Lecture 10: Facebook Photo Storage

CSCI4180

Patrick P. C. Lee

Outline

- **Haystack**: storage of billion of photos
- **f4**: fault-tolerant warm BLOB storage

- Other Facebook attempts in building storage
  - Facebook Messages
    - Using HBase & Haystack
  - Facebook Insight
    - Using HBase for the storing results
- More: http://www.facebook.com/Engineering

Facebook Photo Storage

	April 2009	**October 2010**
Total	15 billion photos 60 billion images 1.5 petabytes (1.5×10^{15})	65 billion photos 260 billion images 20 petabytes (20×10^{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**

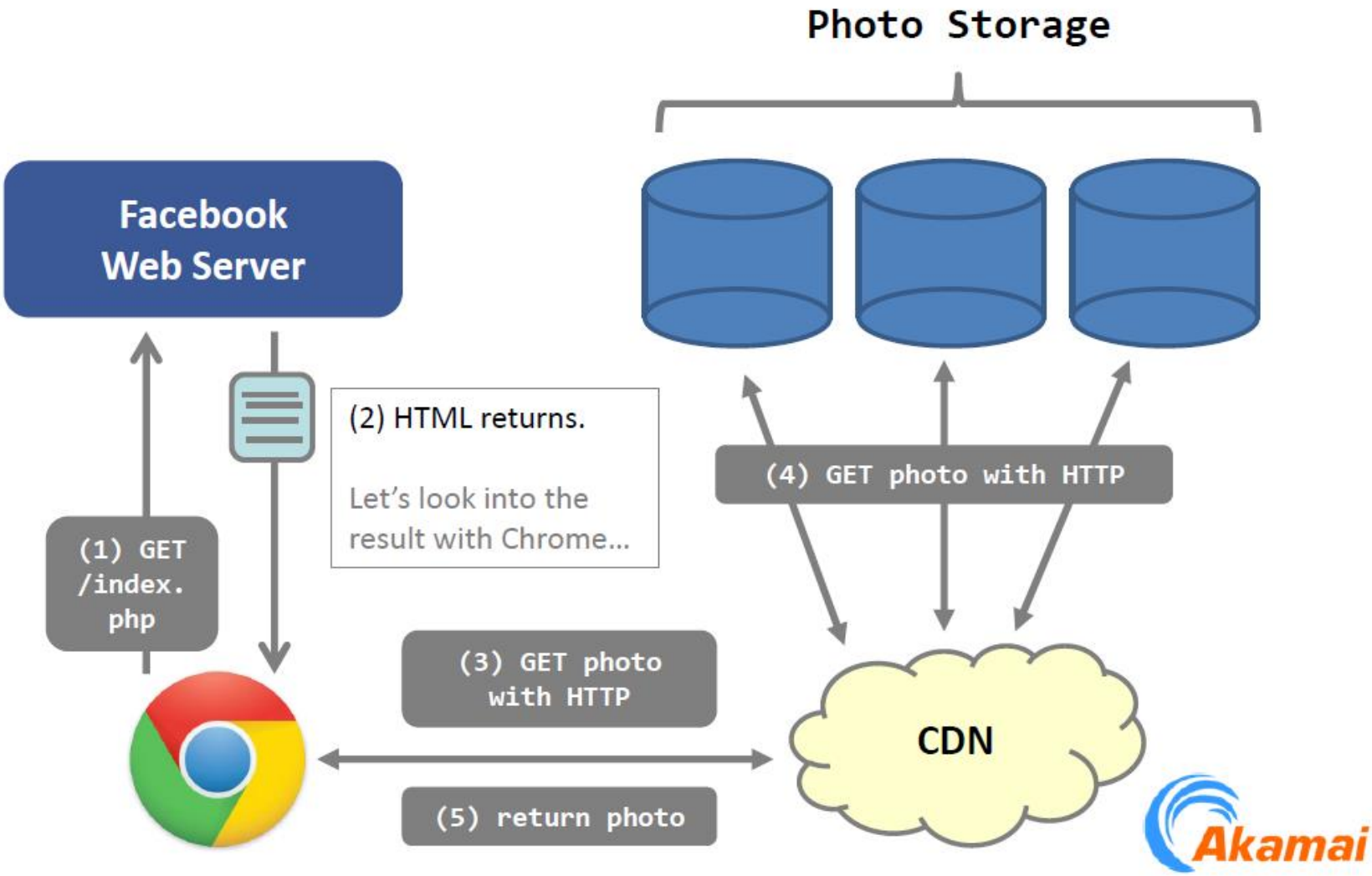
Challenges

- Looking up a single file from a huge set of files… Similar to finding a needle in haystack
- Not a problem, if you understand FS design (from CSCI3150)
  - 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 is critical!

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

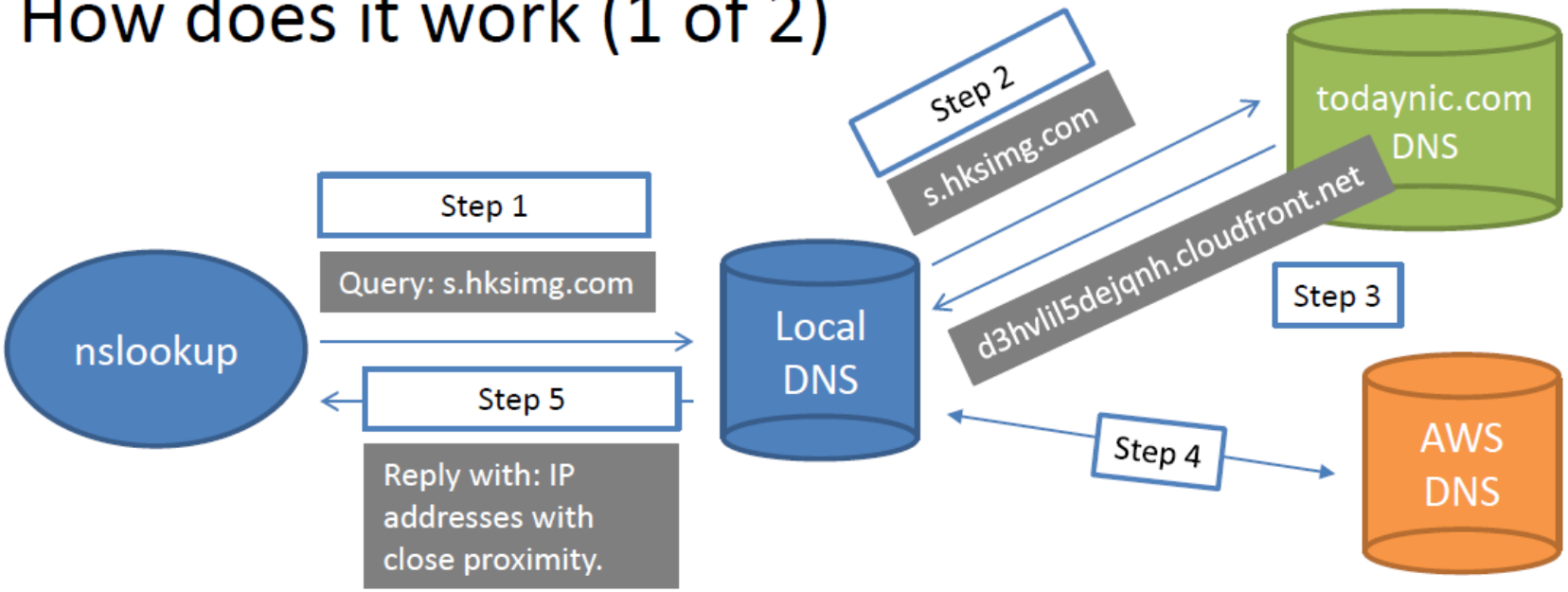
Typical Design



Typical Design

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

- How does it work (1 of 2)

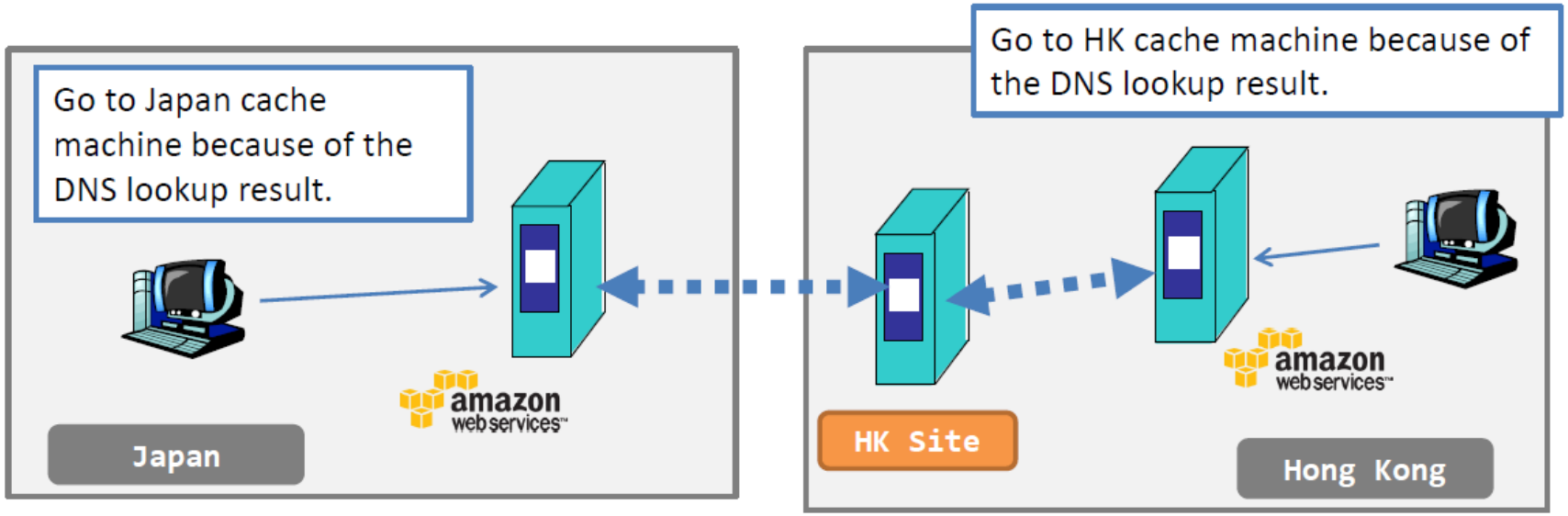


Example: CloudFront, http://aws.amazon.com/cloudfront/

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.

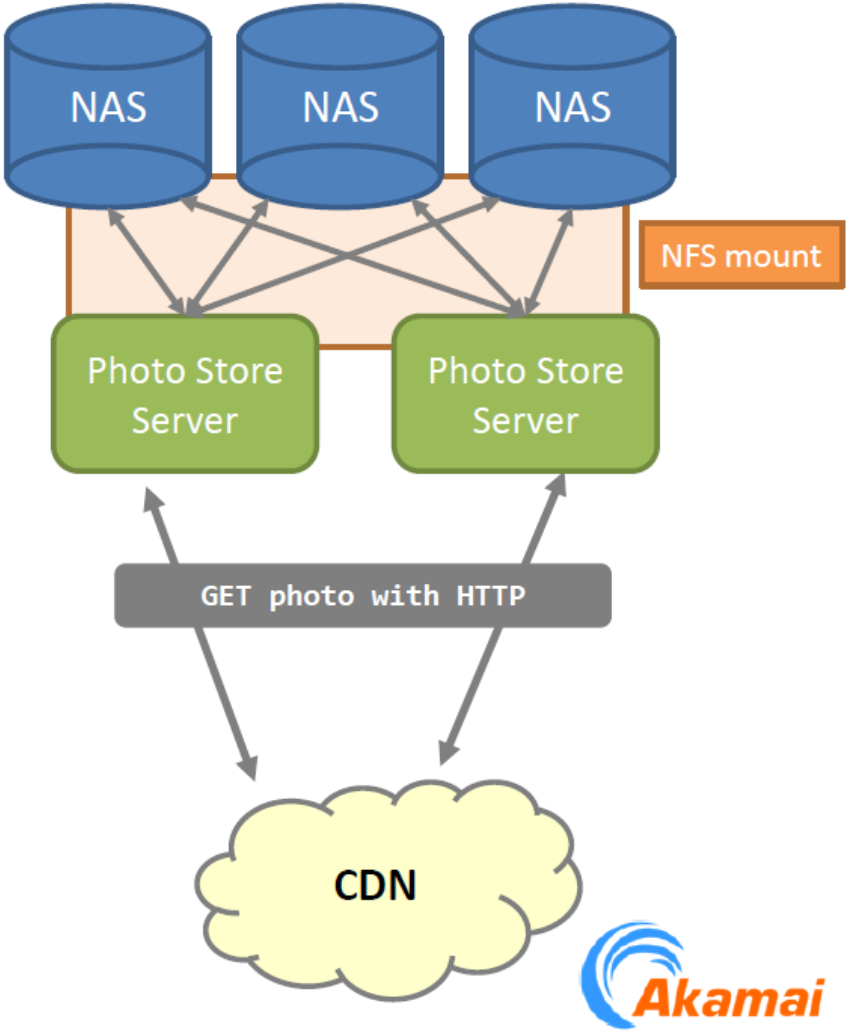
Typical Design: Limitation

- 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.

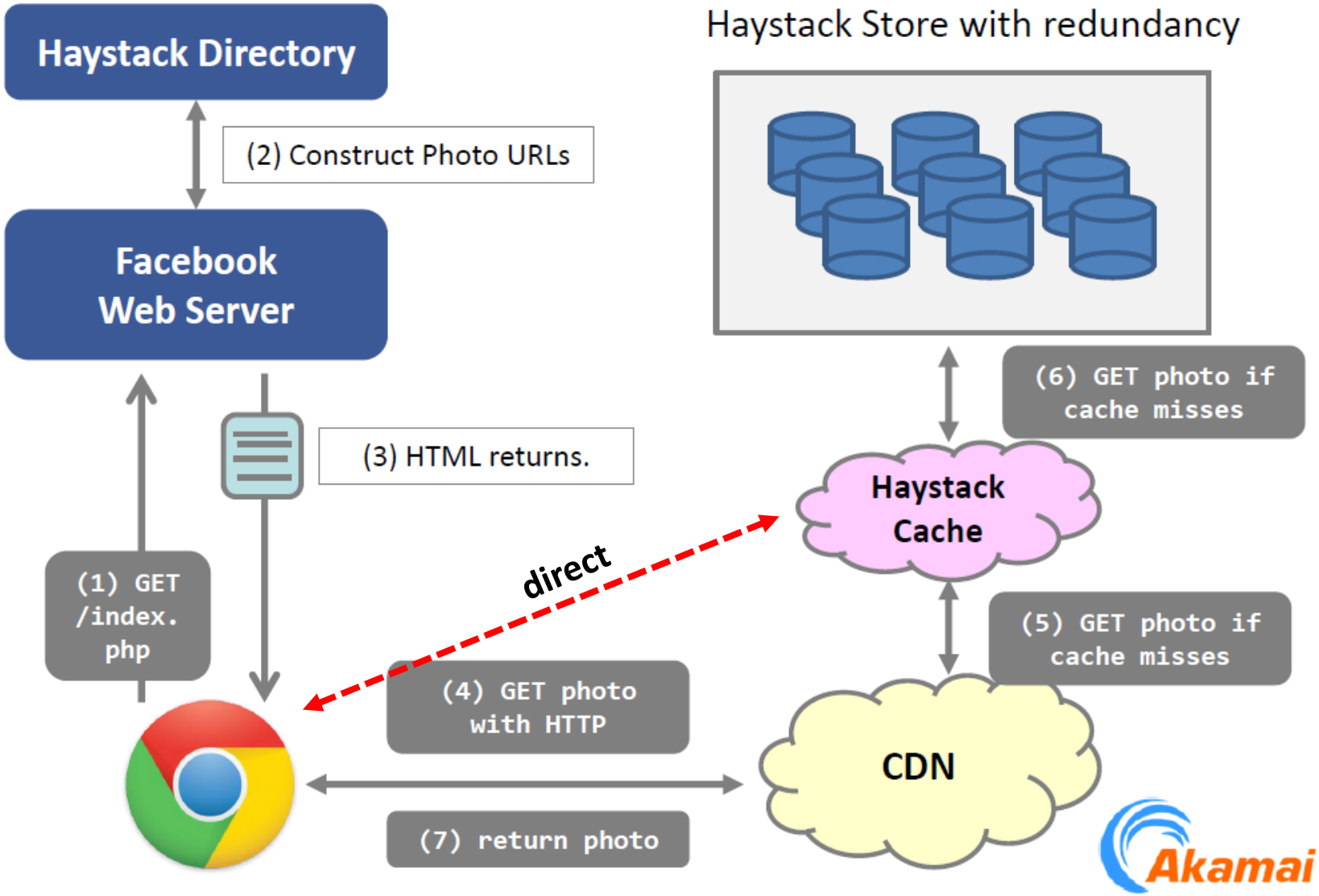
- The amount of metadata far exceeds the caching abilities of the NFS storage tier
  - multiple disk I/Os per photo per upload/read request

- 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
- Facebook's goal: further reduce I/Os
- Question: what is the minimum I/Os?

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

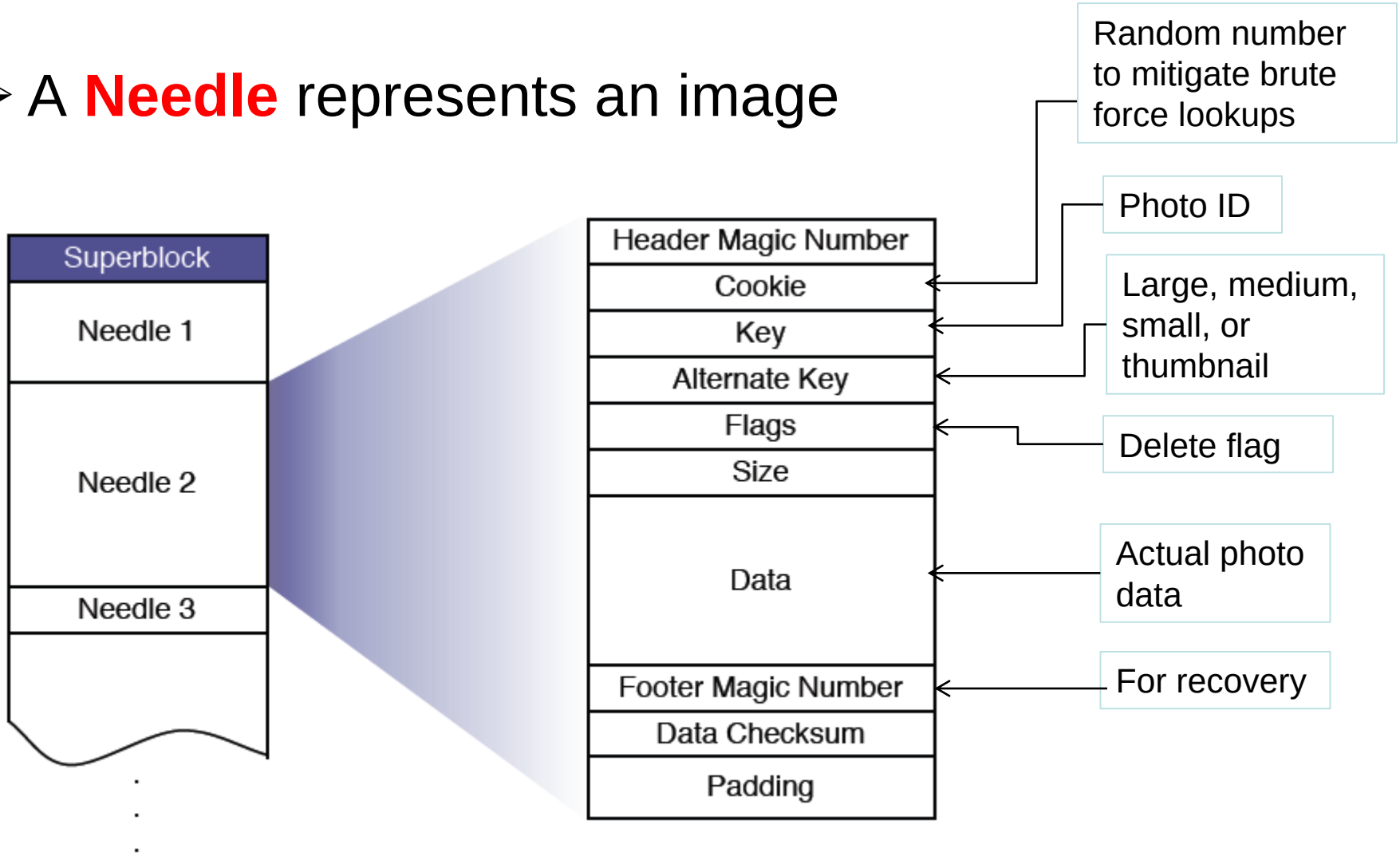
- Four main functions:
  - Provides a mapping from logical volumes to physical volumes
  - Load balances writes across logical volumes
  - Determines whether a photo request should be handled by the CDN or by the Haystack Cache
  - Identifies logical volumes that are read-only
    - operational reasons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

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

- A **Needle** represents an image

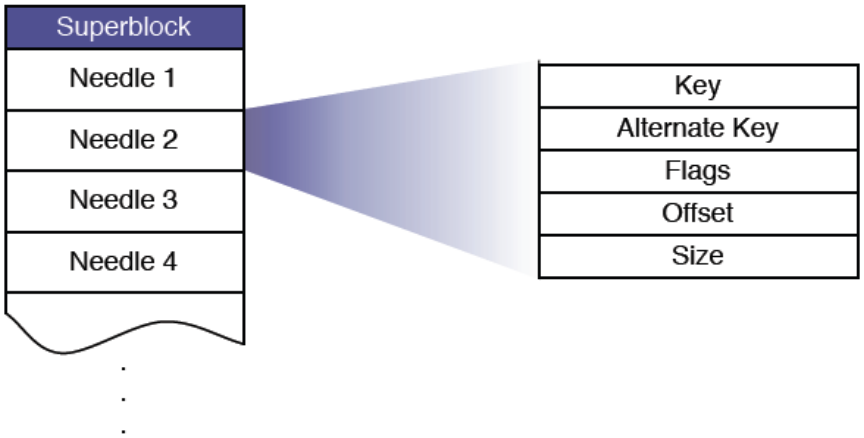


Layout of Haystack Store file

Haystack Store

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

- 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 for images; another 4 for indexes.
- Garbage collection?
  - Create a copy of the store file
  - Skip deleted photos while copying
  - 25% of photos get deleted in a given year

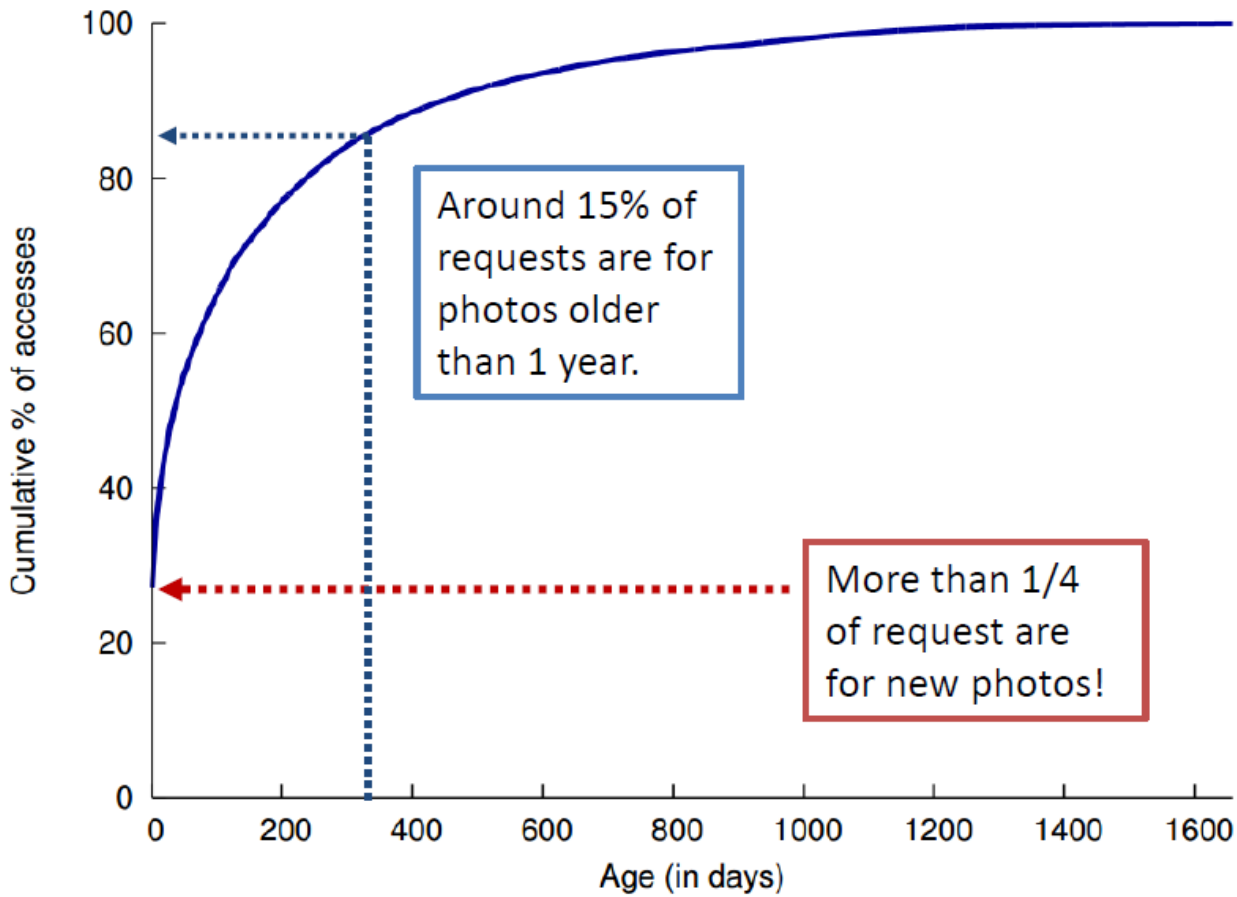
Experiments

- 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

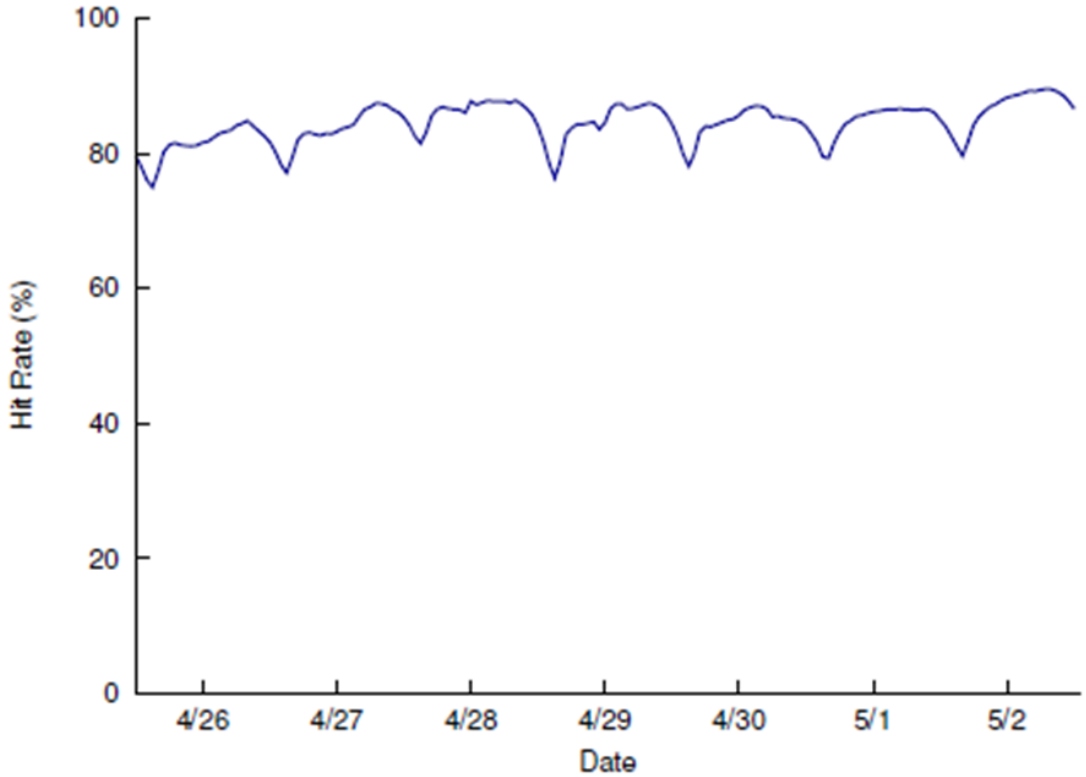


Figure 9: Cache hit rate for images that might be potentially stored in the Haystack Cache.

Volume of Daily Total Traffic

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

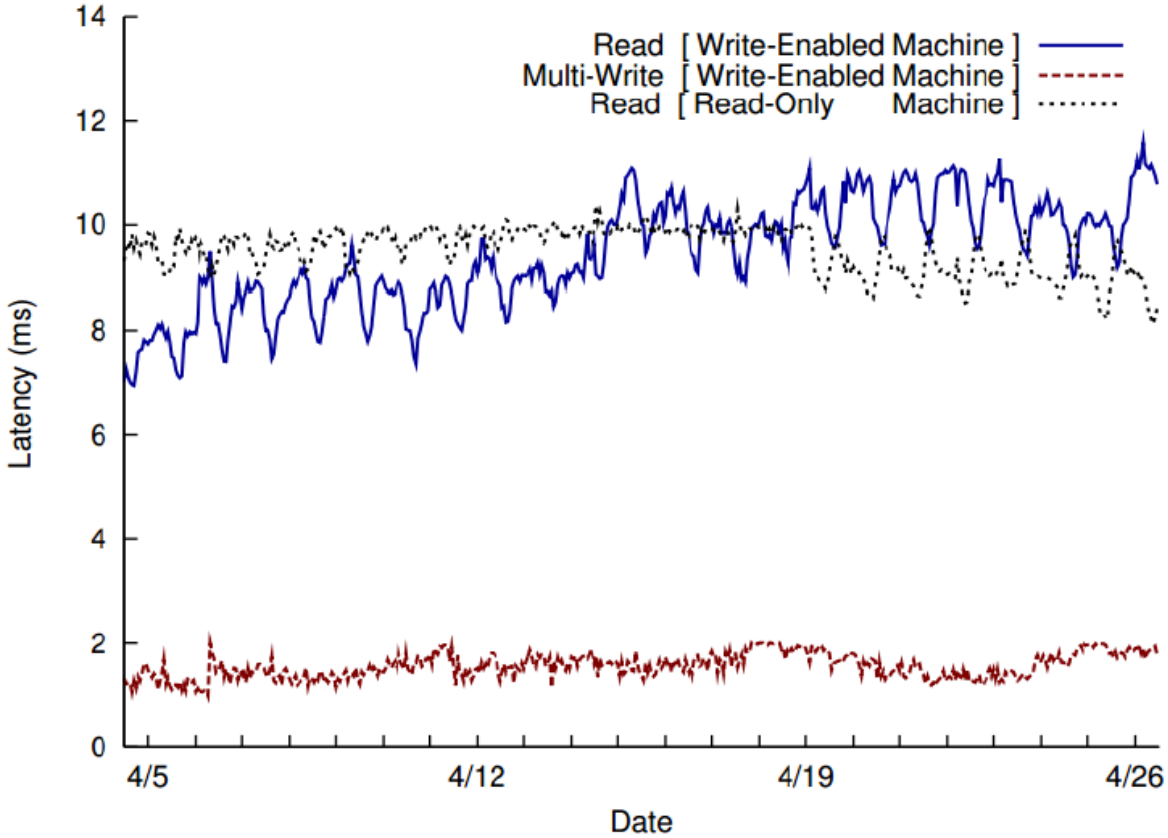
So, this matches Facebook's goal to build a custom system that saves io ops for small files.

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

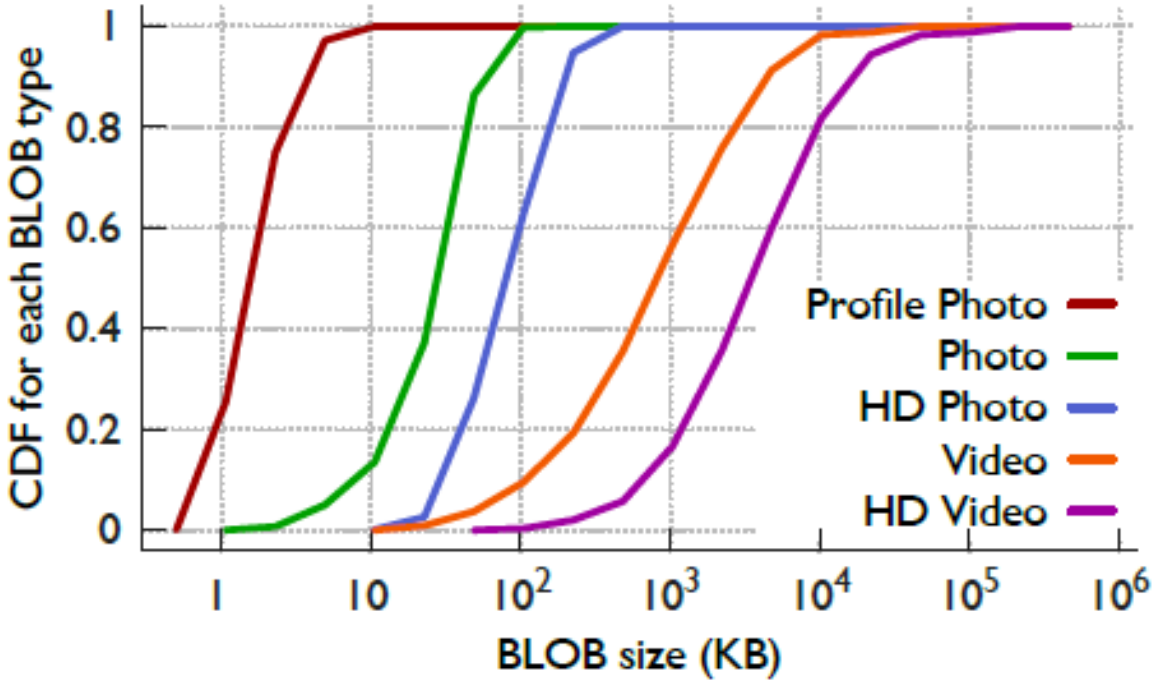
- This is an efficient storage of billions of photos.

- 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!

BLOB Storage

- An important class of data that Facebook stores is **Binary Large OBjects (BLOBs)**
- BLOBs are immutable binary data
  - Created once, read many times, never modified, may be deleted
- Examples of BLOBs
  - Photos, videos, documents, traces, heap dumps, and source code
- As of Feb 2014, Facebook stores 400 billion photos

BLOB Size Distribution

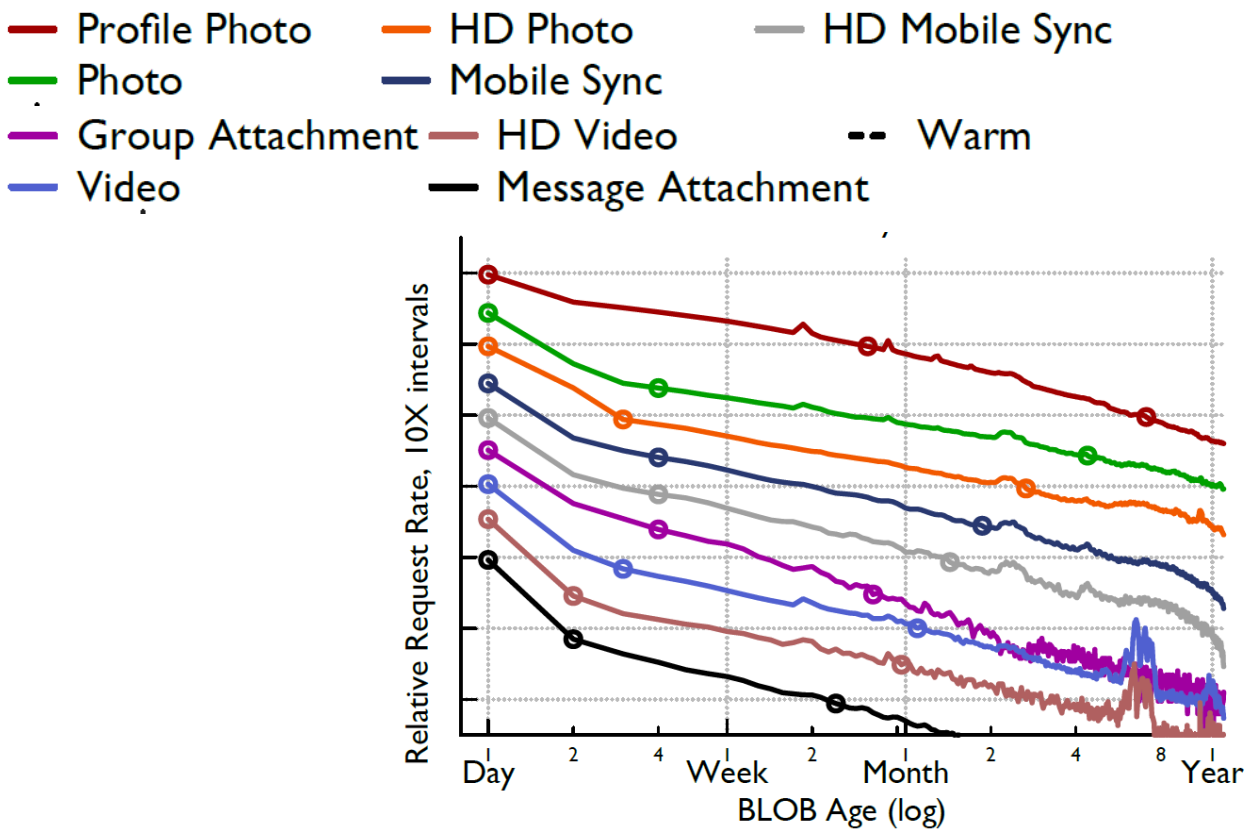


- There exist significant diversities in BLOB sizes for different types

Temperature Zones Exist

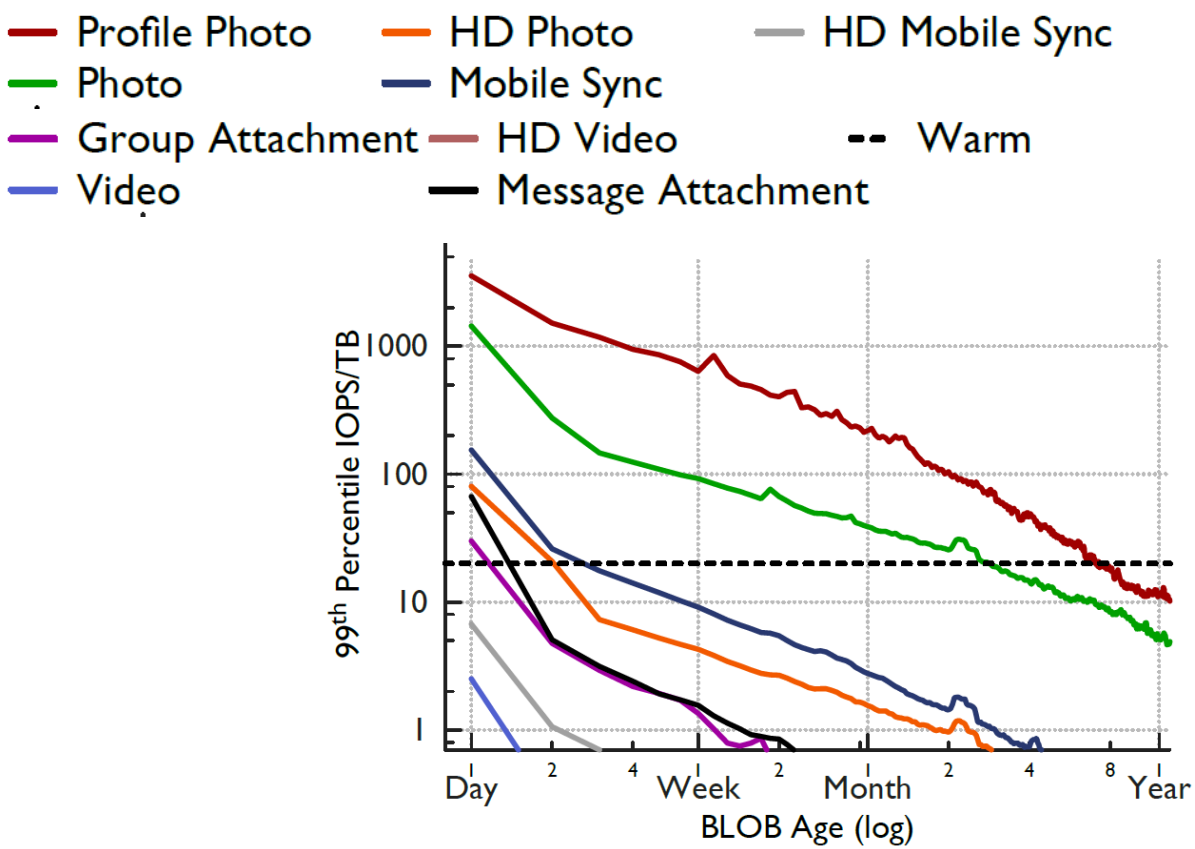
- Temperatures
  - **Hot**: BLOBs receive many requests
  - **Warm**: BLOBs receive few requests
  - **Cold**: BLOBs almost receive no requests
    - Need to be stored persistently
    - Take days or hours to retrieve
- Strong correlation between age and temperature of a BLOB

Request Rates



- Request rates drop over time
  - Content less than one day old receives more than 100 times the request rate of one-year-old content.

99th Percentile Load

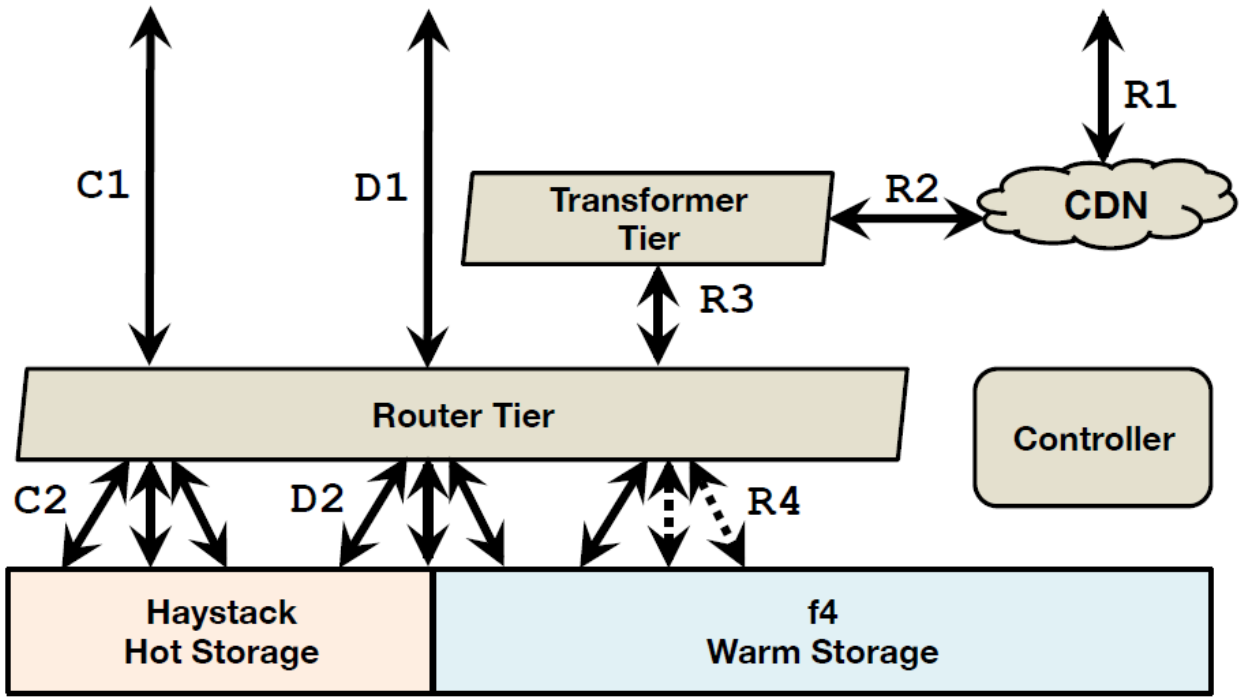


- From hot to warm:
  - 7 of 9 content types: less than a week
  - Photos: three months; Profile photos: 1 year

Warm Content is a Lot!

- Warm cutoff: 1 month
- More than 80% of objects are warm of all types
  - Remaining objects are hot
- Applying **replication** to warm content provides fault tolerance, but introduces very high storage overhead
  - Haystack uses replication for fault tolerance
  - Not suitable for using Haystack for warm storage

Facebook's BLOB Storage



- Overall BLOB Storage Architecture with creates (C1-C2), deletes (D1-D2), and reads (R1-R4). Creates are handled by Haystack, most deletes are handled by Haystack, reads are handled by either Haystack or f4. See paper for details.

Haystack Fault Tolerance

- Triple replication of data files + (10+2)-RAID-6
  - Three replicas (in three hosts, each in a distinct rack)
  - RAID-6 on 12 disks of each replica
- Replication factor = 1.2 x 3 = 3.6x
  - What is the minimum number of disk failures that cause data loss?

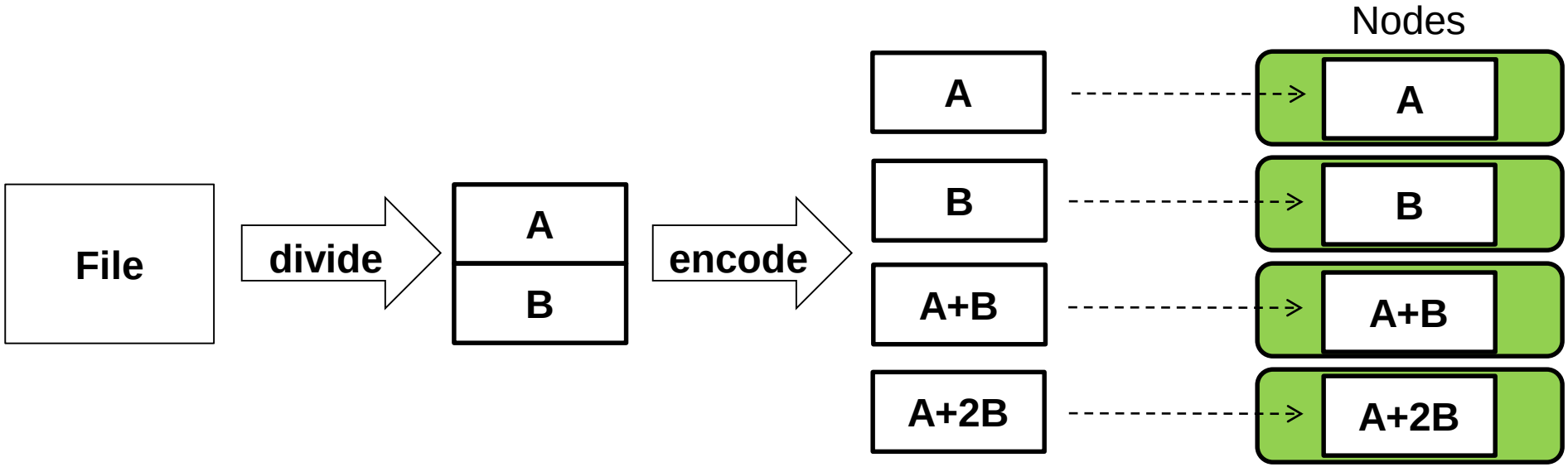
- Haystack is optimized for performance (in IOPS) but not storage efficiency

f4 Fault Tolerance

- Goal: achieve storage-efficient fault tolerance
- Four types of failures:
  - Drive failures, at a low single digit annual rate
  - Host failures, periodically
  - Rack failures, multiple time per year
  - Datacenter failures, extremely rare and usually transient but potentially more disastrous
- A datacenter is partitioned in **cells**, each using 14 racks of 15 hosts with 30 4TB drives per host
  - Each cell has 25PB raw storage
- Key: **Erasure coding**

Erasure Coding

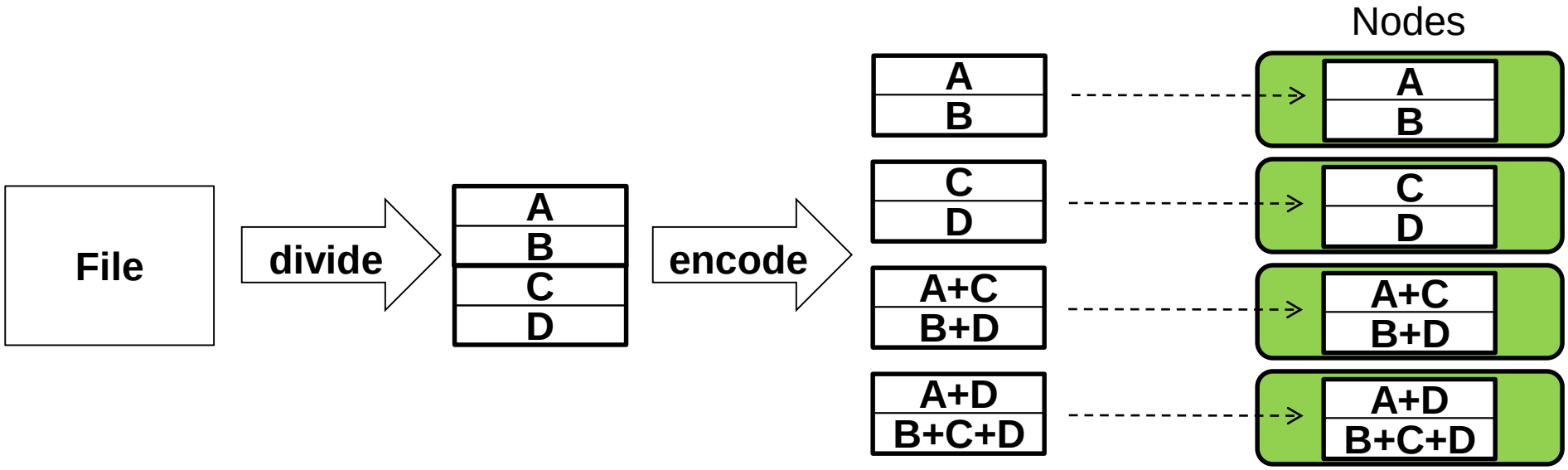
- Divide file to **k** data blocks
- Encode data blocks to additional **n-k** parity blocks
  - Encoding can be done via Galois field arithmetic
- Distribute n data/parity blocks (a stripe) to n nodes
- **Fault tolerance**: any k out of n blocks can recover file data



(n, k) = (4, 2)

Erasure Coding

- Encoding can also be done via purely additions (equivalent to **bitwise XOR** in Galois field arithmetic of base 2)

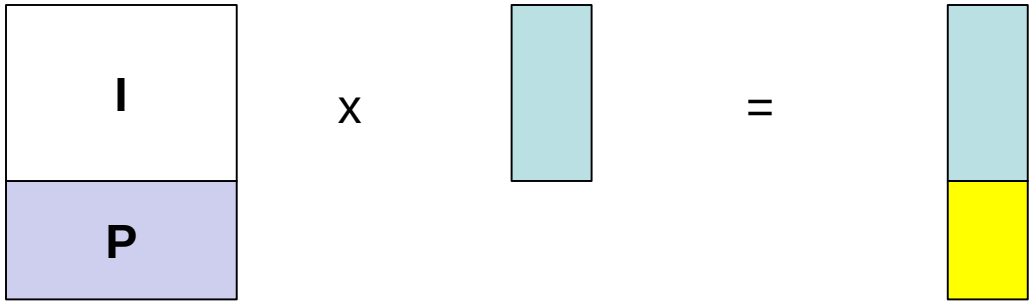


(n, k) = (4, 2)

Erasure Coding

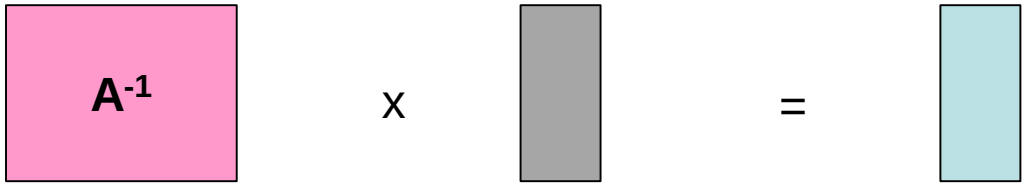
I = k * k identity matrix
P = (n-k) * k parity-check matrix

- Encoding:



n x k generator matrix **G** k data blocks n-k parity blocks

- Decoding:



Inverse of any sub-matrix **A** of k rows of **G** any k available blocks k data blocks

Erasure Coding

- Many erasure coding constructions exist; Reed-Solomon codes are the most popular
- **Reed-Solomon codes**
  - Support any n and k (provided that n > k)
  - Minimum storage redundancy to provide the fault tolerance against any n-k failures
- f4: (n, k) = (14, 10)

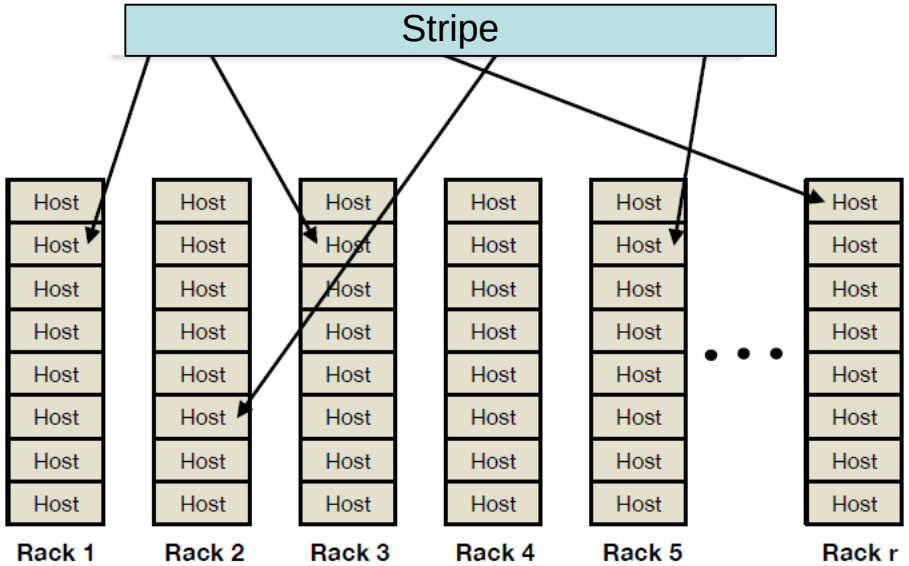
- Note: our definitions of n and k are different from the f4 paper

Erasure Coding: Trade-offs

- Advantages:
  - Reduce storage space
  - High fault tolerance
- Disadvantages:
  - In general, k blocks needed to recover a lost block
    → **expensive repair (read amplification)**

Erasure Coding in f4

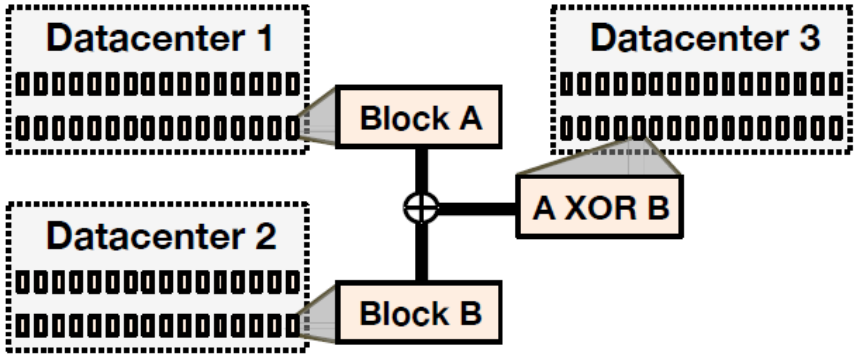
- Block size = 1GB
  - Each block has multiple BLOBs
  - Large block size reduces metadata management
- Block layout in an f4 cell
  - Distribute each RS(14,10) stripe across 14 racks
  - Tolerate rack, host, and disk failures of an f4 cell



Erasure Coding in f4

How to tolerate data center failures?

- **Double replication**: Replicating a stripe in two cells at two different data centers
  - Redundancy factor = 1.4 * 2 = 2.8x
- **Geo-distributed XOR coding**: Add an XOR parity to two stripes
  - Redundancy factor = 1.4 * 1.5 = 2.1x

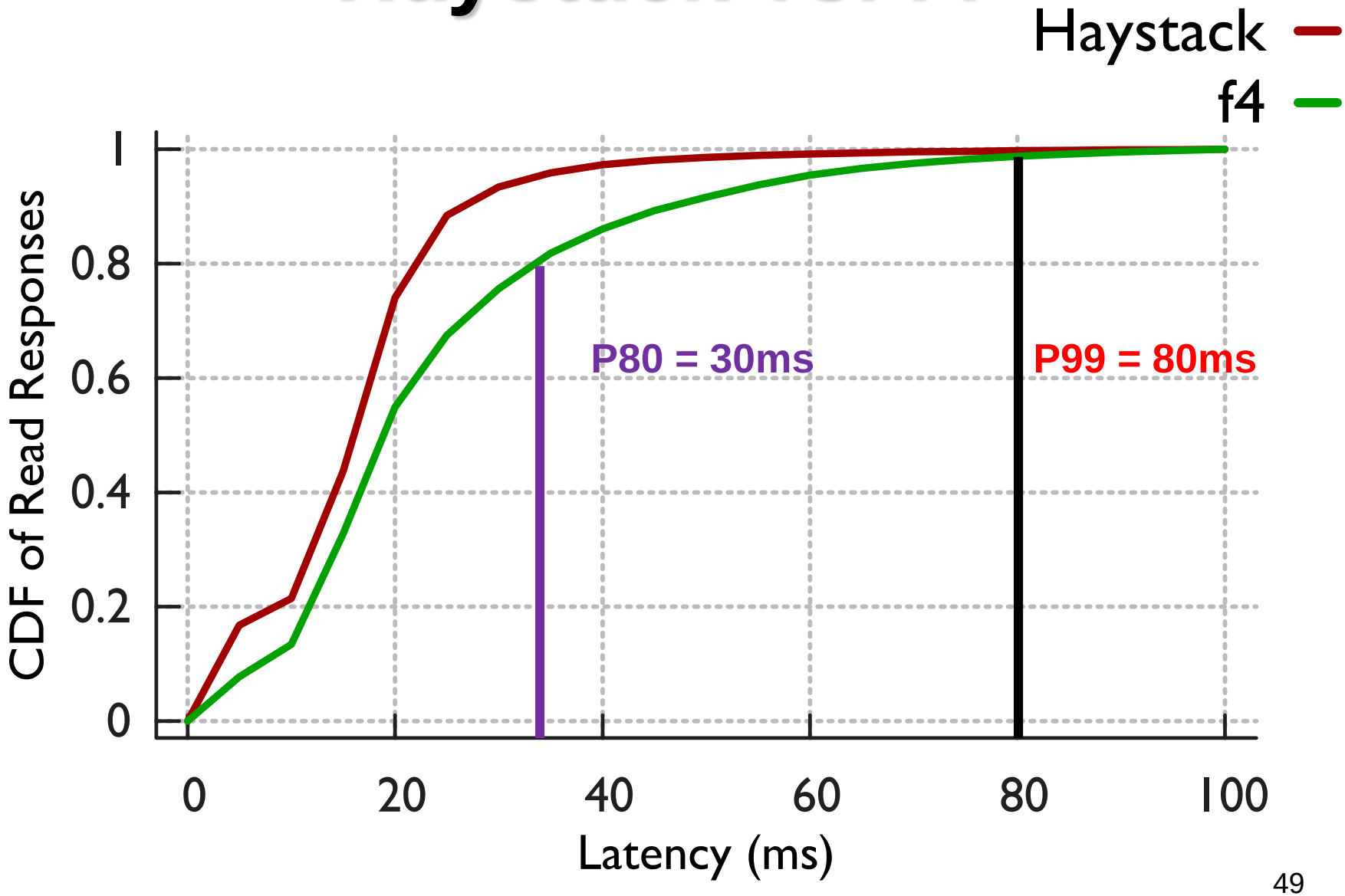


Haystack vs. f4

	Haystack with 3 copies	f4 2.8	f4 2.1
Redundancy Factor	3.6X	*2.8X*	***2.1X***
Irrecoverable Disk Failures	9	***10***	***10***
Irrecoverable Host Failures	3	***10***	***10***
Irrecoverable Rack failures	3	***10***	***10***
Irrecoverable Datacenter failures	***3***	2	2
Load split	***3X***	2X	1X

Source: OSDI'14 slides

Haystack vs. f4



f4: Conclusions

- Facebook's BLOB storage is big and growing
- BLOBs cool down with age
  - ~100X drop in read requests in 60 days

- Haystack's 3.6x replication introduces too much storage overhead for warm data
- f4 encodes data to lower replication to 2.1x