FINDING A NEEDLE IN HAYSTACK: FACEBOOK'S PHOTO STORAGE

D. Beaver, S. Kumar, H. C. Li, J. Sobel, P. Vajgel,
Facebook Inc.

CPCS 722: Advanced Systems Seminar
Ewa Syta

Plan for Today

- Facebook and Photos
- Why a new system was needed?
  - Old system and issues faced by Facebook
- Haystack Design
- Evaluation
- Q&A

Facebook & Photos in Numbers*

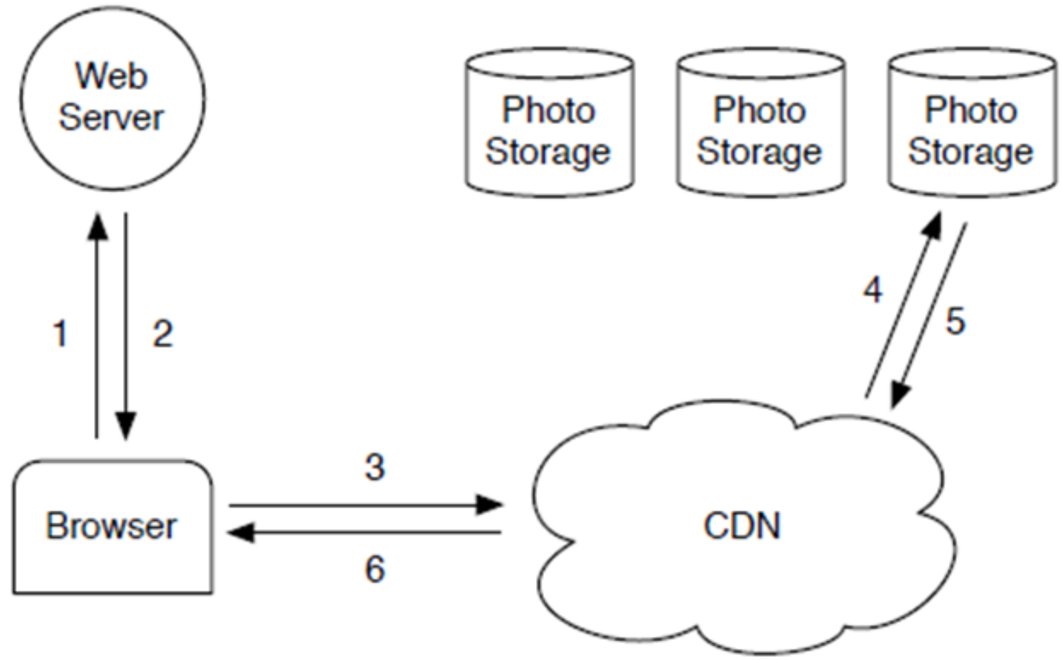
- So far 65 billion photos uploaded
  - Biggest photo sharing website in the world
- One billion new photos uploaded each week
  - ~60 terabytes of data
- One million images per second at peak
- For each photo FB generates and stores four images
  - >260 billion images
  - > 20 petabytes of data

*As of 2010

How Facebook Photos Are Used?

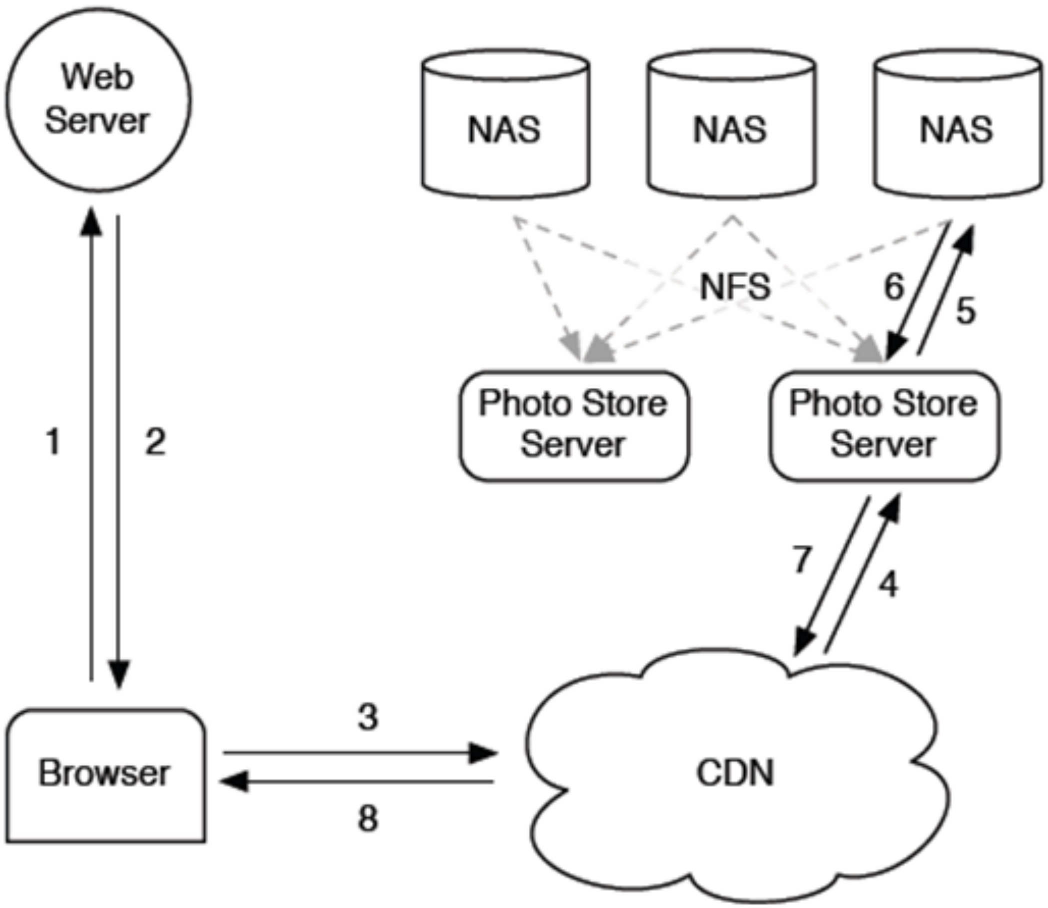
- Profile pictures and pictures recently uploaded
  - Very frequently accessed right after being uploaded
  - Likely to be accessed by different users
  - More likely to be deleted
  - Likely to be cached
- Album photos and older photos
  - Less popular but still frequently accessed
  - Often requested in a sequence by the same user
  - So called ‘long tail’
  - Likely not to be in cache and to be retrieved from the storage hosts
    - So… Why not to cache all of the photos?

TYPICAL DESIGN



1. Browser sends an HTTP request
2. URL for the browser to render
3. For each image there is a URL directing the browser to a location from which to download the data: for popular sites this URL often points to a CDN (Content Delivery Network):
   - If the CDN has it, it responds immediately
   - If not, CDN examines the URL and retrieves the photo from site storage system and updates its cached data

Facebook's Old NFS-Based Design



Old NFS-Based Design

- Each photo stored in its own file on a set of commercial NAS-appliances
- Photo Store Severs (PSS) mount all volumes exported by NAS appliances over NFS
- PSS process HTTP requests for images:
  - Extracts the volume and full path to the file from an image's URL
  - Reads the data over NFS
  - Returns the result to CDN
- Thousands of files stored in each directory of NFS volumes
  - Excessive directory metadata

OLD DESIGN'S ISSUES

- Excessive number of disk operations because of metadata lookups
- Most of metadata not used for photos
  - Waste of storage capacity
  - Requires disk read operations to find the file itself
- Several (~10) disk operations necessary to read a single photo

- The key problem: **disk operations**

First Fix to Reduce Disk Operations

- Reduce directory sizes to hundreds of images per directory
- ~3 disk operations per image
  - (1) read the directory metadata into memory, (2) load the inode, (3) read the file contents

Second Fix

- Let PSS explicitly cache file handles returned by NAS
- Only a minor improvement
- Focusing only on caching has limited impact

Finally… The Haystack!

- No viable solution based on existing systems
  - Existing systems lack the 'right' RAM-to-disk ratio
  - Right ratio? Enough main memory to hold all of the filesystem metadata?
  - One photo corresponds to one file and each file requires at least one inode, which is hundreds of bytes large… Do the math.
- Facebook decided to build their own storage system
  - (not-too) surprising

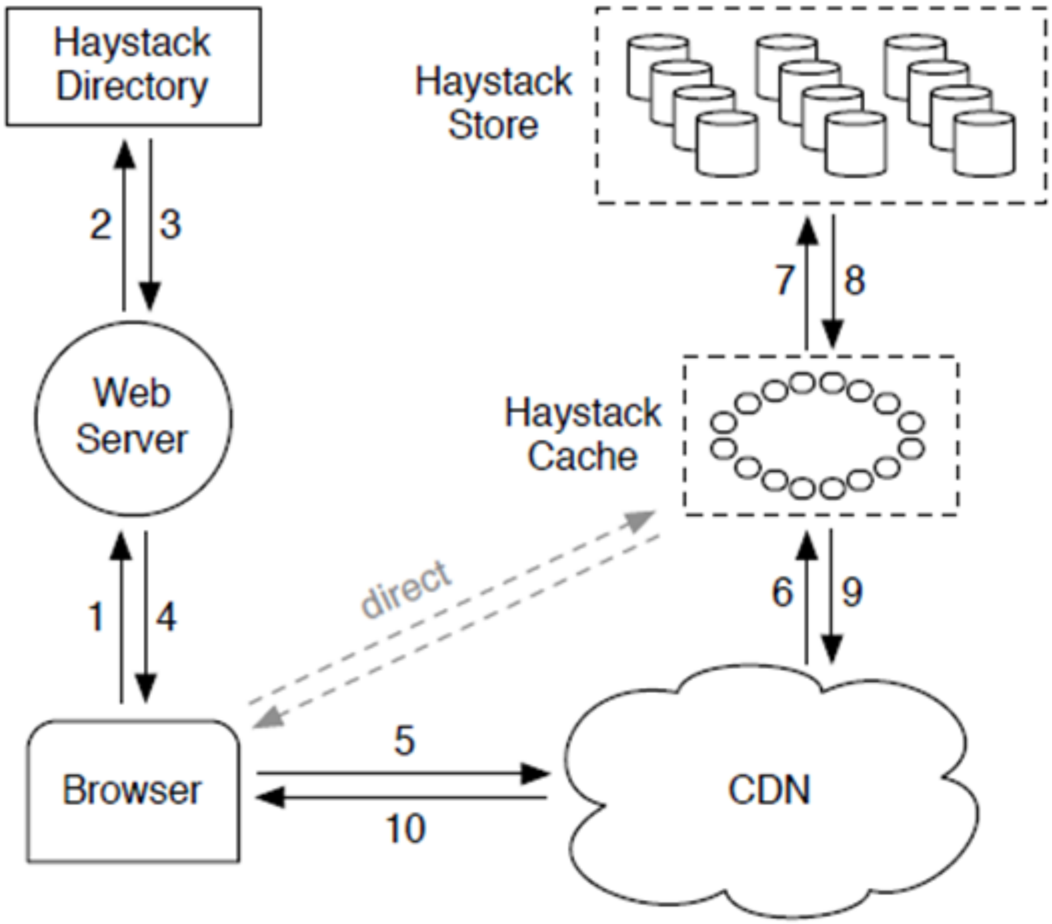
Haystack's Goals

- High throughput and low latency
  - Have to put up with (very frequent) requests
  - Photos served quickly to facilitate a good user experience
- Fault-tolerant
  - Users should not experience errors despite inevitable server crashes and hard drive failures
  - Photos replicated and brought back quickly
- Cost-effective
  - Cost of terabyte of usable storage
  - Read rate normalized for each terabyte of storage
- Simple
  - Obviously, the simpler, the better!

Design

- Use a CDN to serve popular images
- Leverage Haystack to respond to photo requests in long tail efficiently
  - Store multiple photos in a single file and handle large files efficiently
- 3 Core Components
  - Haystack Store
  - Haystack Directory
  - Haystack Cache

HAYSTACK'S DESIGN



Haystack Directory

- Maintains mappings from logical to physical volumes
- Used for constructing image URLs

http://<CDN>/<Cache>/<Machine ID>/<Logical volume,Photo>

- Balances writes across logical volumes and reads across physical volumes
- Determines whether a photo request should be handled by the CDN or by the Cache
- Identifies read-only logical volumes
  - Machine is marked read-only when it exhausts its capacity or for operational reasons

Haystack Cache

- Functions as an internal CDN
- A newly retrieved photo is cached iff
  - Request comes directly from a user and not the CDN
    - Post-CDN caching is ineffective
  - Photo is fetched from a write enabled Store machine
    - Shelter write-enabled Store machines photos are most heavily accessed soon after they are uploaded
    - Haystack performs better when doing either reads or writes

Haystack Store

- Encapsulates the storage system for photos
- Organized by physical volumes
  - 10 terabytes of physical storage split into 100 physical volumes 100 gigabytes each
- Physical volumes on different machines grouped into logical volumes
  - A photo saved to a logical volume is written to all corresponding physical volumes
- Performs basic operations
  - Read
  - Write
  - Delete

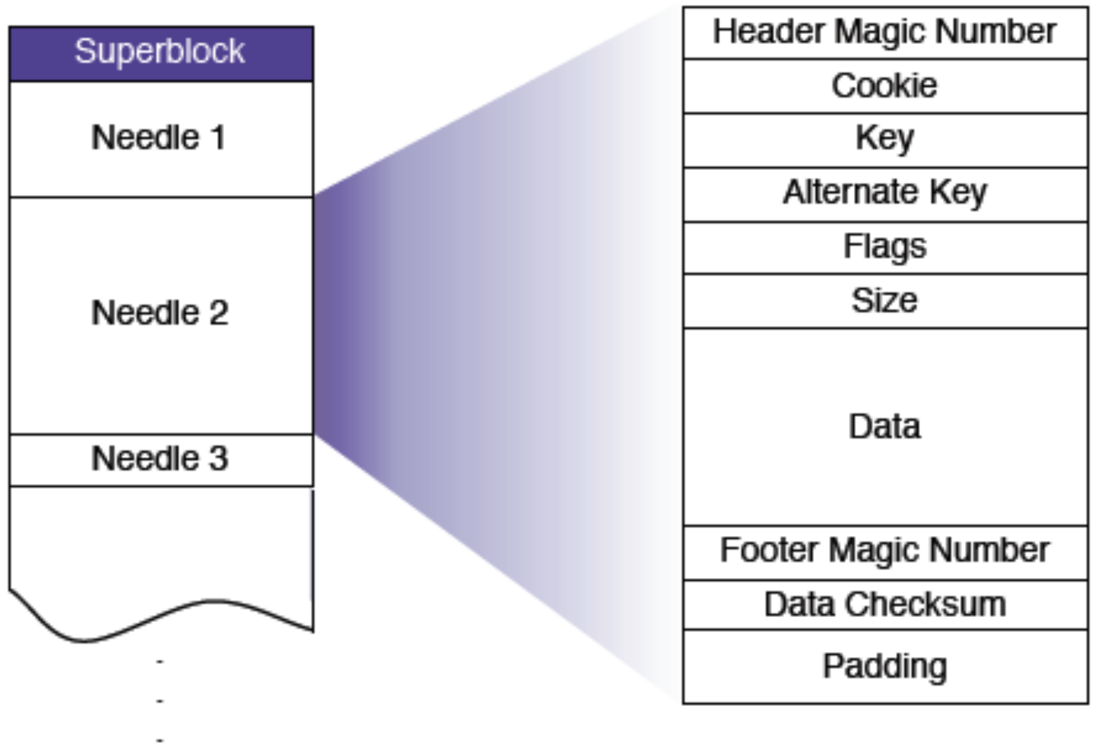
PHYSICAL VOLUME LAYOUT

- Store machine represents a physical volume as a large file consisting of a superblock followed by a sequence of needles
  - Think of a physical volume as a very large file (100 GB) saved as '**/hay/haystack <logical volume id>**'

- Each needle represents a photo stored in Haystack
  - Uniquely identified by

    <Offset, Key, Alternate Key, Cookie>

LAYOUT OF HAYSTACK STORE FILE



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

Photo Read

- Cache machine requests a photo it supplies the logical volume id, key, alternate key, and cookie
  - Cookie's value is randomly assigned by and stored in the Directory at the time that the photo is uploaded
  - Used to eliminates attacks aimed at guessing valid URLs for photos
- Store machine looks up the relevant metadata in its in-memory mappings.
  - Checks if it is not deleted
  - Seeks to the appropriate offset in the volume file
  - Reads the entire needle from disk
  - Verifies the cookie and the integrity of the data
  - Returns the photo if checks passed

Photo Write

- Haystack web servers provide:
  - Logical volume id, key, alternate key, cookie, and data to Store machines
- Each machine synchronously appends needle images to its physical volume files and updates in-memory mappings as needed
- Volumes are append-only so photos can only be modified by adding an updated needle with the same key and alternate key
  - Different logical volume: the Directory updates its application metadata and future requests will never fetch the older version
  - Same logical volume: duplicated distinguished based on their offsets: highest offset =latest version

Uploading a Photo

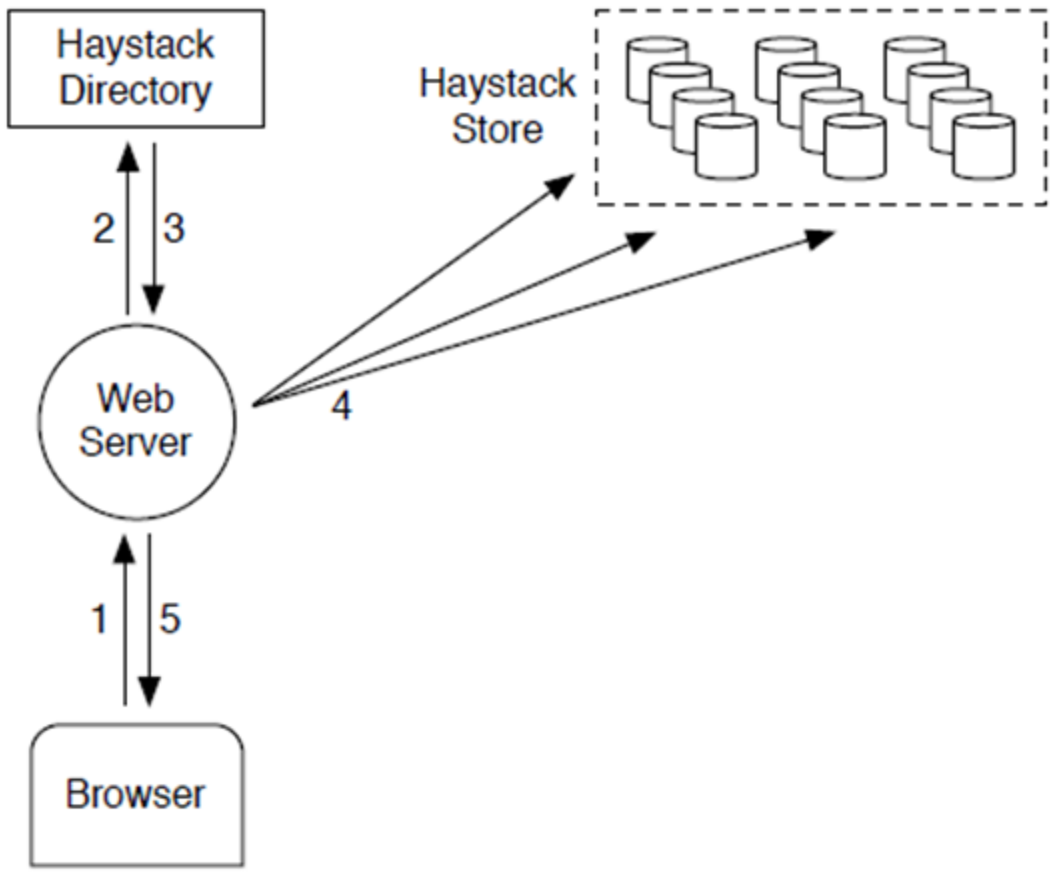


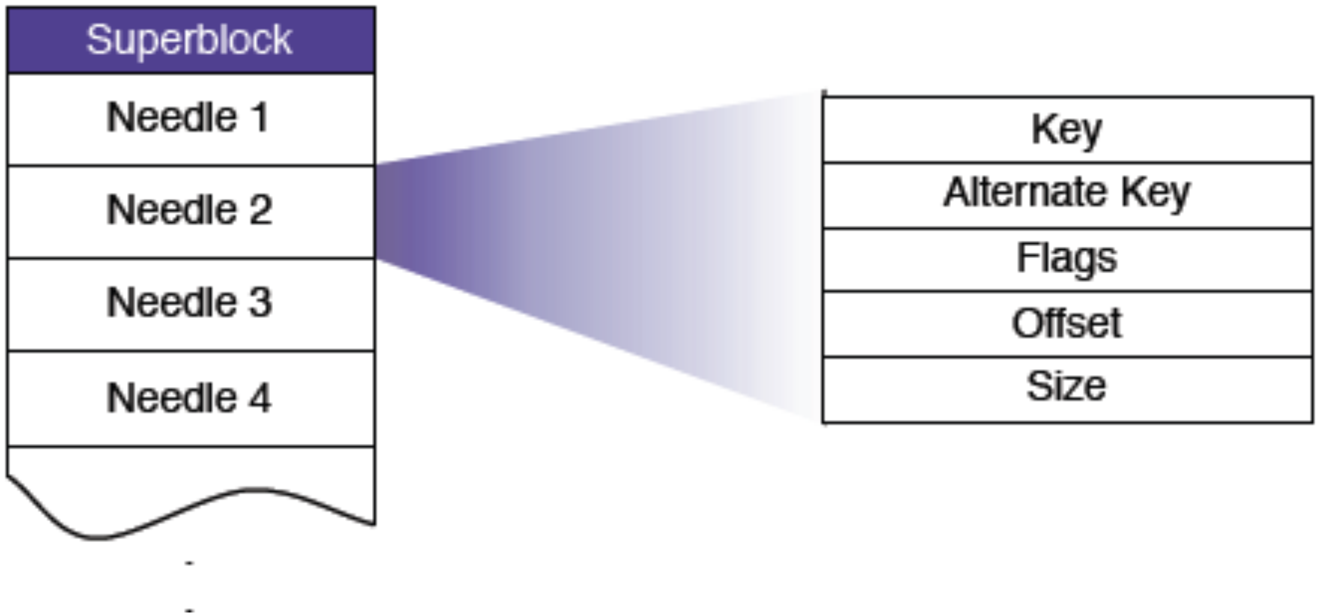
Photo Delete

- Very straightforward
  - Sets the delete flag in both the in-memory mapping and synchronously in the volume file
- Space occupied by deleted needles is lost for some time and reclaimed later via compaction
  - Online operation that reclaims the space used by deleted and duplicate needles
  - Needles are copied into a new file and the new file replaced the current file
- The pattern for deletes is similar to photo views
  - Young photos are a lot more likely to be deleted
  - ~25% of the photos get deleted / yr

Index File

- Store machines maintain an index file for each of their volumes
- Checkpoint of the inmemory data structures used to locate needles efficiently on disk
- Used to quickly reconstruct in-memory mappings shortening restart time
- Index is usually less than 1% the size of the store file

Layout of Haystack Index File



Field	Explanation
Key	64-bit key
Alternate key	32-bit alternate key
Flags	Currently unused
Offset	Needle offset in the Haystack Store
Size	Needle data size

RESULTS

- The point was to store metadata in memory but before Haystack it was too costly
- Haystack overhead
  - Average 10 bytes of main memory per photo
  - Each photo is scaled to four photos with the same key (64 bits), different alternate keys (32 bits), and different data sizes (16 bits).
  - In addition, 2 bytes per image in overheads due to hash tables, bringing the total for four scaled photos of the same image to **40 bytes**
- For comparison, xfs inode t structure in Linux is 536 bytes

Results Cont.

- Significantly less disk operations
  - At most one per photo
- Simplified metadata
  - Less costly lookups
  - Easily cachable
  - 1MB of metadata for every 1GB of usable storage
  - 10TB per node results in 10GB metadata
- Cost per terabyte of usable storage:
  - Haystack costs 28% less
- Read rate normalized for each terabyte of usable storage
  - Processes 4x more reads per second than an equivalent terabyte on a NAS appliance

Daily Photo Traffic

Operations	**Daily Counts**
Photos Uploaded	~120 Million
Haystack Photos Written	~1.44 Billion
Photos Viewed	80-100 Billion
[*Thumbnails*]	10.2 %
[*Small*]	84.4 %
[*Medium*]	0.2 %
[*Large*]	5.2 %
Haystack Photos Read	10 Billion

EVALUATION (STORE)

Benchmark	[Config # Operations]	Reads			Writes		
		Throughput (in images/s)	Latency (in ms)		Throughput (in images/s)	Latency (in ms)	
			Avg.	Std. dev.		Avg.	Std. dev.
Random IO	[Only Reads]	902.3	33.2	26.8	–	–	–
Haystress	[**A** # Only Reads]	770.6	38.9	30.2	–	–	–
Haystress	[**B** # Only Reads]	877.8	34.2	28.1	–	–	–
Haystress	[**C** # Only Multi-Writes]	–	–	–	6099.4	4.9	16.0
Haystress	[**D** # Only Multi-Writes]	–	–	–	7899.7	15.2	15.3
Haystress	[**E** # Only Multi-Writes]	–	–	–	10843.8	43.9	16.3
Haystress	[**F** # Reads & Multi-Writes]	718.1	41.6	31.6	232.0	11.9	6.3
Haystress	[**G** # Reads & Multi-Writes]	692.8	42.8	33.7	440.0	11.9	6.9

Table 4: Throughput and latency of read and multi-write operations on synthetic workloads. Config **B** uses a mix of 8KB and 64KB images. Remaining configs use 64KB images.

- Two benchmarks: Randomio (external) and Haystress (custom built)
- Haystack delivers 85% of the raw throughput of the device while incurring only 17% higher latency (workload A: rnd read of 64KB)
- Multi-writes of 4 and 16 writes improves throughput by 30% and 70% respectivly

EVALUATION (STORE)

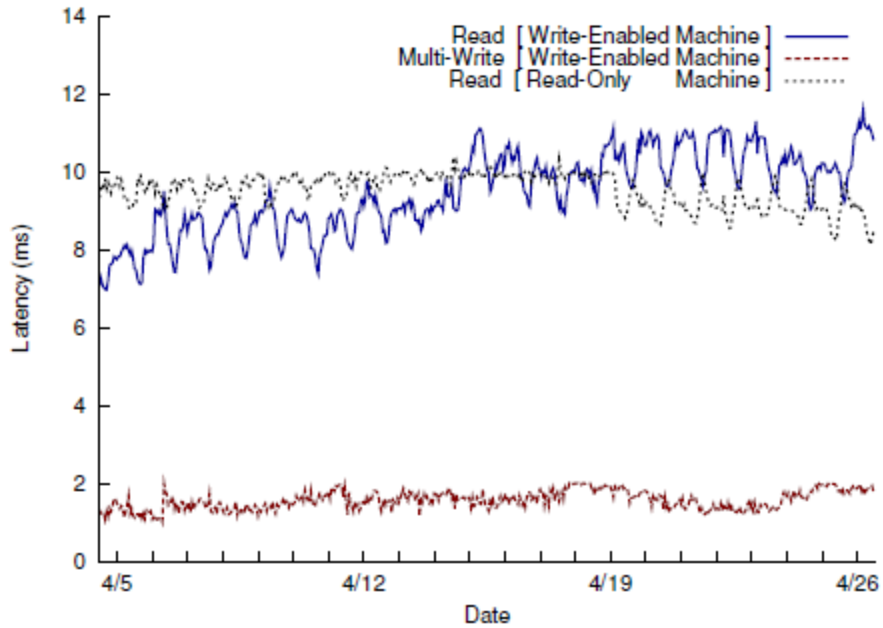


Figure 11: Average latency of Read and Multi-write operations on the two Haystack Store machines in Figure 10 over the same 3 week period.

- Multi-write latency fairly low (1 and 2 ms) and stable (variable traffic)
- Reads on a read-only box latency fairly stable;
- Write-enable: higher latency

EVALUATION (DIRECTORY)

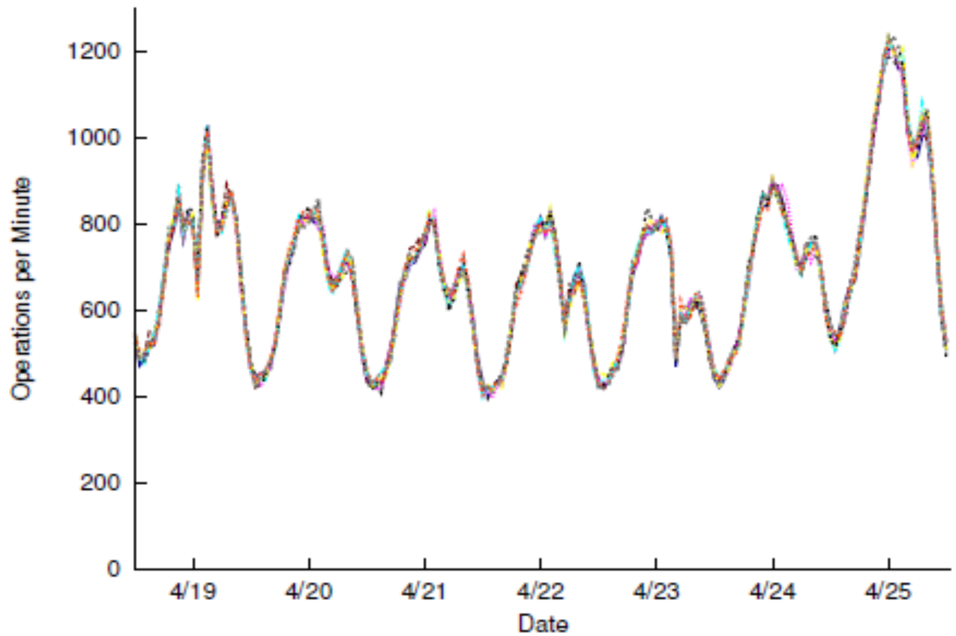


Figure 8: Volume of multi-write operations sent to 9 different write-enabled Haystack Store machines. The graph has 9 different lines that closely overlap each other.

Directory balances (very effectively) reads and writes across Stores

EVALUATION (CACHE)

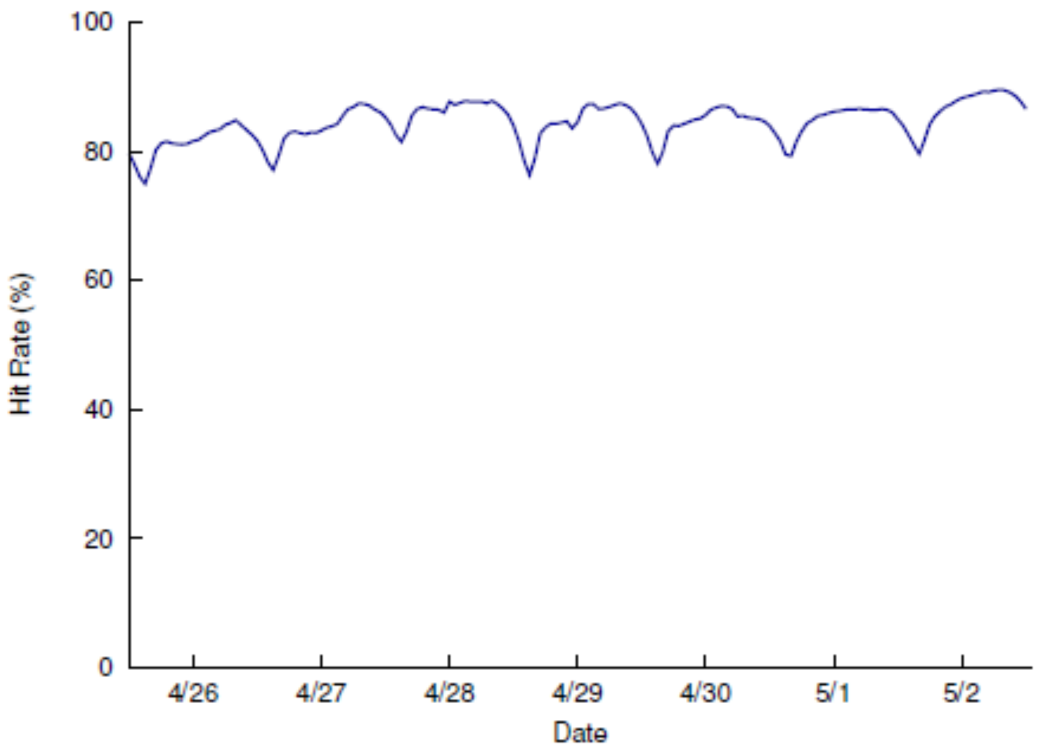


Figure 9: Cache hit rate for images that might be potentially stored in the Haystack Cache.

Notice the high hit rate: ~80%. Why?

Q&A!

Thank you!