Lecture 9: Facebook Photo Storage

CSCI4180 (Fall 2013)

Patrick P. C. Lee

Most slides are adapted from Dr. T. Y. Wong's lecture in 2012 spring

Outline

- Haystack: an efficient storage of billion of photos.
  - http://static.usenix.org/event/osdi10/tech/full_papers/Beaver.pdf

- Other Facebook attempts in building storage.
  - Facebook Messages.
    - Using HBase & Haystack.
  - Facebook Insight.
    - Using HBase for the storing results.

Facebook Photo Storage

	April 2009	October 2010
Total	15 billion photos 60 billion images 1.5 petabytes (1.5 x 10^{15})	65 billion photos 260 billion images 20 petabytes (20 x 10^{15})
Upload Rate	220 million photos per week (25 terabytes per week)	1 billion photos per week (60 terabytes per week)
Serving Rate	550,000 images /sec	1 million images / sec

For each photo, four versions (called **images**) are stored and they are: **large**, **medium**, **small**, and **thumbnail**

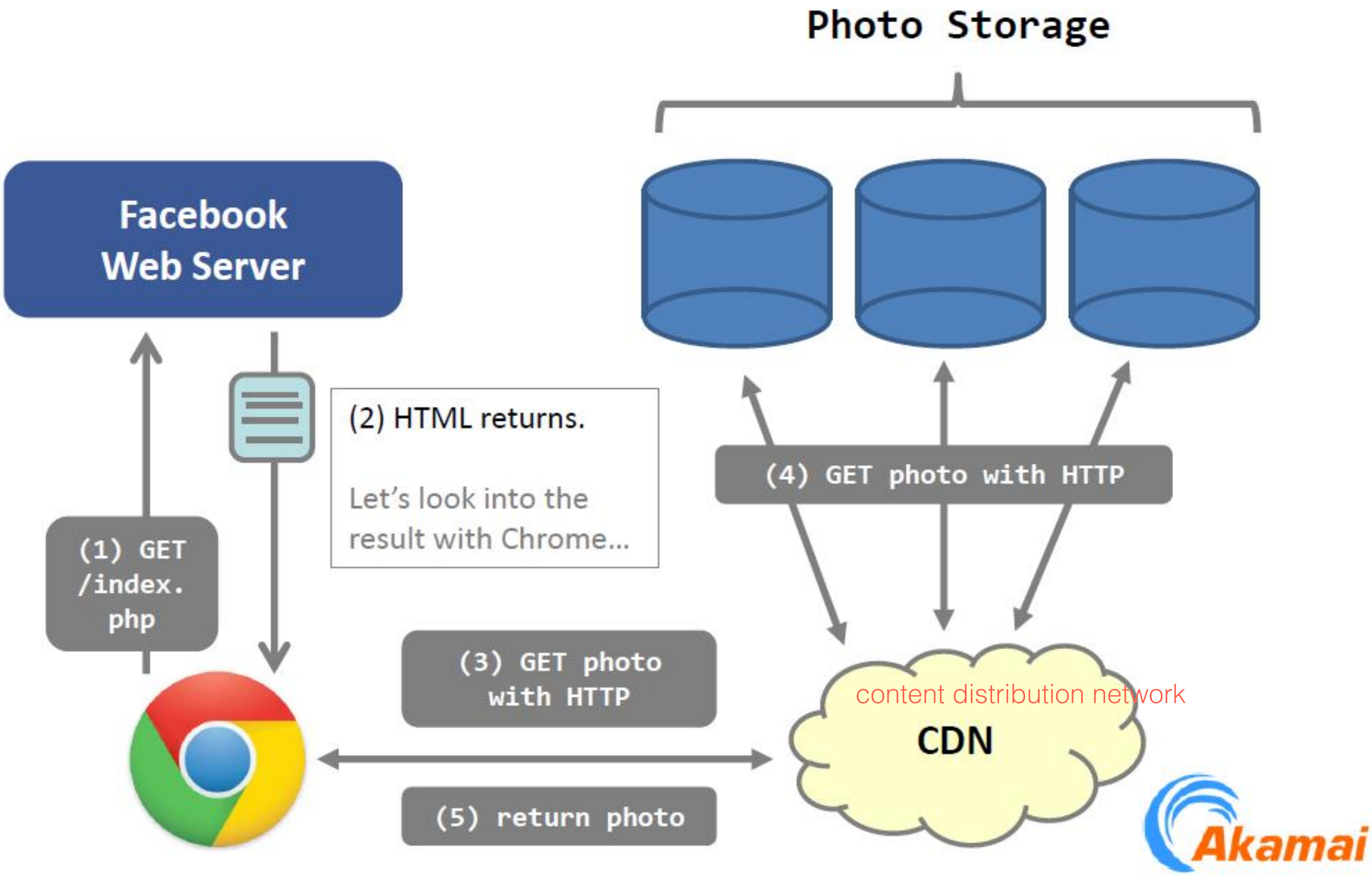
Challenges

- Looking up a single file from a huge set of files… Similar to finding a needle in haystack
- Not a problem, if you understood FS design (from CSCI3150) not scalable
  - You need to know the pathname, i.e., the key.
  - The kernel digs out the metadata.
  - Size, block allocations, etc.
  - Return the file content
- Performance? That's the question!

Design Goals

- High throughput and latency
  - Minimize disk I/Os
- Fault tolerant
  - Replicate photos in geographically distinct locations
- Cost effective
  - Cost per terabyte of usage must be kept low
- Simple
  - Straightforward to implement and maintain

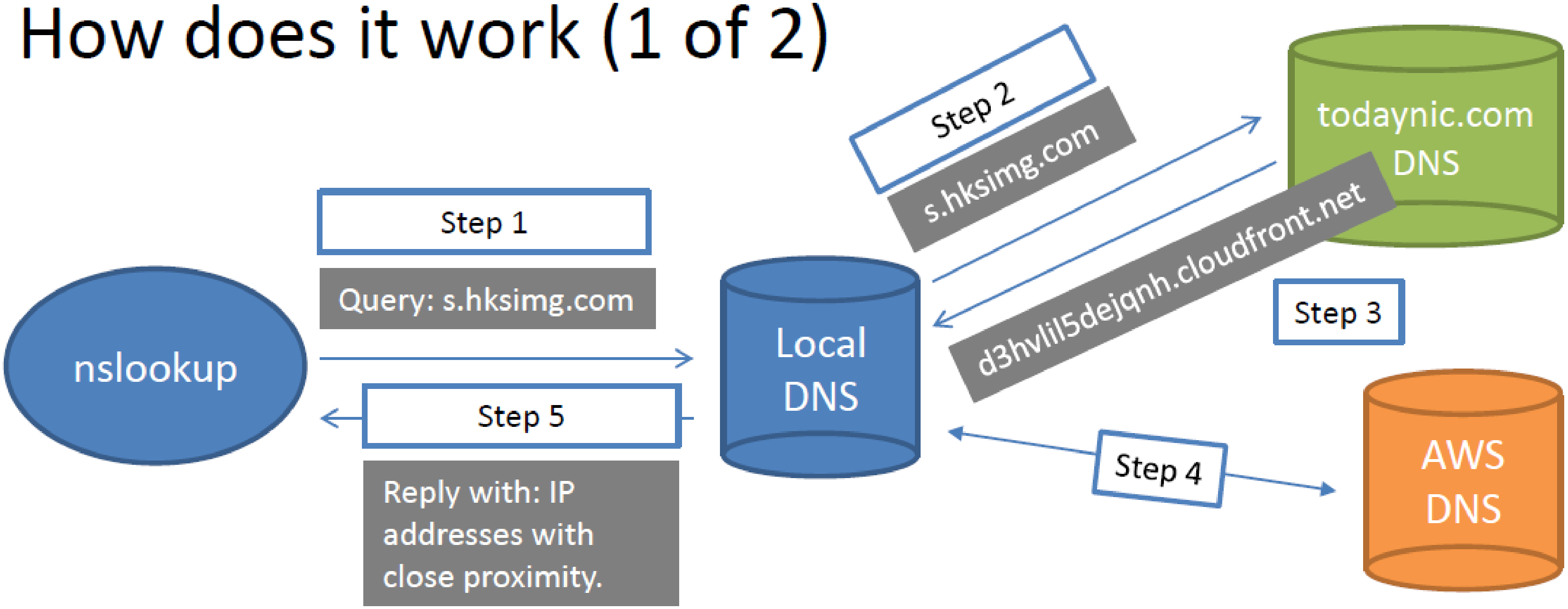
Typical Design



Typical Design

- CDN – **content delivery network** – is a network of nodes that **cache contents**.

- How does it work (1 of 2)

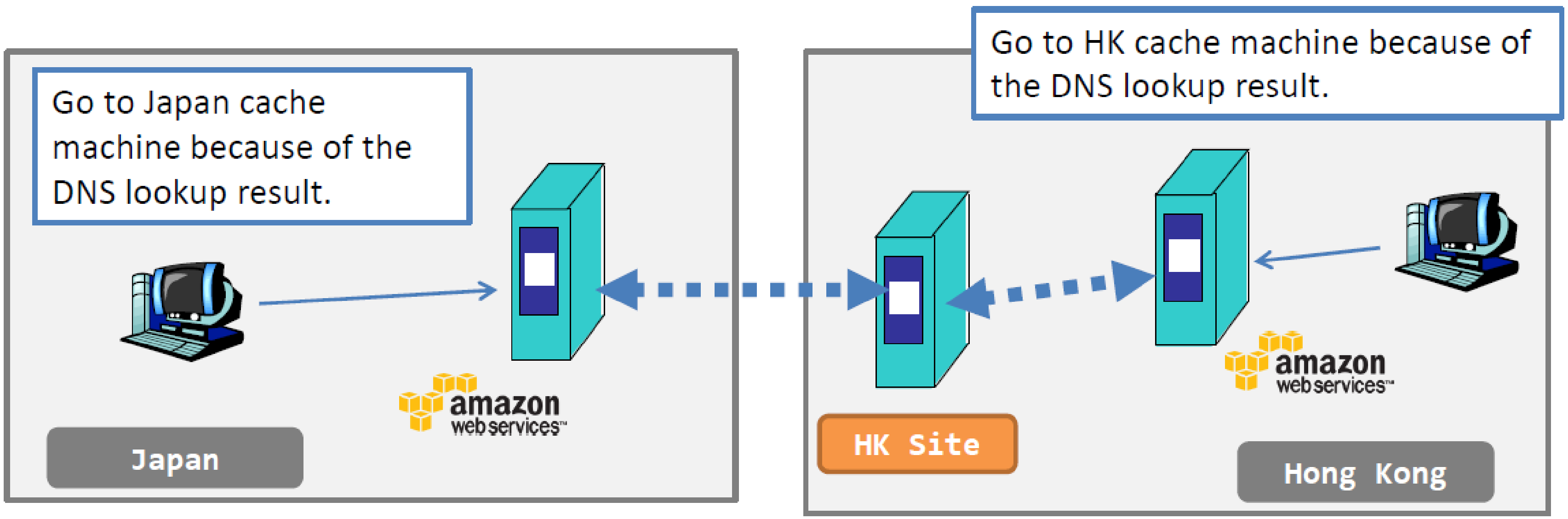


Example: CloudFront, http://aws.amazon.com/cloudfront/

Typical Design

- ➢ CDN – **content delivery network** – is a network of nodes that **cache contents**.

- How does it work? (2 of 2)



CDN illustration: http://www.youtube.com/watch?v=jnUFXXWECQw

Typical Design

- CDN serves a large-scale cache
- Pros and Cons of CDN:
  - Pros: world-wide scale deployment, fast response time
  - Cons: control expiry of content

- Yet, CDN fits well typical web sites:
  - People access up-to-date data most of the time!
  - Old contents are seldom read.

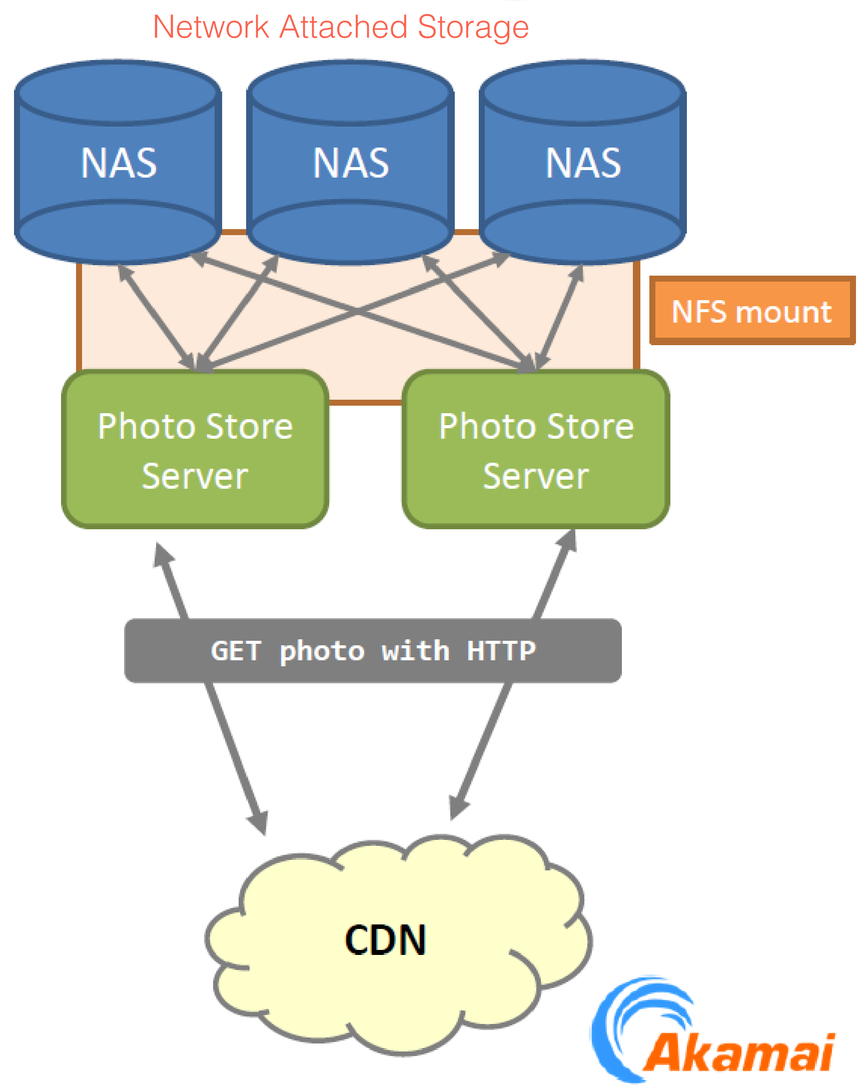
Typical Design: Limitation

- Facebook (or social networking sites in general) has a very different access pattern from web sites
  - Old contents are frequently visited
  - Think about how you view your friend’s album

- **Long tail** behavior
  - Requests from the long tail (or very old content) account of a significant amount of time

(Old) NFS-based Design

- The old photo infrastructure had several tiers:
  - Upload tier receives users' photo uploads, scales the original images and saves them on the NFS storage tier.
  - Photo serving tier processes HTTP requests for images
  - NFS storage tier built on top of commercial storage appliances.



(Old) NFS-based Design

- An enormous amount of metadata is generated on the storage tier due to the namespace directories and file inodes.

- The amount of metadata far exceeds the caching abilities of the NFS storage tier
  - multiple disk I/Os per photo per upload/read request

- High degree of reliance on CDNs = expensive

(Old) NFS-based Design

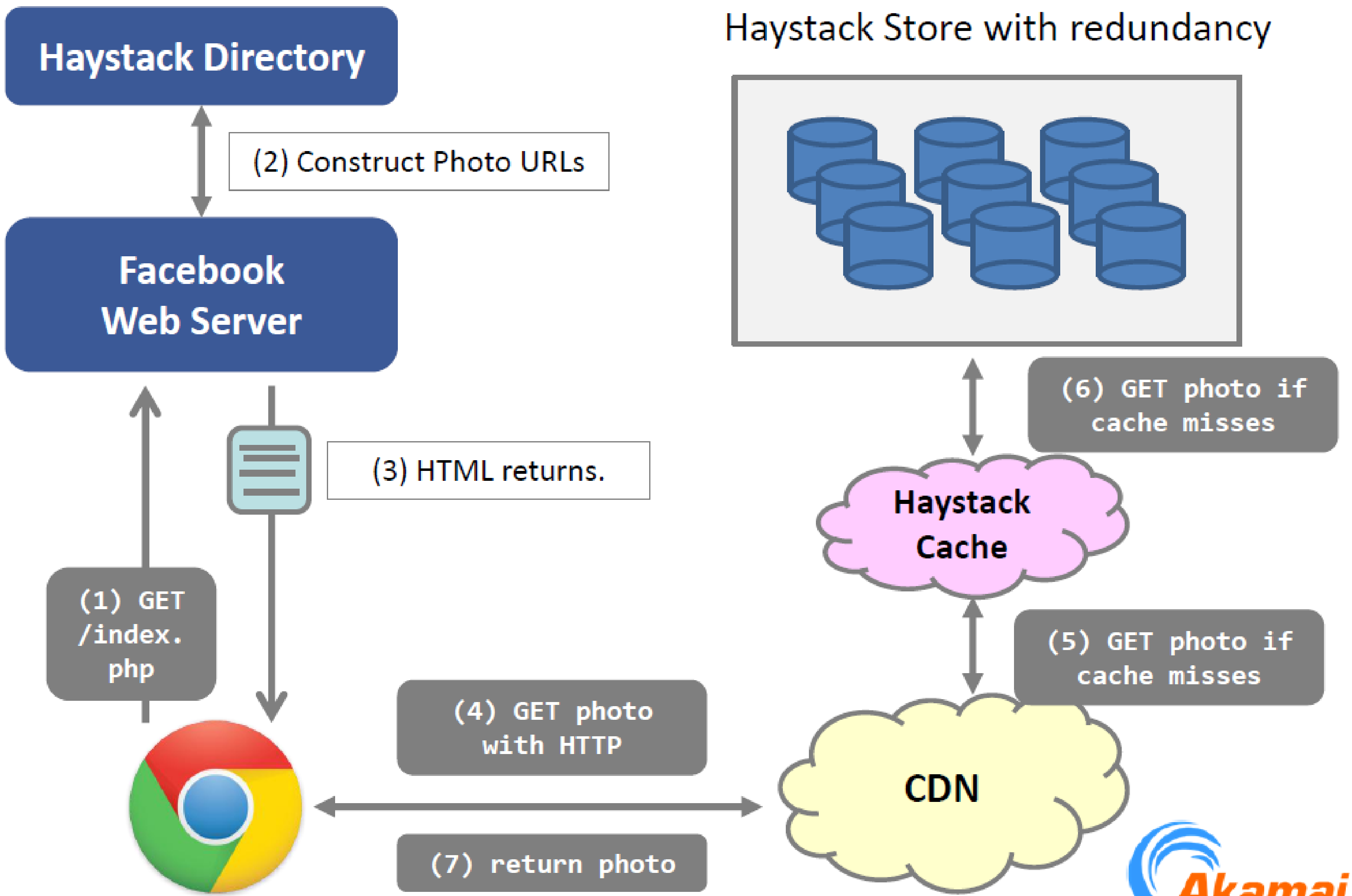
- With optimizations, we still need at least 3 I/O operations for each photo request:
  - Read the directory metadata into memory
  - Load inode into memory
  - Read file contents

  too much I/O (4 versions of images for 1 photo), need to reduce the no. of I/O to save time

- Facebook's goal: further reduce I/Os

- Question: what is the minimum I/Os? 1 : read the file content

Haystack Architecture



Haystack: Key Components

- Develop customized file system with lightweight metadata operations
  - Haystack directory and Haystack store
- Offload CDN with a customized cache
  - Haystack cache

User Operations

- User visits a page
  - Web server forwards the request to the Directory
  - The Directory constructs URL for each photo
    - http://**<CDN>**/**<Cache>**/**<Machine id>**/**<Logical volume, Photo>**
  - CDN looks up the photo internally
    - If CDN lookup fails, the CDN strips the <CDN> part from the URL and contacts the Cache
    - If Cache lookup fails, the Cache strips the <Cache> part from the URL and contacts the Store

Haystack Directory

- Four main functions:
  - Provides a mapping from logical volumes to physical volumes
  - Load balances writes across logical volumes
  - Determines whether a photo request should be handled by the CDN or by the Haystack Cache
  - Identifies logical volumes that are read-only
    - operational reasons
    - reached storage capacity
- Serves like the namenode in MapReduce

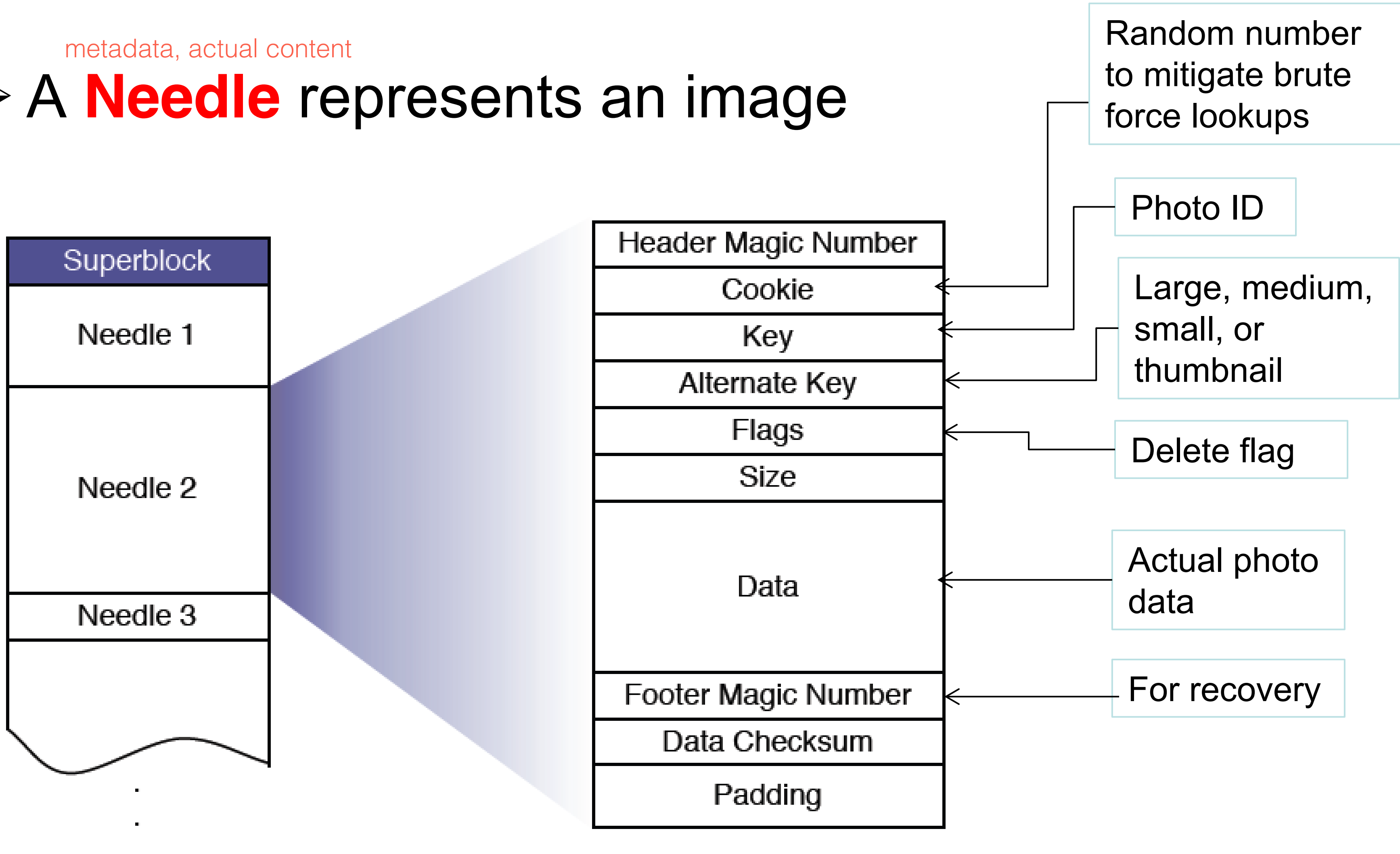
Haystack Cache

- Distributed hash table, uses photo’s id to locate cached data
- Receives HTTP requests from CDNs and browsers
  - If photo is in Cache, return the photo
  - If photo is not in Cache, fetches photo from the Store and returns the photo
- Add a photo to Cache if two conditions are met:
  - The request comes directly from a browser, not the CDN
  - The photo is fetched from a write-enabled Store machine

Haystack Store

metadata, actual content

- A **Needle** represents an image



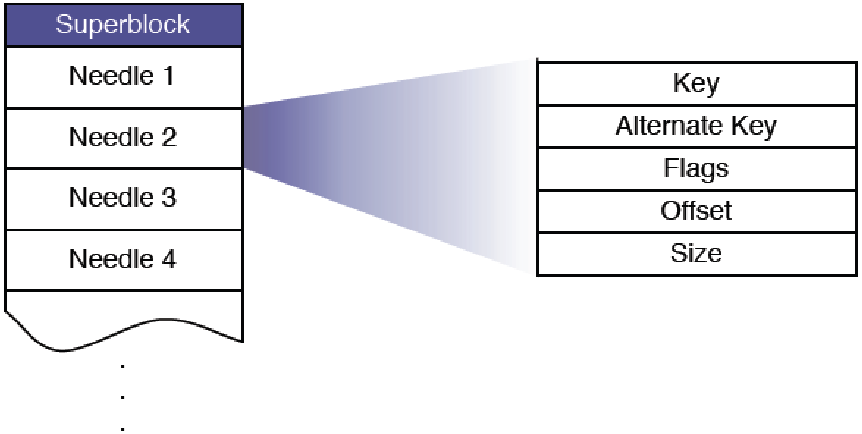
Layout of Haystack Store file

Haystack Store

- Haystack store file
  - A superblock file accompanied by many needles
  - Log-structured format: append-only

    to reduce the seek

- Haystack index file
  - provides minimal metadata required to locate a particular needle in the store
  - Same order as store file
  - A subset of store file, but without photo data
  - For disaster recovery
  - Less than 1% of store file



Layout of Haystack index file

Haystack Store

- Each Store machine manages multiple physical volumes
- Can access a photo quickly using only:
  - the id of the corresponding logical volume
  - the file offset of the photo
- Handles three types of requests:
  - Read
  - Write
  - Delete

Haystack Store: Read

- Each store machine maintains an in-memory index
- If both CDN and Haystack cache miss the photo:
  - Look up the in-memory index **[0 I/O]**
  - The offset in the Haystack store file is found.
  - Assume that the **store file is always opened**.
  - Seek and read **[1 I/O]**
    - Seek takes 1 I/O.
    - The succeeding read does not take any I/O ops since the seek operation has already position the disk head.
    - Since the photo is **sequentially written**, no further I/O ops incurred.
- •Totally, **1 I/O**

Haystack Store: Write

- A multi-write operation: to three replicas
- Append 4 images to the haystack store file.
  - Seek to the end of file and sequentially write **[1 I/O]**.
- Then, append 4 images to the haystack index file.
  - Seek and then sequentially write **[1 I/O]**.
- Update the in-memory index **[0 I/O]**.

Haystack Store: Delete

- A multi-delete operation: to three replicas
- First, remove the files in the in-memory index.
  - This gives an illusion of fast file deletion.
  - Because any queries to the delete file would cause the in-memory index returning errors.
- Then, mark the 4 images of file deleted.
  - 4 seek-and-write.
- Garbage collection?
  - Create a copy of the store file
  - Skip deleted photos while copying
  - 25% of photos get deleted in a given year

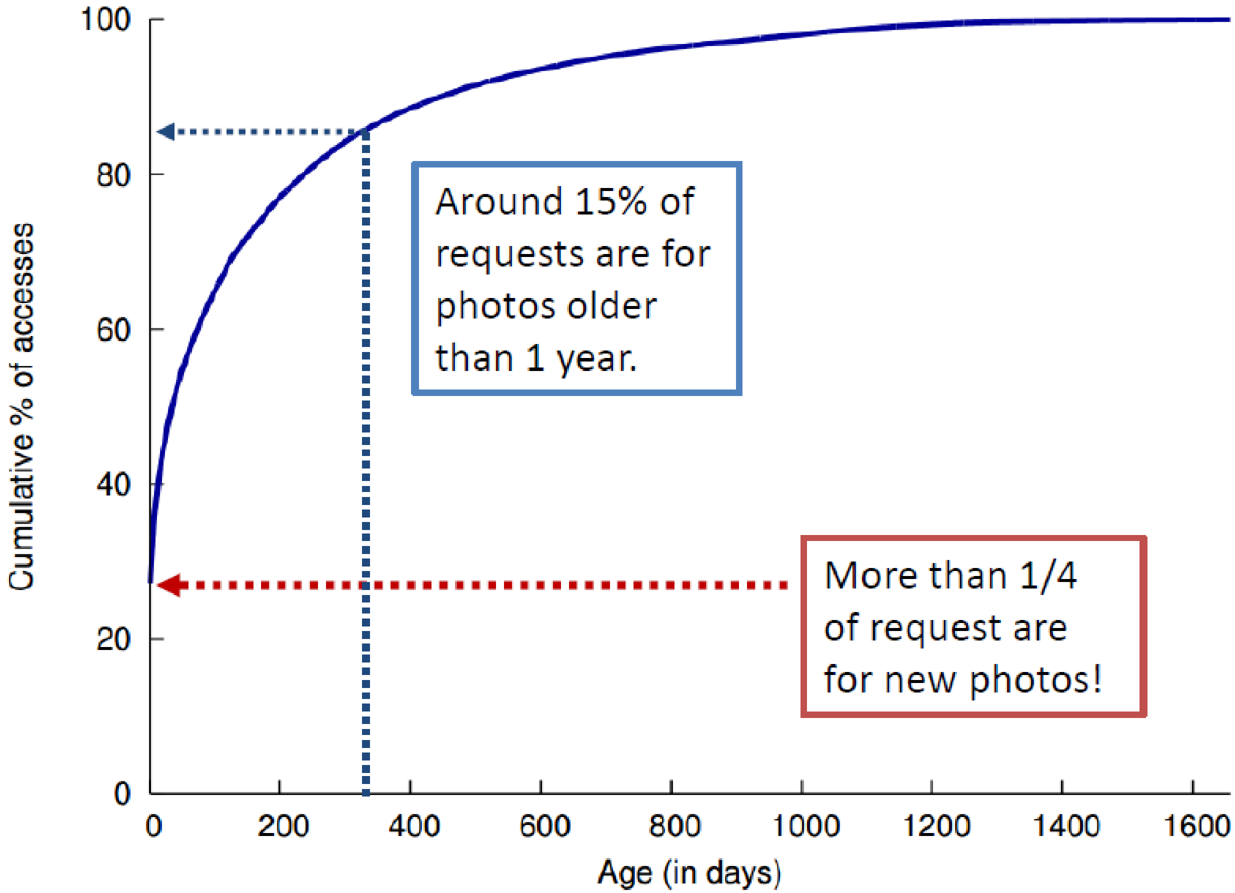
Experiments

- Commodity machines running Haystack.
  - 2 x Quad-core CPUs.
  - 16GB – 32GB memory.
  - Hardware RAID controller.
  - 12+ 1TB SATA HDD

- Your machine can also be a Haystack

Popularity

- Long tail behavior



Cache Hit Rate

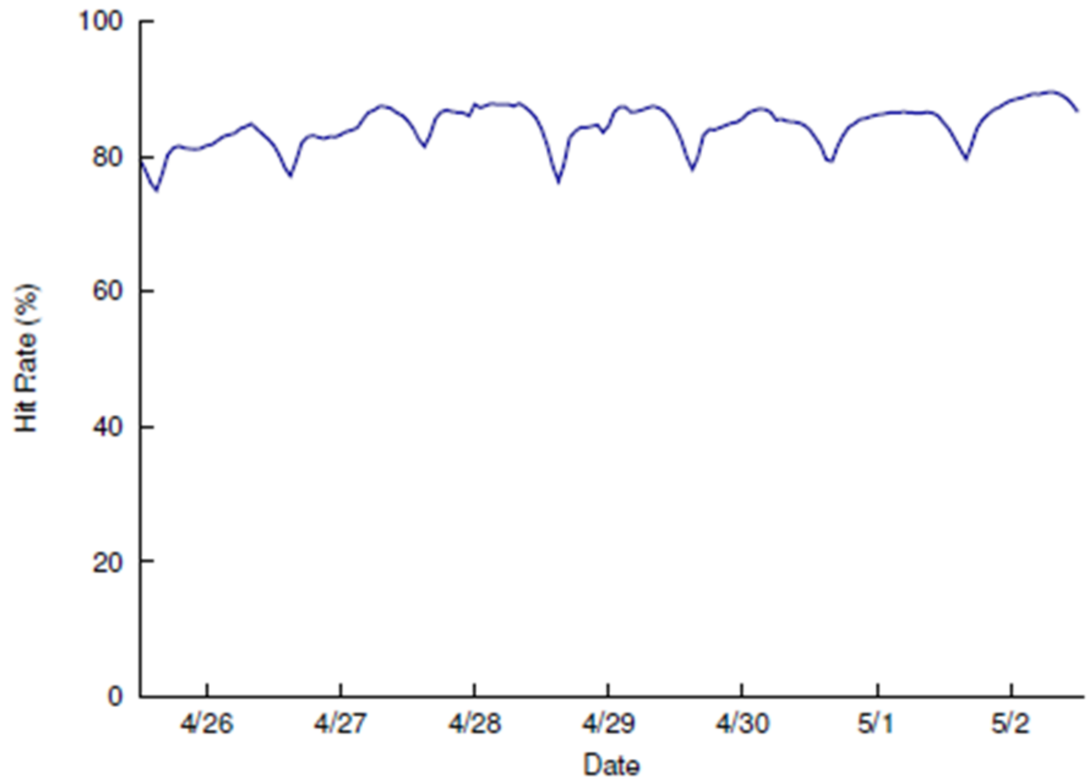


Figure 9: Cache hit rate for images that might be potentially stored in the Haystack Cache.

Volume of Daily Total Traffic

4 different versions and each image is replicated to 3 servers.

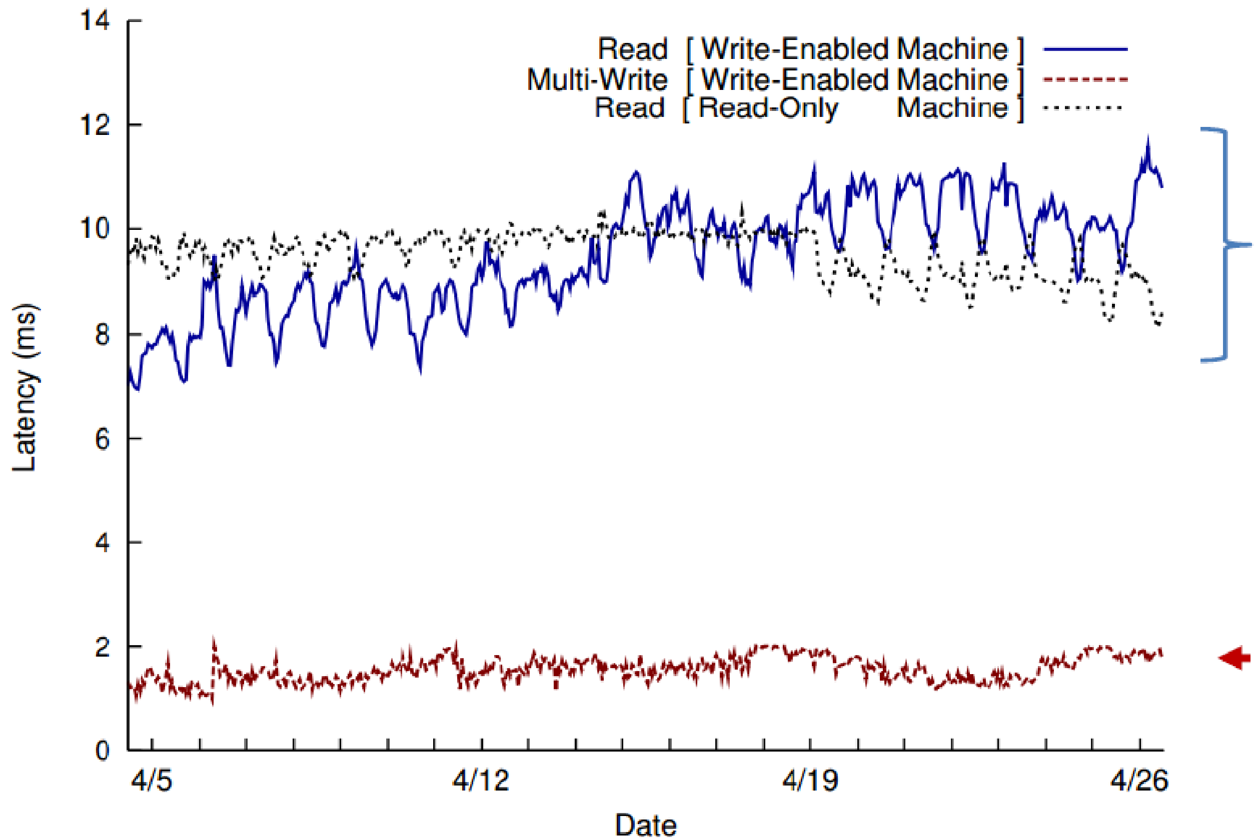
So, this matches Facebook's goal to build a custom system that saves io ops for small files.

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

Read is at least 700 times more than write.

Small photos in News feed!

Duration of Workload: 3 weeks



A stable range between 7 ms and 12 ms.

Lightening fast!
Because of the asynchronous write performed by the RAID controller.

Haystack: Key Contributions

- Reduced disk I/O
  - 10 TB/node -> 10 GB of metadata
    - ~10 bytes per image, ~40 bytes per photo
    - This amount is easily cacheable!
- Simplified metadata
  - No directory structures/file names
    - 64-bit ID
  - Results in easier lookups
- Single photo serving and storage layer
  - Direct I/O path between client and storage
  - Results in higher bandwidth

Haystack: Conclusions

- This is an efficient storage of billions of photos.
- What can we learn from Haystack?
  - Understand your problem!
    - Facebook problem is the haunting I/O requests over small photos.
  - Keep your solution simple.
    - Everybody who passed CSCI3150 will understand the merit of Haystack!

Open Issues

- Is compaction (garbage collection) the best solution? Seems a bit expensive. Better ideas?
- What about album level abstraction?
  - Important/better if photos from the same album are placed sequentially or at least close together?
- Privacy concerns
  - Are cookies sufficient protection? Is there a better way?
  - What about security levels in Facebook? How are they enforced with respect to Haystack?
- How is consistency maintained between the Haystack and the CDN?
- Can we use erasure coding rather than replication?

Inside Facebook

- Facebook messages
  - Use **Apache HBase** stores
    - Small messages,
    - Message metadata, and
    - Search index of the users' messages.
- Facebook photos
  - Use **Haystack** stores:
    - Large messages, and
    - Attachment.
- More: http://www.facebook.com/Engineering