INGI2145: CLOUD COMPUTING (Fall 2014)

Cloud Storage

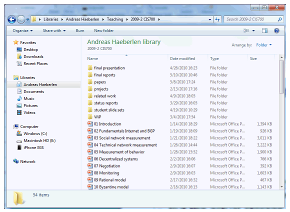
13 November 2014

Certain lecture slides adapted from Upenn NETS212 by A. Haeberlen, Z. Ives
Reproduced with permission

Non final version

- These slides will be updated afterwards with contents from the paper discussed in class

Complex service, simple storage



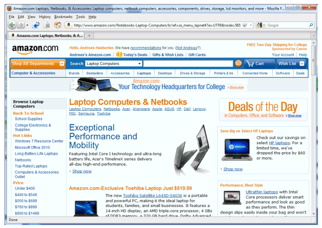

Variable-size files
- read, write, append
- move, rename
- lock, unlock
- ...

Operating system

Fixed-size blocks
- read
- write

- PC users see a rich, powerful interface
  - Hierarchical namespace (directories); can move, rename, append to, truncate, (de)compress, view, delete files, ...
- But the actual storage device is very simple
  - HDD only knows how to read and write fixed-size data blocks
- Translation done by the operating system

Analogy to cloud storage



Shopping carts
Friend lists
User accounts
Profiles
...

Web service

Key/value store
- read, write
- delete

- Many cloud services have a similar structure
  - Users see a rich interface (shopping carts, product categories, searchable index, recommendations, ...)
- But the actual storage service is very simple
  - Read/write 'blocks', similar to a giant hard disk
- Translation done by the web service

What's Wrong with Relational DBs?

- Most applications interact through a database
- Recall RDBMS:
  - Manage data access, enforce data integrity, control concurrency, support recovery after a failure
- Many applications push traditional RDBMS solutions to the limit by demanding:
  - High scalability
  - Very large amounts of data
  - Minimal latency
  - High availability
- Solution is far from ideal

Ideal data stores on the Cloud

- Many situations need hosting of large data sets
  - Examples: Amazon catalog, eBay listings, Facebook pages, …
- Ideal: Abstraction of a 'big disk in the clouds', which would have:
  - Perfect durability – nothing would ever disappear in a crash
  - 100% availability – we could always get to the service
  - Zero latency from anywhere on earth – no delays!
  - Minimal bandwidth utilization – we only send across the network what we absolutely need
  - Isolation under concurrent updates – make sure data stays consistent

The inconveniences of the real world

- Why isn't this feasible?

- The “cloud” exists over a physical network
  - Communication takes time, esp. across the globe
  - Bandwidth is limited, both on the backbone and endpoint
- The “cloud” has imperfect hardware
  - Hard disks crash
  - Servers crash
  - Software has bugs

- Can you map these to the previous desiderata?

Finding the right tradeoff

- In practice, we can't have everything
  - ... but most applications don't really need 'everything'!
- Some observations:
  1. Read-only (or read-mostly) data is easiest to support
     - Replicate it everywhere! No concurrency issues!
     - But only some kinds of data fit this pattern – examples?
  2. Granularity matters: "Few large-object" tasks generally tolerate longer latencies than "many small-object" tasks
     - Fewer requests, often more processing at the client
     - But it's much more expensive to replicate or to update!
  3. Maybe it makes sense to develop separate solutions for large read-mostly objects vs. small read-write objects!
     - Different requirements → different technical solutions

Many situations need hosting of large data sets

Examples: Amazon catalog, eBay listings, Facebook pages, …

- General trend:

From performance at any cost to …

reliability at the lowest possible cost

Key-value stores

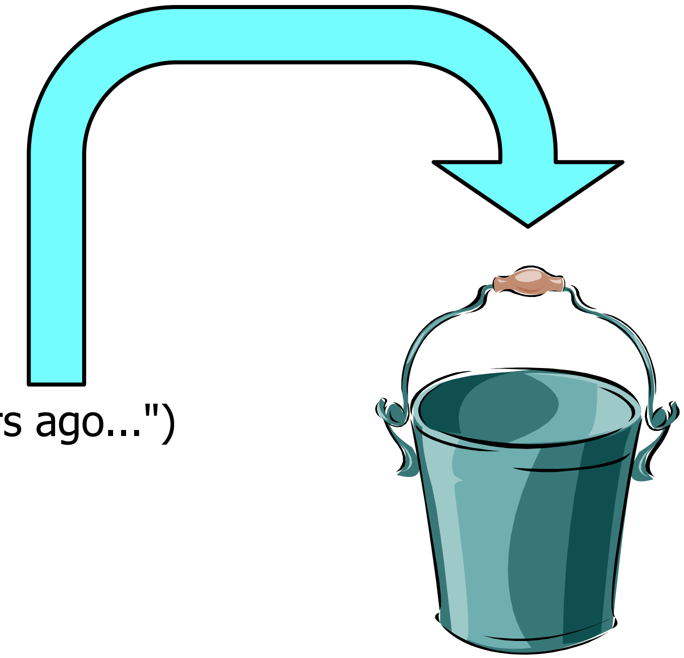
Keys Values

(bob, bschmitt@foo.com)

(gettysburg, "Four score and seven years ago...")

(29ck2dxa1, 0128ckso1$9#*!!8349e)

(windows, )



- The key-value store (KVS) is a simple abstraction for managing persistent state
  - Data is organized as (key,value) pairs
  - Only three basic operations:
    - PUT(key, value)
    - GET(key) → value
    - Delete(key)

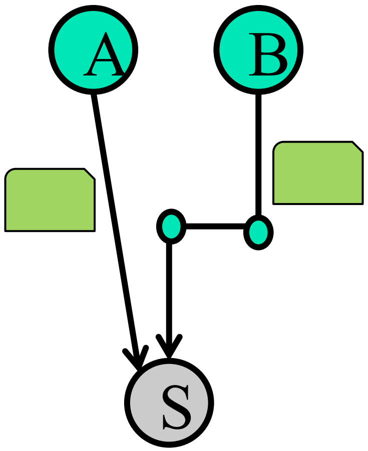
Examples of KVS

- Where have you seen this concept before?

- Conventional examples outside the cloud:
  - In-memory associative arrays and hash tables – limited to a single application, only persistent until program ends
  - On-disk indices (like BerkeleyDB)
  - "Inverted indices" behind search engines
  - Database management systems – multiple KVSs++
  - Distributed hashtables
    - Decentralized distributed systems inspired by P2P (see LSINF2345)
    - Examples: Chord/Pastry

Supporting an Internet service with a KVS

- We'll do this through a central server, e.g., a Web or application server



- Two main issues:
  1. There may be multiple concurrent requests from different clients
     - These might be GETs, PUTs, DELETEs, etc.
  2. These requests may come from different parts of the network, with message propagation delays
     - It takes a while for a request to make it to the server!
     - We'll have to handle requests in the order received (why?)

Managing concurrency in a KVS

- What happens if we do multiple GET operations in parallel?
  - ... over different keys?
  - ... over the same key?

- What if we do multiple PUT operations in parallel? or a GET and a PUT?

- What is the unit of protection (concurrency control) that is necessary here?

Concurrency control

- Most systems use locks on individual items
  - Each requestor asks for the lock
  - A lock manager processes these requests (typically in FIFO order) as follows:
    - Lock manager grants the lock to a requestor
    - Requestor makes modifications
    - Then releases the lock when it's done

Limitations of per-key concurrency control

- Suppose I want to transfer credits from my WoW account to my friend's?
  - … while someone else is doing a GET on my (and her) credit amounts to see if they want to trade?

- This is where one needs a database management system (DBMS) or transaction processing manager (app server)
  - Allows for "locking" at a higher level, across keys and possibly even systems (see LINGI2172 for more details)
- Could you implement higher-level locks within the KVS? If so, how?

No longer one-size-fits-all solution

Specialized data stores

- Example: Amazon’s solutions

- Dynamo [SOSP’07]
  - Many services only store and retrieve data by primary key
    - Examples: user preferences, shopping cart, best seller lists
  - Don’t require querying and management RDBMS functionality
- Simple Storage Service (S3)
  - Need to store large objects that change infrequently
    - Examples: virtual machines, pictures

Specialized data stores

- Example: Google's solutions

- The Google File System [SOSP'03]
  - Distributed file system for large data-intensive applications
  - No POSIX API; focus on multi-GB files divided in fixed-size chunks (64 MB); mostly mutated by appending new data
  - Single master node maintains all file metadata
- Bigtable [OSDI'06]
  - Distributed storage system for structured data
  - Data model is a sparse multi-dimensional sorted map indexed by row and column keys and a timestamp
  - Each value in the map is opaque to the storage system

Specialized data stores

- Example: Facebook's solutions
- Cassandra [Ladis'09]
  - A distributed storage system for large sets of structured data
  - Optimized for very high write throughput; no master nodes
- Haystack [OSDI'10]
  - Object store system optimized for photos
  - In 2010, over 260 billion images; 20 PB of data; 60 TB/week
  - Data written once, read often, never modified, rarely deleted
- TAO [ATC'13]
  - A read-optimized graph data store to serve the social graph
  - Sustains 1 billion reads/s on a changing data set of many PBs
  - Explicitly favors availability over consistency

Specialized data stores

- Example: LinkedIn's solutions

- Kafka [NetDB'11]
  - A high-throughput distributed messaging system
  - Pub/sub architecture designed for aggregating log data
  - Messages are persisted on disk for durability and replicated for fault tolerance; guarantees at-least-once delivery
- Voldemort
  - A distributed key-value store supporting only get/put/delete
  - Inspired by Amazon's Dynamo: tunable consistency, highly available

Let's dive into these 7 systems

- We form groups of 3-4, each with an assigned paper
- We will break off for ~30 minutes
- During that time
  - For 20 minutes, each of you read/scan the assigned paper looking for answers to **one** of a set of questions (next slide)
  - For 10 minutes, discuss within your group what you found
- Then we spend time to share with the class our findings about these systems
  - Short presentation of 5 minutes of the findings of each group and a short Q&A
  - I will discuss too!

What you should look out for

- Not so much the exact details of how it works
  - Highly technical, and somewhat problem-specific

Rather:

- What requirements pushed for a specialized solution?
- What principles were used?
  - How did they make it scale?
  - Why did they make the design decisions the way they did?
  - What kinds of problems did they face?
- What guarantees do these systems give?
- What are the experiences, lessons and practical applications results?

Dynamo

Google File System

Bigtable

Cassandra

Tao

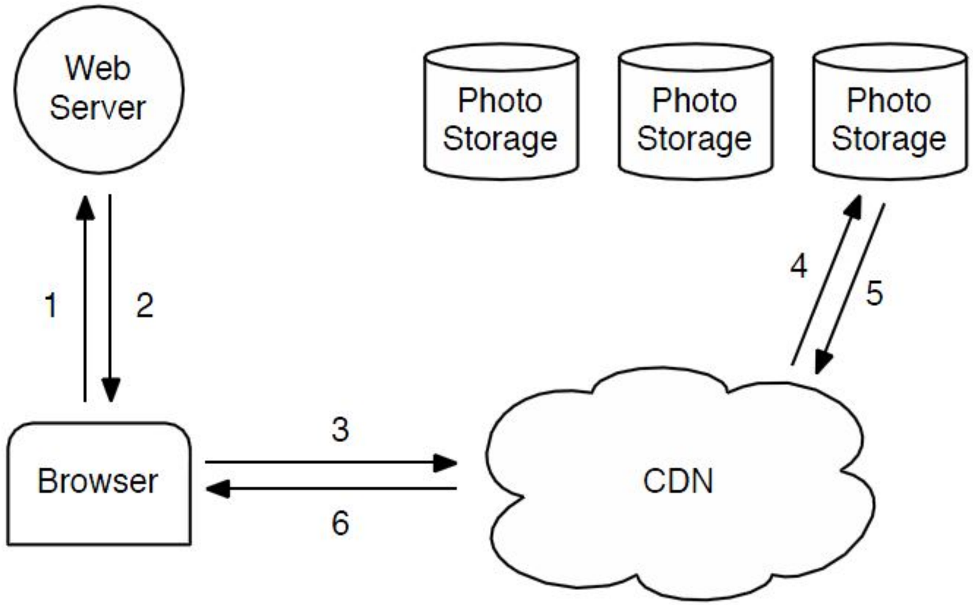
Haystack

Finding a needle in Haystack: Facebook's photo storage

Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, Peter Vajgel

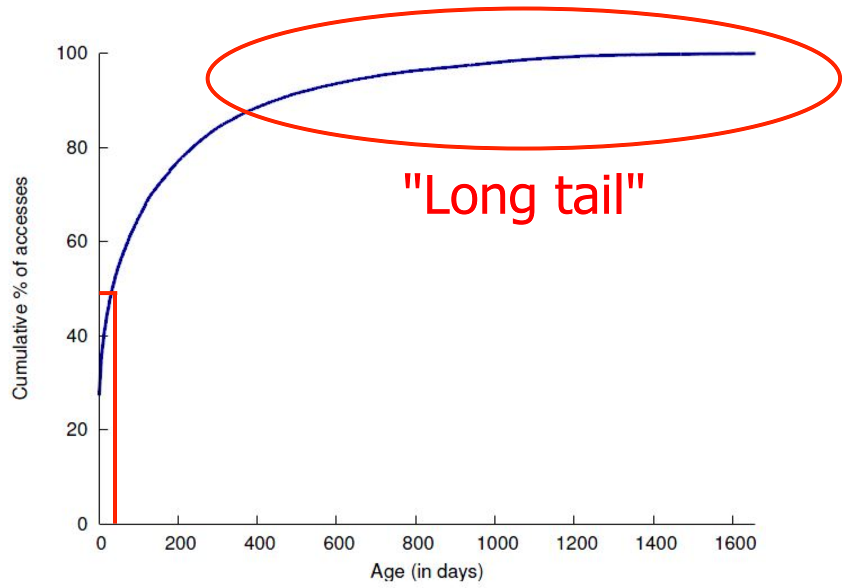
OSDI 2010

Motivation



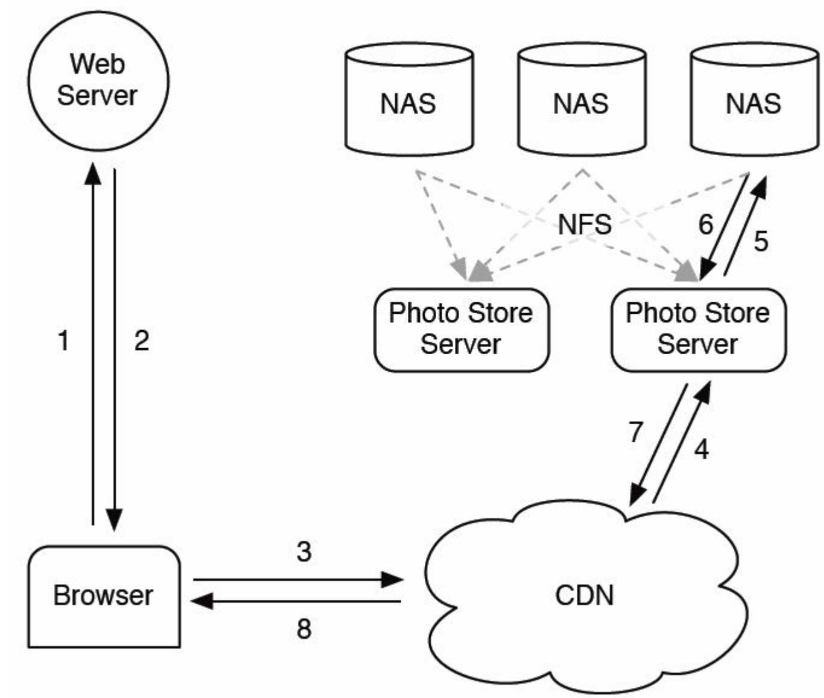
- Facebook stores a huge number of images
  - In 2010, over 260 billion (~20PB of data)
  - One billion (~60TB) new uploads each week
- How to serve requests for these images?
  - Typical approach: Use a CDN (and Facebook does do that)

The problem



- When would the CDN approach work well?
  - If most requests were for a small # of images
  - But is this the case for Facebook photos?
- Problem: "Long tail" of requests for old images!
  - CDN can help, but can't serve everything!
  - Facebook's system still needs to handle a lot of requests

Facebook pre-2010 (1/2)



- Images were kept on NAS devices
  - NAS = network-attached storage
  - File system can be mounted on servers via NFS
  - CDN can request images from the servers

Facebook pre-2010 (2/2)

Directories, inodes, block maps, ...

- Problem: Accesses to metadata
  - OS needs to translate file name to inode number, read inode from disk, etc., before it can read the file itself
  - Often more than 10 disk I/Os - and these are really slow! (Disk head needs to be moved, wait for rotating media, ...)
    - Hmmm... what happens once they have SSDs?

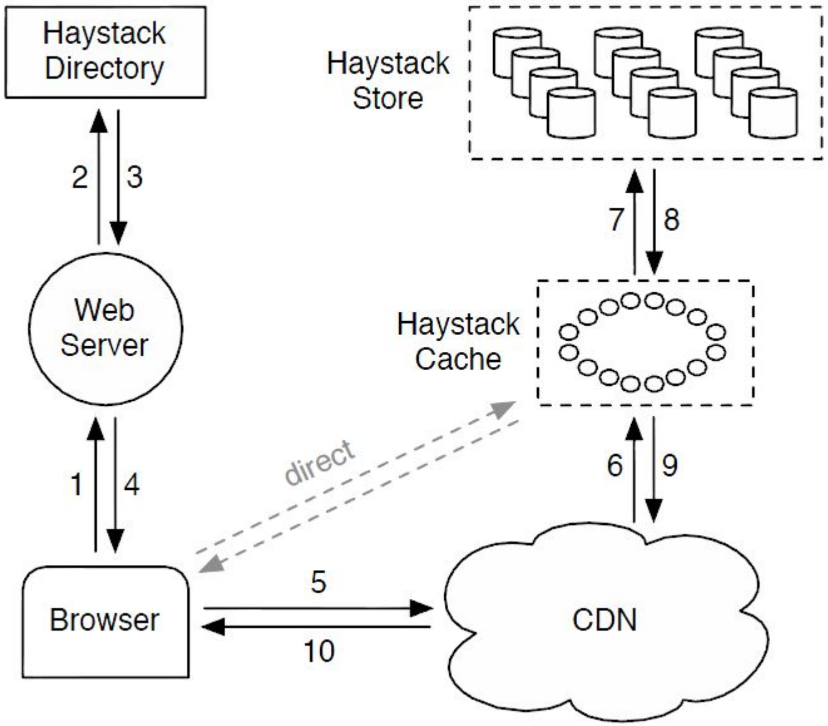
- Result: Disk I/Os for metadata were limiting their read throughput
  - Could you have guessed that this was going to be the bottleneck?

What could be done?

- They considered various ways to fix this...
  - ... including kernel extensions etc.
- But in the end they decided to build a special-purpose storage system for images
  - Goal: Massively reduce size of metadata
- When is this a good idea (compared to using an existing storage system)?
  - Pros and cons - in general?
  - ... and for Facebook specifically?

Haystack overview

- Three components:
  - Directory
  - Cache
  - Store
    - Physical volumes
    - Several of these make up a logical volume
    - Replication - why?

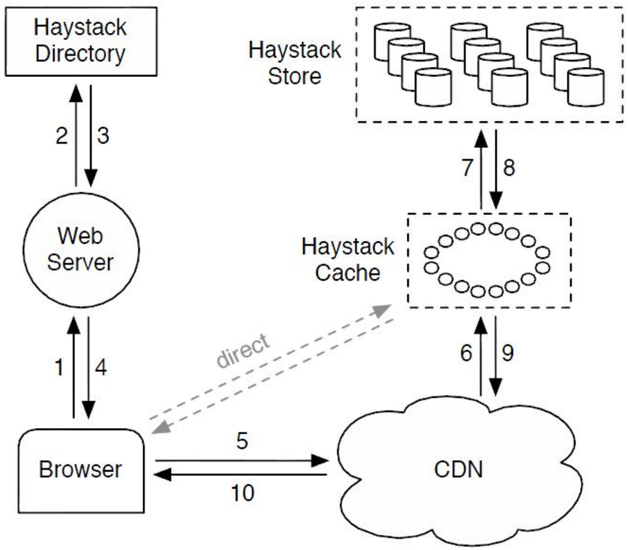


- When the user visits a page:
  - Web server constructs a URL for each image
  - http://(CDN)/(Cache)/(MachineID)/(Logical volume, photo)

Reading an image

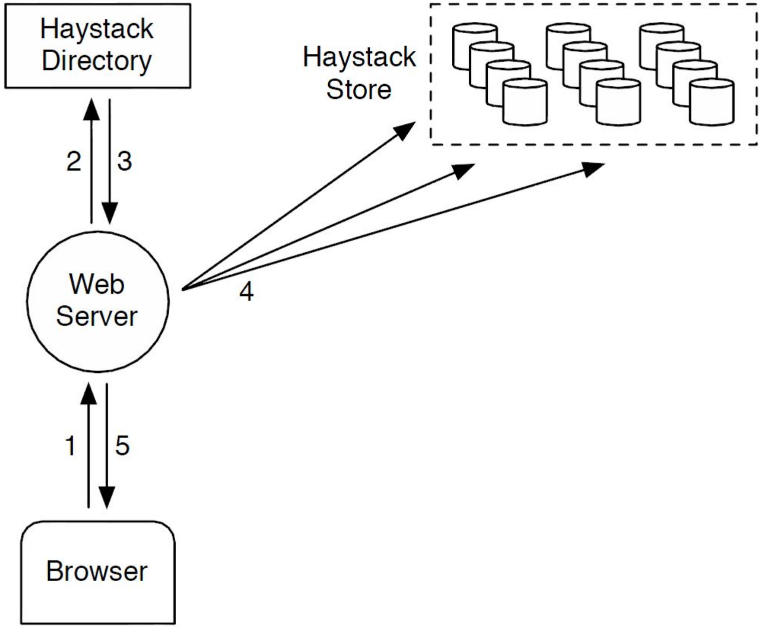
http://(CDN)/(Cache)/(MachineID)/(Logical volume, photo)

- Read path:
  - CDN tries to find the image first
  - If not found, strips CDN part off and contacts the Cache
  - If not found there either, Cache strips off the cache part and contacts the specified Store machine
  - Store machine finds the volume, and the photo within the volume
  - All the necessary information comes from the URL!



Uploading an image

- User uploads image to a web server
- Web server requests a write-enabled volume
  - Volumes become read-only when they are full, or for operational reasons



- Web server assigns unique ID and sends image to each of the physical volumes that belong to the chosen logical volume

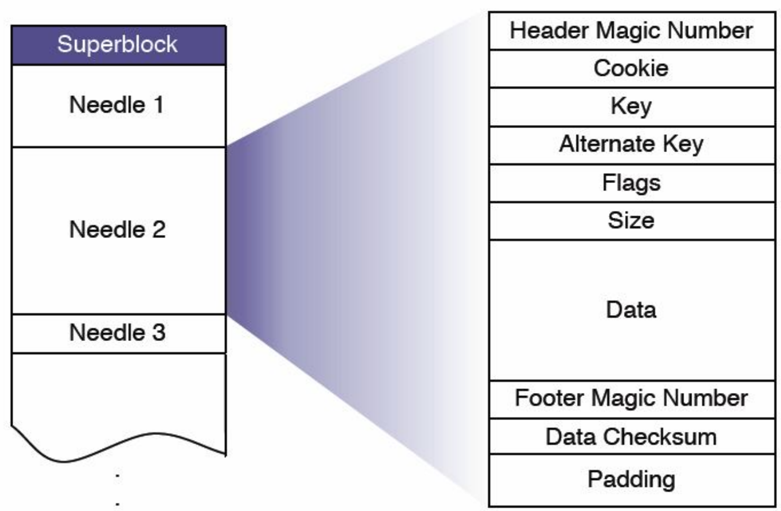
Haystack: The Directory

- Directory serves four functions:
  - Maps logical to physical volumes
  - Balances writes across logical volumes, and reads across physical volumes
  - Determines whether request should be handed by the CDN or by the Cache
  - Identifies the read-only volumes
- How does it do this?
- Information stored as usual: replicated database, Memcache to reduce latency

Haystack: The Cache

- Organized as a distributed hashtable
  - Remember the Pastry lecture?
- Caches images ONLY if:
  - 1) the request didn't come from the CDN, <u>and</u>
  - 2) the request came from a write-enabled volume
- Why?!?
  - Post-CDN caching not very effective (hence #1)
  - Photos tend to be most heavily accessed soon after they uploaded (hence #2)
    - ... and file systems tend to perform best when they're <u>either</u> reading <u>or</u> writing (but not both at the same time!)

Haystack: The Store (1/2)



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

- Volumes are simply very large files (~100GB)
  - Few of them needed → In-memory data structures small
- Structure of each file:
  - A header, followed by a number of 'needles' (images)
  - Cookies included to prevent guessing attacks
  - Writes simply append to the file; deletes simply set a flag

Haystack: The Store (2/2)

- Store machines have an in-memory index
  - Maps photo IDs to offsets in the large files
- What to do when the machine is rebooted?
  - Option #1: Rebuild from reading the files front-to-back
    - Is this a good idea?
  - Option #2: Periodically write the index to disk
- What if the index on disk is stale?
  - File remembers where the last needle was appended
  - Server can start reading from there
  - Might still have missed some deletions - but the server can 'lazily' update that when someone requests the deleted img

Recovery from failures

- Lots of failures to worry about
  - Faulty hard disks, defective controllers, bad motherboards...
- Pitchfork service scans for faulty machines
  - Periodically tests connection to each machine
  - Tries to read some data, etc.
  - If any of this fails, logical (!) volumes are marked read-only
    - Admins need to look into, and fix, the underlying cause
- Bulk sync service can restore the full state
  - ... by copying it from another replica
  - Rarely needed

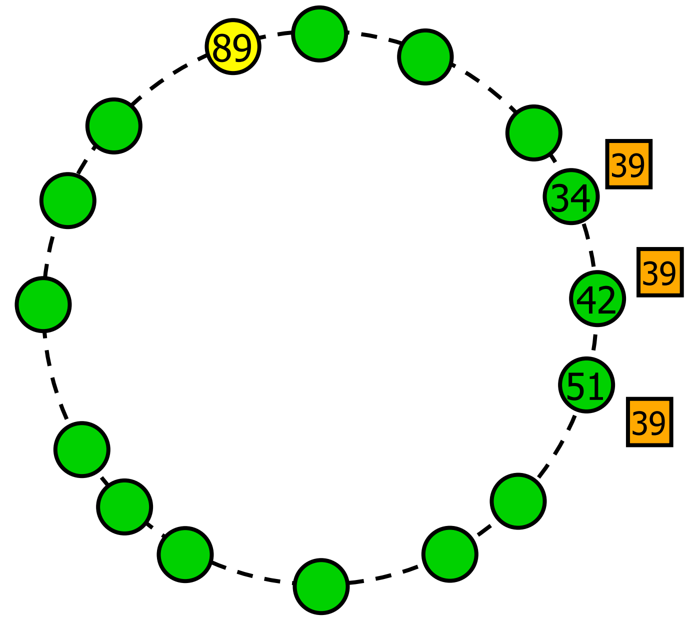
How well does it work?

- How much metadata does it use?
  - Only about 12 bytes per image (in memory)
  - Comparison: XFS inode alone is 536 bytes!
  - More performance data in the paper
- Cache hit rates: Approx. 80%

Summary

- Presence of "long tail" → caching won't help as much
- Interesting (and unexpected) bottleneck
  - To get really good scalability, you need to understand your system at all levels!
- In theory, constants don't matter - but in practice, they do!
  - Shrinking the metadata made a big difference to them, even though it is 'just' a 'constant factor'
  - Don't (exclusively) think about systems in terms of big-O notations!

Consistent hashing



- On which nodes should objects be stored?
  - Assumption: Each object has an identifier too ('key')
- Idea #1: Hashing
  - Example: k nodes, object O is stored on node (O mod k)
  - What happens when nodes join or leave?
- Idea #2: Consistent hashing
  - Object O is stored on node whose ID is closest to O
  - What happens when nodes join or leave?
  - If each object has k replicas, where should we put these?