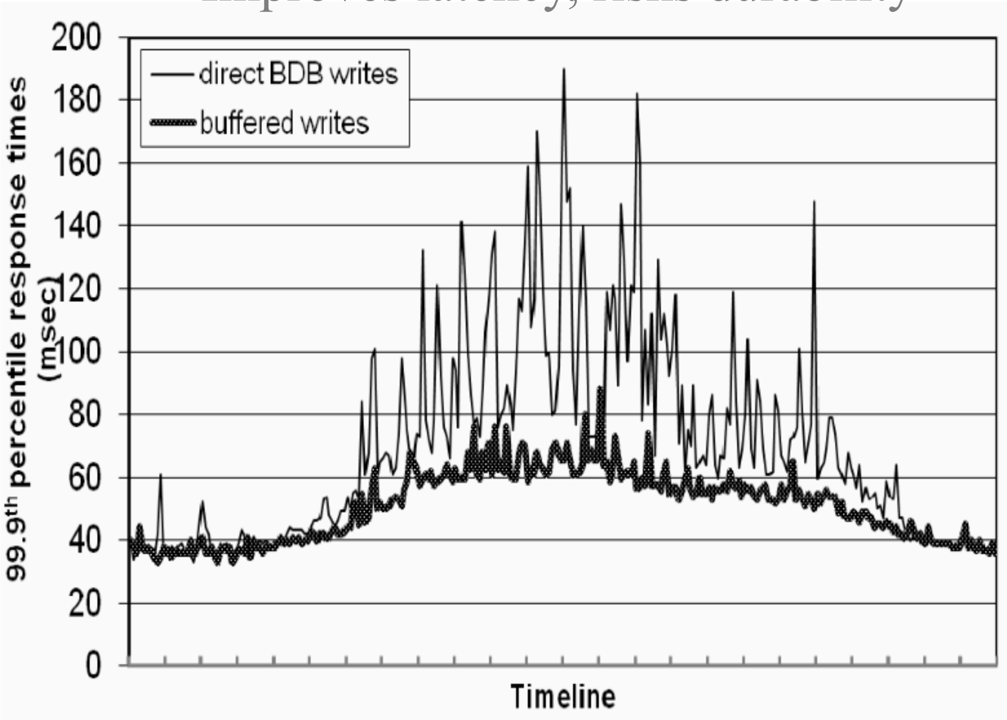
**Hinted handoff** (replication) for **always writable** (availability) & durability.

Syntactic **reconciliation** at Dynamo node, semantic one at client.

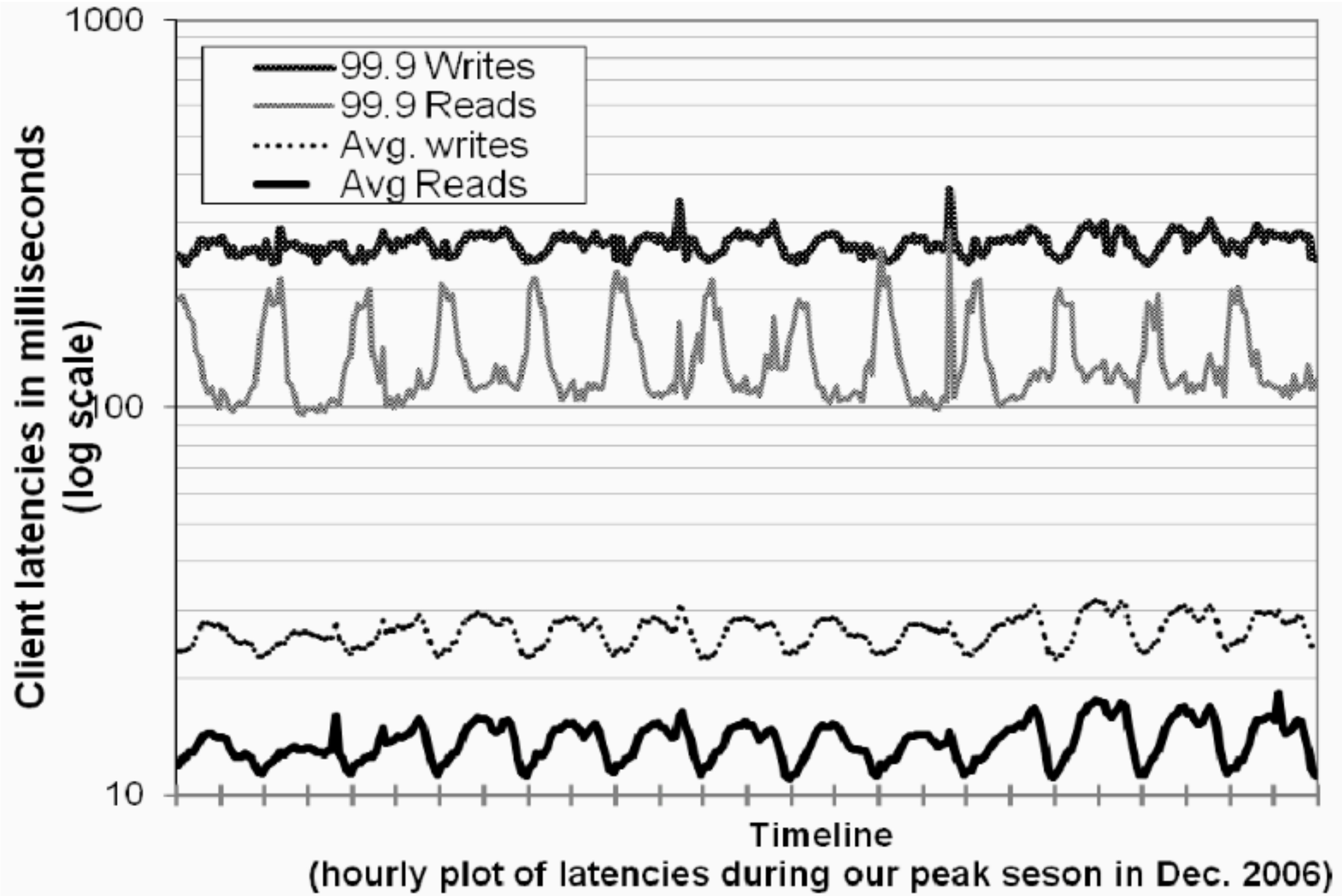
# Heuristic optimizations

⌘ Gossip algorithms
- Failure detection
- Membership information propagation

⌘ Buffered writes
- Writes stored in main memory buffer, periodically written to storage
- Improves latency, risks durability



# Dynamo in action



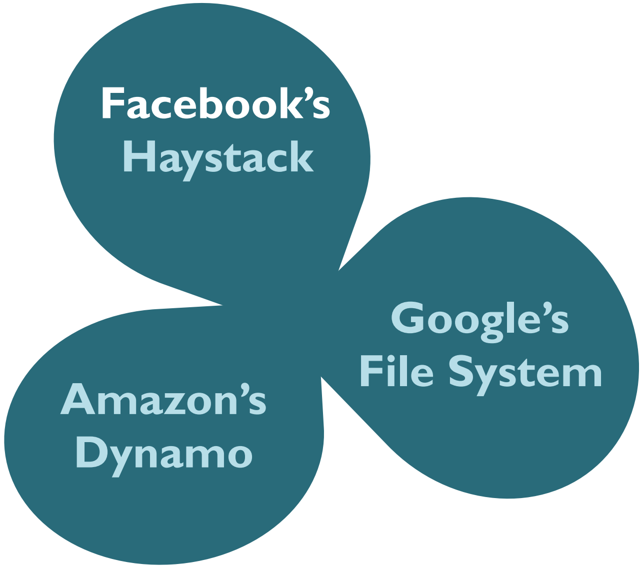
# Dynamo wrap up

- Dynamo provides a bare-bone storage service
  - Key-Value Store
- Foundations for **DynamoDB**
- High availability (always write)
  - Sacrifices consistency, durability

| 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. |

# HAYSTACK

## OSN PHOTO STORAGE



*Reference*

**Finding a needle in Haystack: Facebook's photo storage**
Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, Peter Vajgel
**OSDI 2010**

# Workload characteristics

- 260 billion images (~20 petabytes of data)
  65 billion images, stored in four different sizes
- One billion new images images per week (~60 terabytes)
- Serves ~ one million images per second at peak

Facebook's stats from 2009-10

| Excessive disk I/Os due to meta-data lookup | Reduce per-photo metadata. All metadata lookups in main memory | Conserve disk operations for reading actual data |
| --- | --- | --- |
| Traditional design pain point | Guiding design principle | Ultimate objective |

# NAS mounted over NFS

- POSIX based file system is an overkill
  unused metadata: directories, permissions, …

- Disk IOs for metadata cause bottleneck
  - Poor read throughput
  - Insignificant in small scale, but
  - Significant for billions of files and
    millions of read operations per second

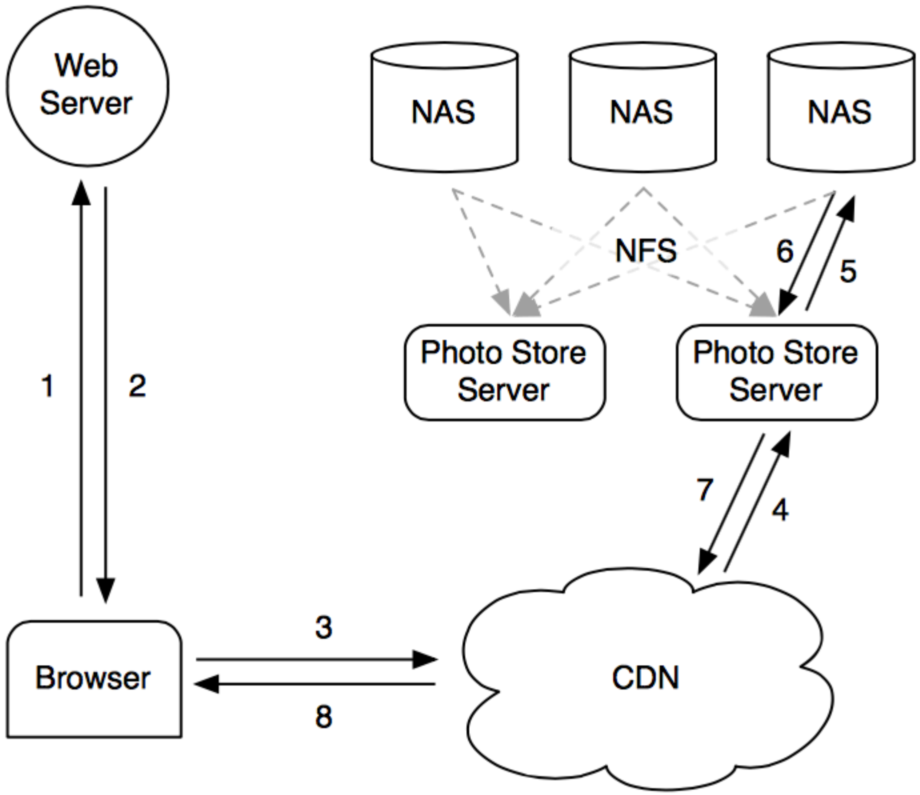
disk I/Os due to **meta-data lookup** in order to find and access the actual file

Multiple I/Os for **Filename → inode number** → **Read inode** from disk → Read the **actual file**

# Typical NAS over NFS design

- CDN is good for hot data
  OSNs have a long *tail request* pattern

- Caching at any level has limited benefit with long tail requests

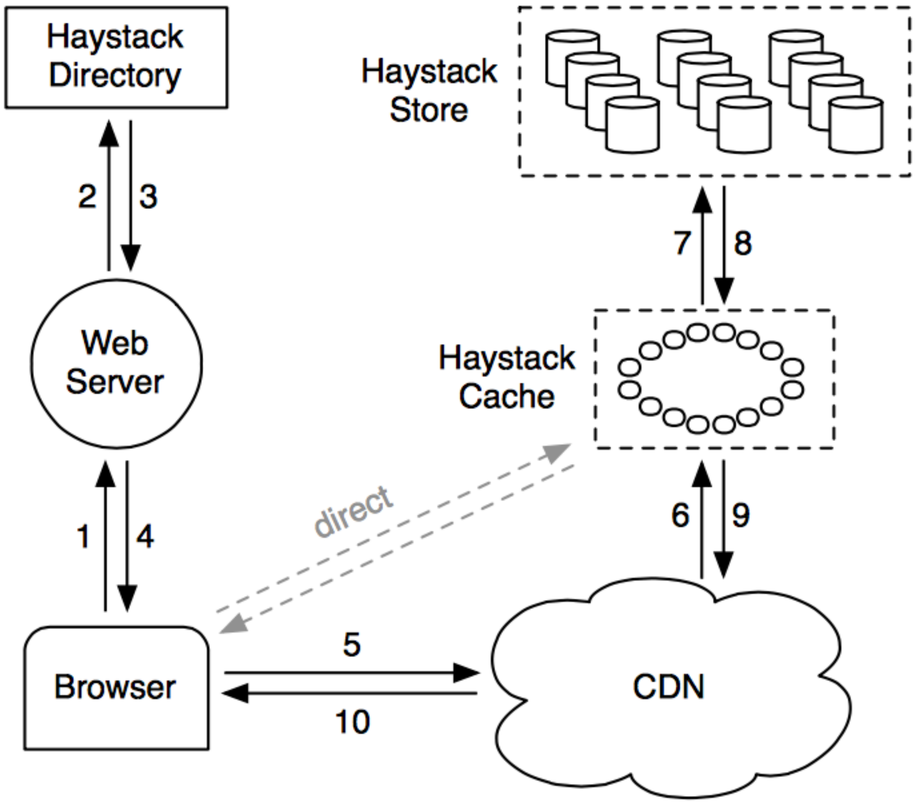


# Haystack: Big picture

⌘ Three components
  Store, Directory & Cache

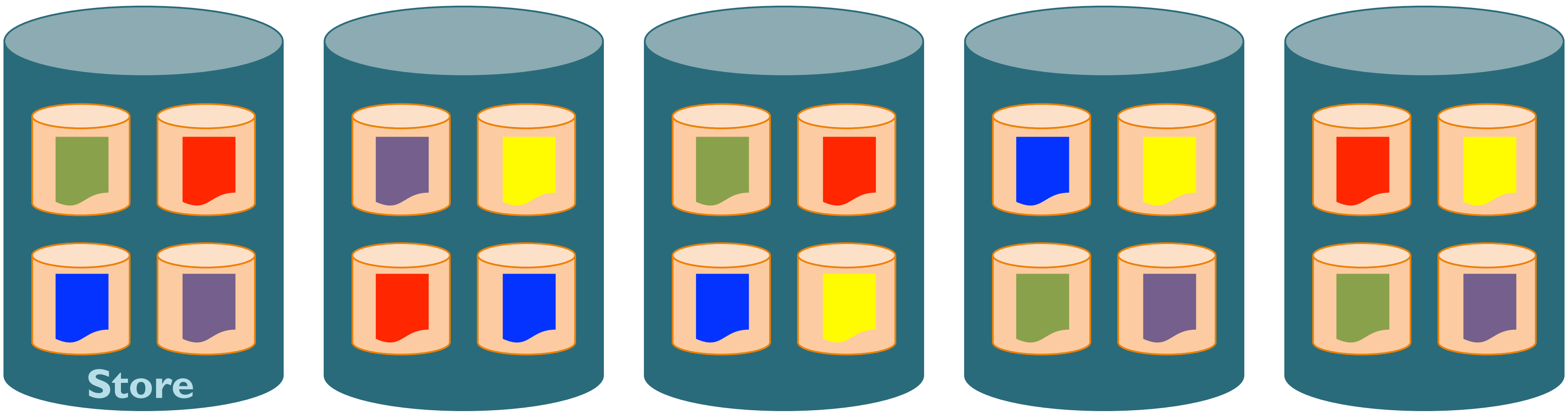
⌘ Physical volumes
  e.g., 10 TB server as
        100 physical volumes of 100GB

⌘ Logical volumes
  - group physical volumes from different machines
  - photo stored on a logical volume
    ➔ written to all corresponding physical volumes



# Haystack: Big picture

- Physical volumes
  e.g., 10 TB server as
  100 physical volumes of 100GB
- Logical volumes
  - group physical volumes from different machines
  - replication based fault-tolerance



Store machine

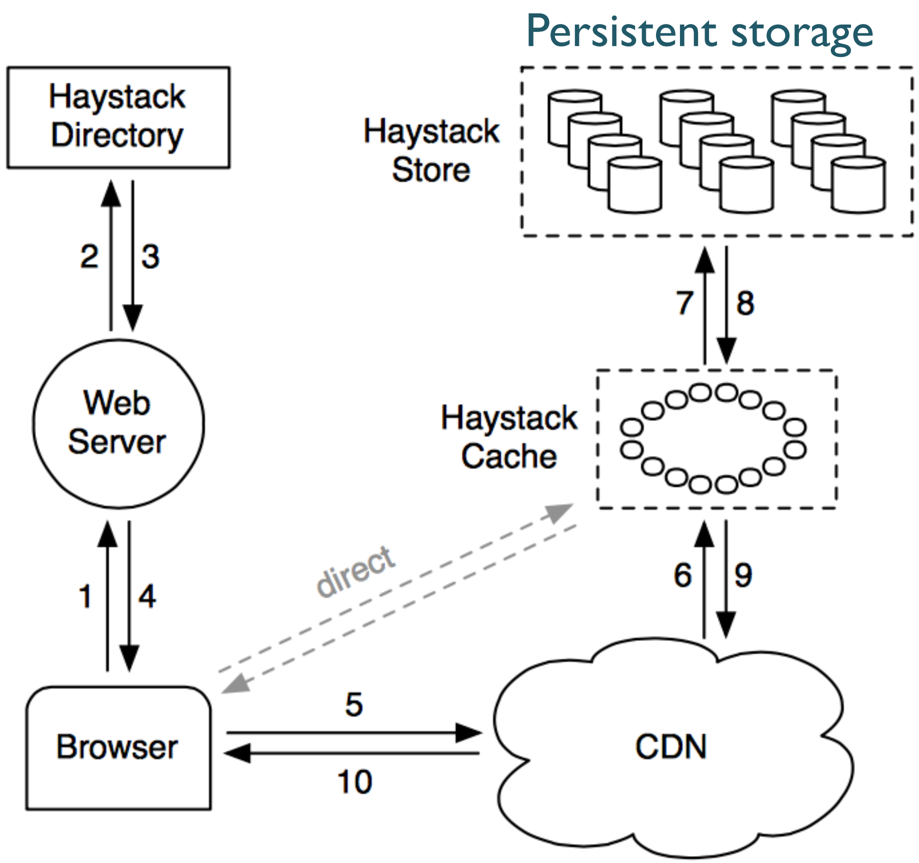
Uses RAID within

# Haystack: Big picture

⌘ Webserver uses directory
to create URL for each photo

http://⟨CDN⟩/⟨Cache⟩/⟨Machine id⟩/
⟨Logical volume, Photo⟩

- Tells: Which CDN to use? Or to use the cache directly
- **IF** not found in CDN and/or cache **THEN** contact machine



# Haystack: Directory

- Provides mapping from logical volumes to physical volumes
  Web servers use this mapping when
  - uploading pictures
  - constructing image URLs for a page request

Replicated database + memcache accessed via PHP

- Load balance
  - writes across logical volumes
  - reads across physical volumes

http://⟨CDN⟩/⟨Cache⟩/⟨Machine id⟩/⟨Logical volume, Photo⟩

- CDN or cache?
  - adjust dependence on cache

http://⟨CDN⟩/⟨Cache⟩/⟨Machine id⟩/⟨Logical volume, Photo⟩

- Identifies read-only logical volumes
  - operational reasons
  - reached storage capacity

# Haystack: Cache

⌘ HTTP requests from both CDN or browser

⌘ Cache complements (and not supplement) CDN

- Organized as a DHT
- PhotoID as key

**Request is from user (not CDN)**

- Experience: **Post-DCN caching ineffective**
- CDN miss unlikely to result in a hit for internal cache either

**Photo fetched from write-enabled machine**

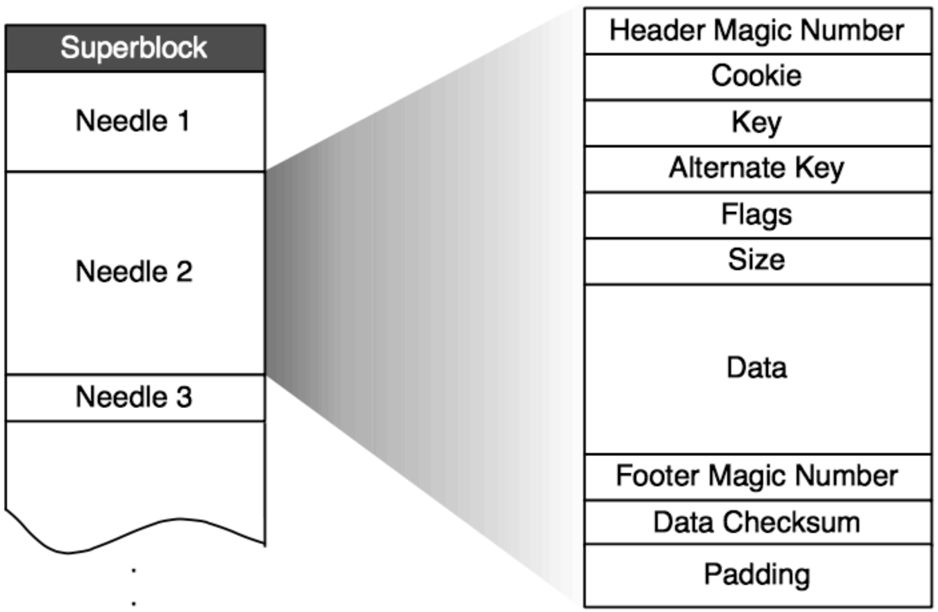
- **Shelter write-enabled Store machines**
- Photos are mostly accessed just after upload
- Underlying file-system **performance** for the given workloads is **better** when carrying out **only reads or only writes**, rather than both

Possible optimization: Prefetch/Proactively push new pictures to Cache

# Haystack: Store

- Multiple physical volumes per Store machine
  - each volume is essentially a very large file (~100GB)
  - holds millions of photos
- Each Needle: Representing a photo

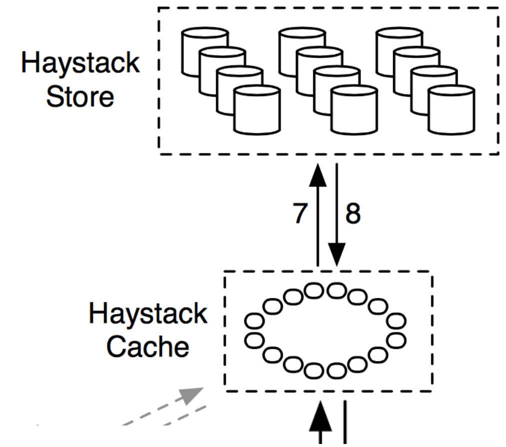
Machines maintain an **in-memory data structure** for each volume, mapping key-pairs to needle's flags, volume offset and size.




| Field | Explanation |
|---|---|
| Header | Magic number used for recovery |
| Cookie | Random number to mitigate brute force lookups |
| Key | 64-bit photo id |
| Alternate key | 32-bit supplemental id |
| Flags | Signifies deleted status |
| Size | Data size |
| Data | The actual photo data |
| Footer | Magic number for recovery |
| Data Checksum | Used to check integrity |
| Padding | Total needle size is aligned to 8 bytes |

# Haystack: Store – photo read

- Read request to Store machine comes from a Cache machine
  Why?




**Cache machine**

Volume ID,
Key,
Alt Key,
Cookie

Cookie prevents random lookups

**Store machine**

Look up metadata in-memory

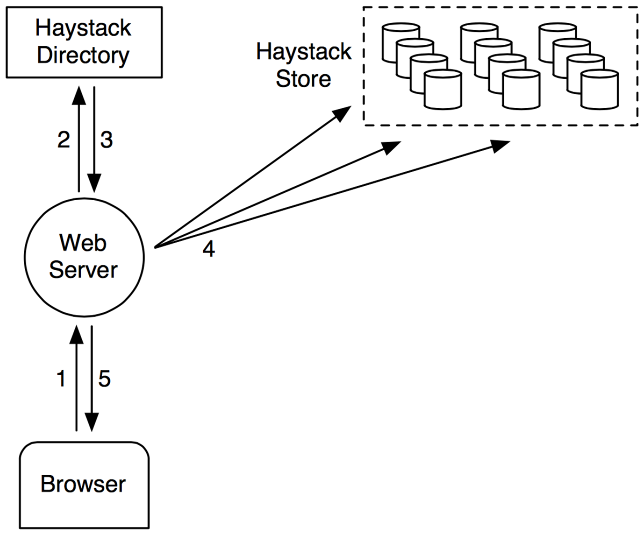
- If file present, read the needle (based on offset & size), check integrity and return

- If file deleted or corrupted, return Error

# Haystack: Store – photo write

⌘ Web server provides to each Store machine
  - logical volume ID, key, alt key, cookie
  - data (the actual photo)

⌘ Each Store machine synchronously appends needle images to the right physical volume
  - update local in-memory mappings




Complications from **append-only** restriction: Photo modifications (e.g., rotate) stored as new needle with same keys.

- If new needle is *in same logical volume*, **Store machines** use *highest offset* to determine *latest version*.
- If new needle is *in different logical volume*, **Directory** updates metadata, future requests never fetch old versions.

# Haystack: Store – photo delete

- ⌘ Store machine sets delete flag
  - in the in-memory mapping information
  - in the volume file
  (why is flag maintained in two places?)
- ⌘ Storage space is temporarily wasted
  - A separate compaction operation to reclaim space
  - Copy a volume file, omitting *duplicate entries* (obsolete versions) and *deleted entries*
  - Freezes the volume from further modifications
  - Meta-data structures updated accordingly

Most deletions happen for "young" photos, and ~ **25% photos are deleted**. Compaction is thus desirable and useful.

But what are the **implications** of such a lazy garbage collection? The photo may continue to survive in the system long after an user has "deleted" it. This has ***privacy implications***, and may have ***legal implications*** too!!

# Haystack: Rebooting Store machines

- Reading physical volumes to reconstruct in-memory mappings is a slow process
  - All the data in the disk has to be read
  - Instead maintain a check-point file (index file) per physical machine
- Index file is updated asynchronously
  - May be stale and exclude some needle entry (*orphans*)
  - Does not contain deletion information

**During machine restarts:**
Store machine sequentially ***examines only all orphans*** (follows immediately after the last needle entry in the index file).

In-memory mappings initialized accordingly, using the index file.

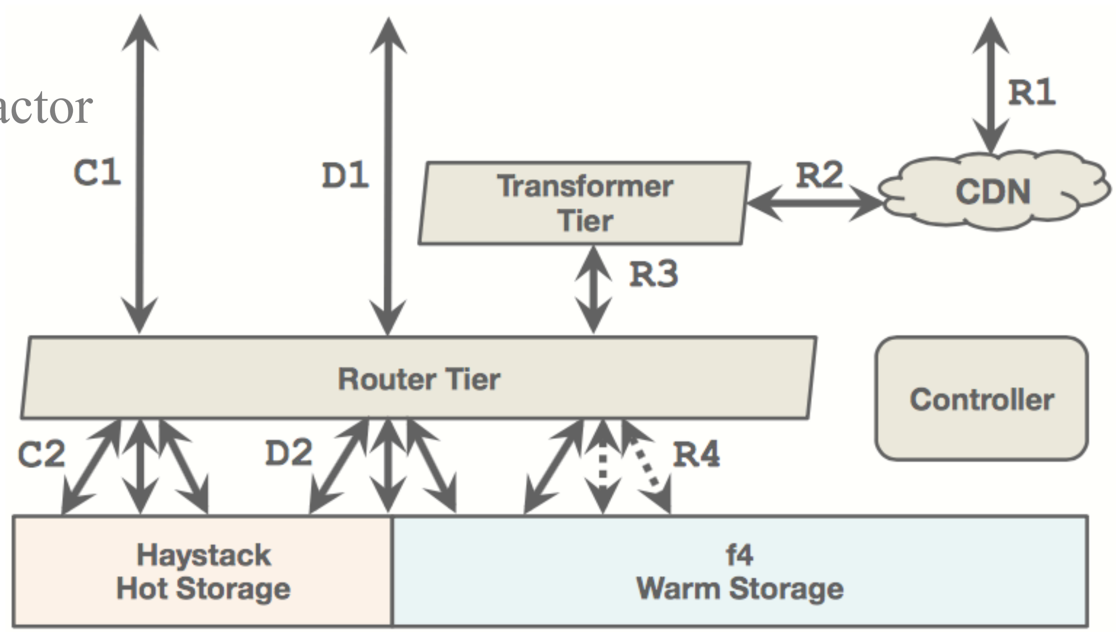
**During normal operations:**
Deleted photos are ***read from volume but not returned*** (based on flag inside needle). *Memory mapping for the needle is updated upon first access.*

# Haystack: Concluding remarks

- Underlying RAID-6 striping may necessitate multiple disk IOs
- Batching multiple writes together can further improve throughput
- Several other optimizations and practical considerations
  pruning in-memory data structure info, choice of file system on Store machines, etc.

# 4 years later … f4

- From 20 petabytes to 65 petabytes
- **Haystack:** Hot storage using **replication** (OSDI 2010)
- **f4: Erasure coding** for not so hot data (OSDI 2014)
  reducing effective-replication-factor from 3.6 to either 2.8 or 2.1



# Conclusion: Cloud scale storage

- Huge Physical Infrastructure
  Requires matching **software solutions**
  Work in progress, but **many solutions** already in place