Finding a needle in Haystack: Facebook's photo storage

April 9, 2025

1 Introduction

Data is written once, read often, never modified, and rarely deleted.

In our experience, we find that the disadvantages of a traditional POSIX based filesystem are directories and per file metadata.

- For the Photos application most of this metadata, such as permissions, is unused and thereby wastes storage capacity.
- Yet the more significant cost is that the file's metadata must be read from disk into memory in order to find the file itself.

2 Background & Previous Design

2.1 Background

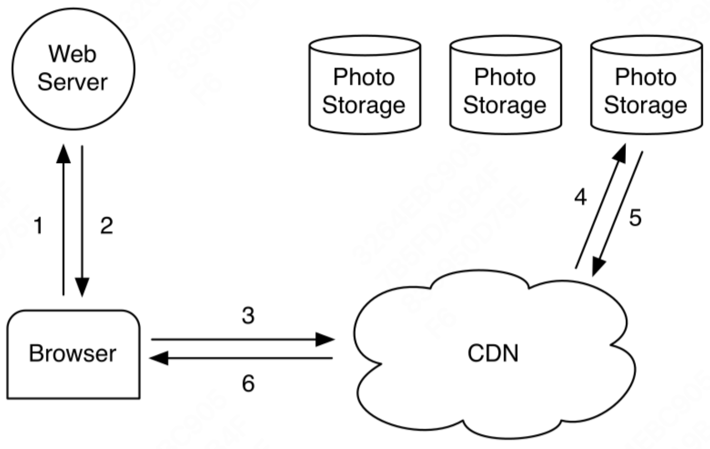


Figure 1: Typical Design

Figure 1 depicts the steps from the moment when a user visits a page containing an image until she downloads that image from its location on disk.

1. When visiting a page the user's browser first sends an HTTP request to a web server which is responsible for generating the markup for the browser to render.

2. For each image the web server constructs a URL directing the browser to a location from which to download the data.

3. For popular sites this URL often points to a CDN. If the CDN has the image cached then the CDN responds immediately with the data.

4. Otherwise, the CDN examines the URL, which has enough information embedded to retrieve the photo from the site's storage systems.

5. The CDN then updates its cached data and sends the image to the user's browser.

2.2 NFS-based Design

In our first design we implemented the photo storage system using an NFS-based approach.

The major lesson we learned is that CDNs by themselves do not offer a practical solution to serving photos on a social networking site. CDNs do effectively serve the hottest photos, but a social networking site like Facebook also generates a large number of requests for less popular (often older) content, which we refer to as the long tail. Requests from the long tail account for a significant amount of our traffic, almost all of which accesses the backing photo storage hosts as these requests typically miss in the CDN.

Our NFS-based design stores each photo in its own file on a set of commercial NAS appliances. A set of machines, Photo Store servers, then mount all the volumes exported by these NAS appliances over NFS.

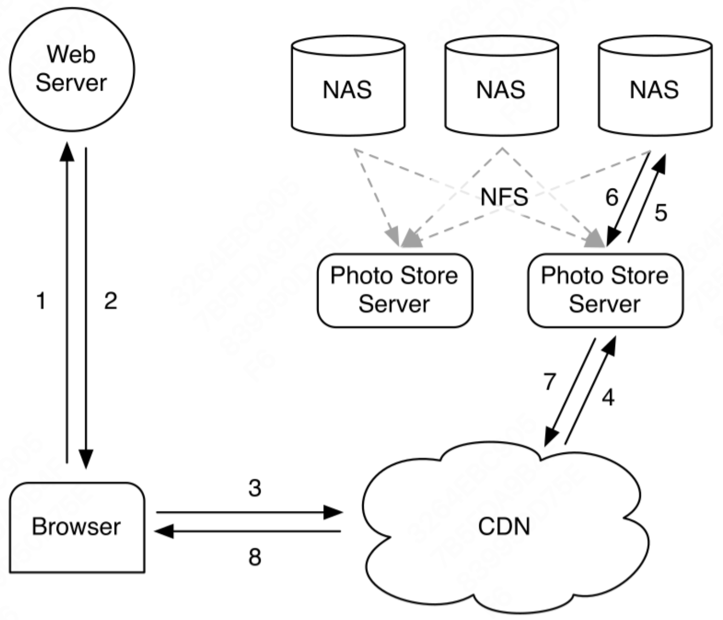


Figure 2: NFS-based Design

We initially stored thousands of files in each directory of an NFS volume which led to an excessive number of disk operations to read even a single image. Because of how the NAS appliances manage directory metadata, placing thousands of files in a directory was extremely inefficient as the directory's blockmap was too large to be cached effectively by the appliance.

Consequently it was common to incur more than 10 disk operations to retrieve a single image. After reducing directory sizes to hundreds of images per directory, the resulting system would still generally incur 3 disk operations to fetch an image: one to read the directory metadata into memory, a second to load the inode into memory, and a third to read the file contents.

To further reduce disk operations we let the Photo Store servers explicitly cache file handles returned by the NAS appliances. When reading a file for the first time a Photo Store server opens a file normally but also caches the filename to file handle mapping in memcache. When requesting a file whose file handle is cached, a Photo Store server opens the file directly using a custom system call, `open_by_filehandle`, that we added to the kernel. Regrettably, this file handle cache provides only a minor improvement as less popular photos are less likely to be cached to begin with.

The major lesson we learned from the NAS approach is that focusing only on caching— whether the NAS appliance's cache or an external cache like memcache—has limited impact for reducing disk operations.

2.3 Discussion

One could phrase the dilemma we faced as existing storage systems lacked the right RAM-to-disk ratio. However, there is no right ratio. The system just needs enough main memory so that all of the filesystem metadata can be cached at once. In our NAS-based approach, one photo corresponds to one file and each file requires at least one inode, which is hundreds of bytes large. Having enough main memory in this approach is not cost-effective. To achieve a better price/performance point, we decided to build a custom storage system that reduces the amount of filesystem metadata per photo so that having enough main memory is dramatically more cost-effective than buying more NAS appliances.

3 Design & Implementation

In the following description of Haystack, we distinguish between two kinds of metadata. **Application metadata** describes the information needed to construct a URL that a browser can use to retrieve a photo. **Filesystem metadata** identifies the data necessary for a host to retrieve the photos that reside on that host's disk.

3.1 Overview

The Haystack architecture consists of 3 core components: the Haystack Store, Haystack Directory, and Haystack Cache.

The Store encapsulates the persistent storage system for photos and is the only component that manages the filesystem metadata for photos. We organize the Store's capacity by physical volumes. For example, we can organize a server's 10 terabytes of capacity into 100 physical volumes each of which provides 100 gigabytes of storage. We further group physical volumes on different machines into logical volumes. When Haystack stores a photo on a logical volume, the photo is written to all corresponding physical volumes. This redundancy allows us to mitigate data loss due to hard drive failures, disk controller bugs, etc.

The Directory maintains the logical to physical mapping along with other application metadata, such as the logical volume where each photo resides and the logical volumes with free space.

The Cache functions as our internal CDN, which shelters the Store from requests for the most popular photos and provides insulation if upstream CDN nodes fail and need to refetch content.

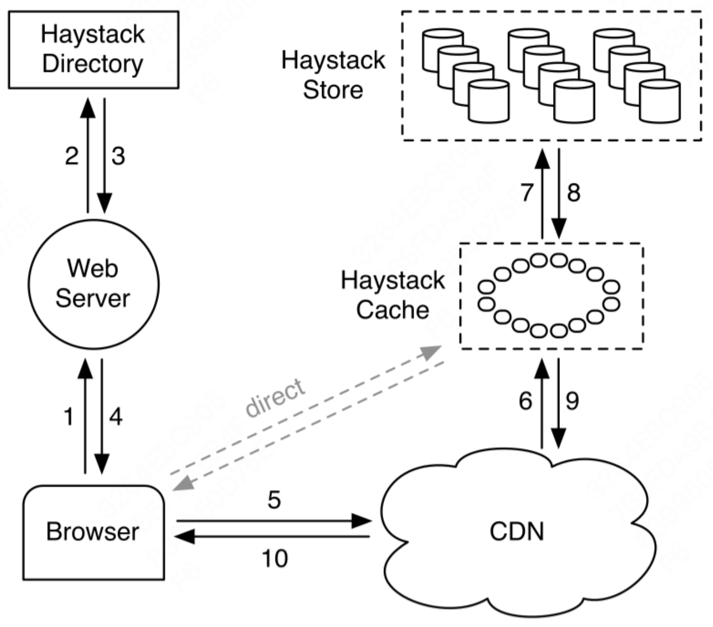


Figure 3: Serving a photo

When a user visits a page the web server uses the Directory to construct a URL for each photo. The URL contains several pieces of information, each piece corresponding to the sequence of steps from when a user's browser contacts the CDN (or Cache) to ultimately retrieving a photo from a machine in the Store. A typical URL that directs the browser to the CDN looks like the following:

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

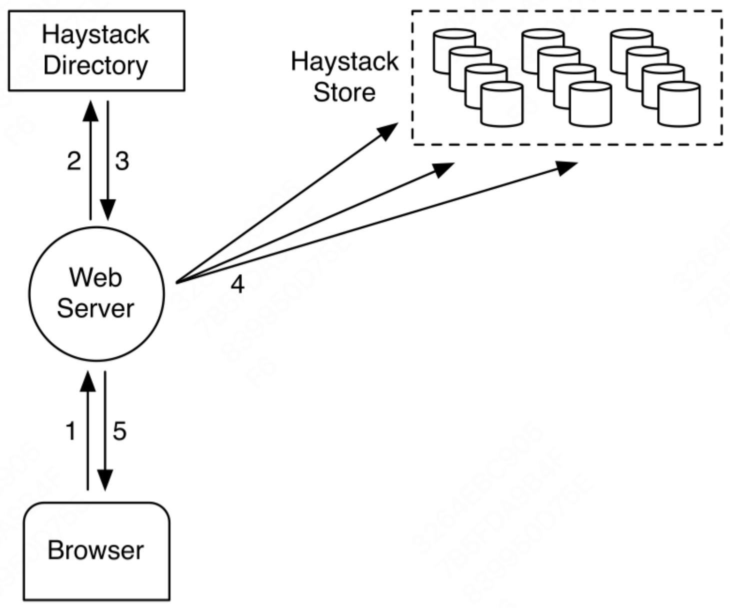


Figure 4: Uploading a photo

3.2 Haysatack Directory

The Directory serves four main functions.

1. It provides a mapping from logical volumes to physical volumes. Web servers use this mapping when uploading photos and also when constructing the image URLs for a page request.

2. The Directory load balances writes across logical volumes and reads across physical volumes.

3. the Directory determines whether a photo request should be handled by the CDN or by the Cache. This functionality lets us adjust our dependence on CDNs.

4. the Directory identifies those logical volumes that are read-only either because of operational reasons or because those volumes have reached their storage capacity. We mark volumes as read-only at the granularity of machines for operational ease.

When we increase the capacity of the Store by adding new machines, those machines are write-enabled; only write-enabled machines receive uploads. Over time the available capacity on these machines decreases. When a machine exhausts its capacity, we mark it as read-only.

3.3 Haystack Cache

The Cache receives HTTP requests for photos from CDNs and also directly from users' browsers. We organize the Cache as a distributed hash table and use a photo's id as the key to locate cached data. If the Cache cannot immediately respond to the request, then the Cache fetches the photo from the Store machine identified in the URL and replies to either the CDN or the user's browser as appropriate.

We now highlight an important behavioral aspect of the Cache. It caches a photo only if two conditions are met:

1. the request comes directly from a user and not the CDN
2. the photo is fetched from a write-enabled Store machine.

The justification for the first condition is that our experience with the NFS-based design showed post-CDN caching is ineffective as it is un- likely that a request that misses in the CDN would hit in our internal cache.

The reasoning for the second is indirect. We use the Cache to shelter write-enabled Store machines from reads because of two interesting properties:

1. photos are most heavily accessed soon after they are uploaded
2. filesystems for our workload generally perform better when doing either reads or writes but not both.

Thus the write-enabled Store machines would see the most reads if it were not for the Cache. Given this characteristic, an optimization we plan to implement is to proactively push recently uploaded photos into the Cache as we expect those photos to be read soon and often.

3.4 Haystack Store

Reads make very specific and well-contained requests asking for a photo with a given id, for a certain logical volume, and from a particular physical Store machine.

Each Store machine manages multiple physical volumes. Each volume holds millions of photos. For concreteness, the reader can think of a physical volume as simply a very large file (100 GB) saved as `/hay/haystack <logical volume id>`. A Store machine can access a photo quickly using only the id of the corresponding logical volume and the file offset at which the photo resides. This knowledge is the keystone of the Haystack design: retrieving the filename, offset, and size for a particular photo without needing disk operations. A Store machine keeps open file descriptors for each physical volume that it manages and also an in-memory mapping of photo ids to the filesystem metadata (i.e., file, offset and size in bytes) critical for retrieving that photo.

A Store machine represents a physical volume as a large file consisting of a superblock followed by a sequence of **needles**. Each needle represents a photo stored in Haystack.

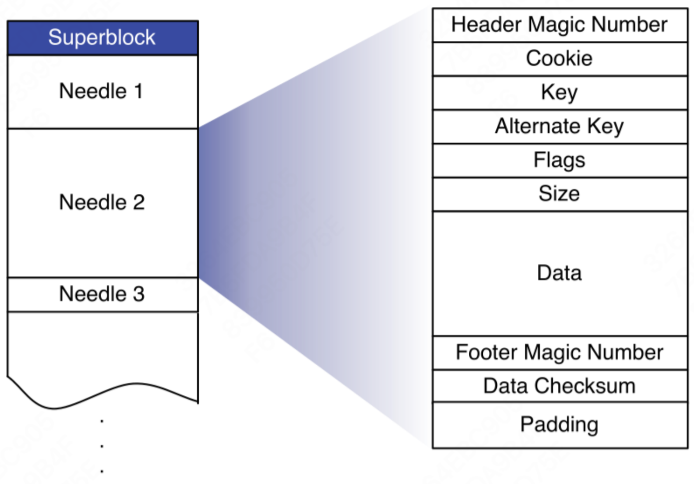
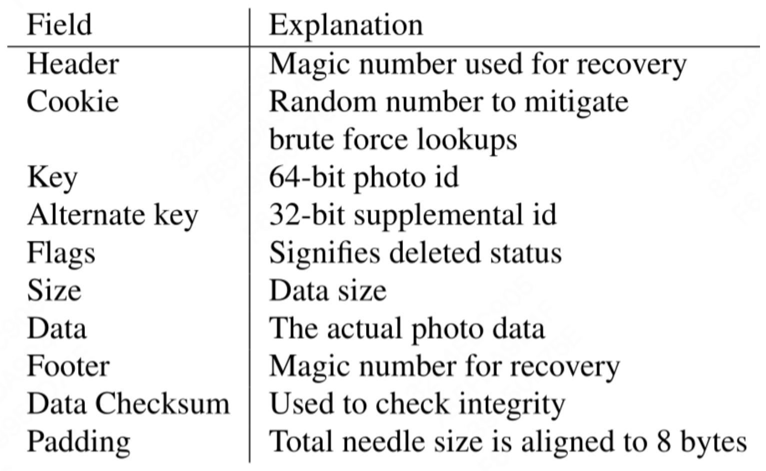


Figure 5: Layout of Haystack Store file



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

Figure 6: Explanation of fields in a needle

To retrieve needles quickly, each Store machine maintains an in-memory data structure for each of its volumes. That data structure maps pairs of (key, alternate key) to the corresponding needle's flags, size in bytes, and volume offset. After a crash, a Store machine can reconstruct this mapping directly from the volume file before processing requests.

3.4.1 Photo Read

3.4.2 Photo Write

When uploading a photo into Haystack web servers provide the 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.

While simple, this append-only restriction complicates some operations that modify photos, such as rotations. As Haystack disallows overwriting needles, photos can only be modified by adding an updated needle with the same key and alternate key.

- If the new needle is written to a different logical volume than the original, the Directory updates its application metadata and future requests will never fetch the older version.

- If the new needle is written to the same logical volume, then Store machines append the new needle to the same corresponding physical volumes.

Haystack distinguishes such duplicate needles based on their offsets. That is, the latest version of a needle within a physical volume is the one at the highest offset.

3.4.3 Photo Delete

Deleting a photo is straight-forward. A Store machine sets the delete flag in both the in-memory mapping and synchronously in the volume file.

3.4.4 The Index File

Store machines use an important optimization—the **index file** —when re-booting. Index files allow a Store machine to build its in-memory mappings quickly, shortening restart time.

Store machines maintain an index file for each of their volumes. The index file is a checkpoint of the in-memory data structures used to locate needles efficiently on disk. An index file's layout is similar to a volume file's, containing a superblock followed by a sequence of index records corresponding to each needle in the su- perblock. These records must appear in the same order as the corresponding needles appear in the volume file.

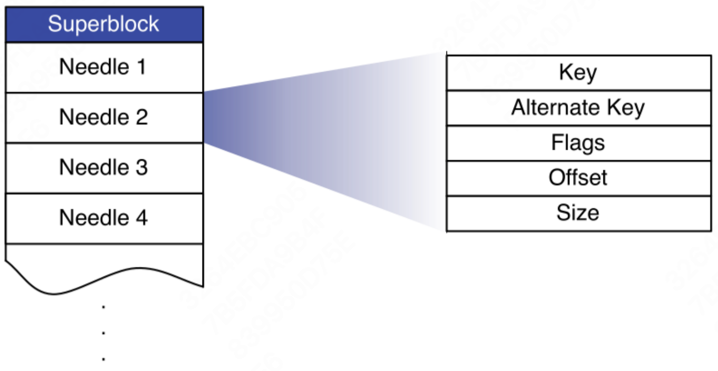


Figure 7: Layout of Haystack Index file

Restarting using the index is slightly more complicated than just reading the indices and initializing the in-memory mappings. The complications arise because index files are updated asynchronously, meaning that index files may represent stale checkpoints.

- When we write a new photo the Store machine synchronously appends a needle to the end of the volume file and asynchronously appends a record to the index file.

- When we delete a photo, the Store machine synchronously sets the flag in that photo's needle without updating the index file.

These design decisions allow write and delete operations to return faster because they avoid additional synchronous disk writes. They also cause two side effects we must address:

- needles can exist without corresponding index records

- index records do not reflect deleted photos.

We refer to needles without corresponding index records as **orphans**. During restarts, a Store machine sequentially examines each orphan, creates a matching index record, and appends that record to the index file. Note that we can quickly identify orphans because the last record in the index file corresponds to the last non-orphan needle in the volume file.

Since index records do not reflect deleted photos, a Store machine may retrieve a photo that has in fact been deleted. To address this issue, after a Store machine reads the entire needle for a photo, that machine can then inspect the deleted flag. If a needle is marked as deleted the Store machine updates its in-memory map- ping accordingly and notifies the Cache that the object was not found.

3.4.5 Filesystem

In particular, the Store machines should use a filesystem that does not need much memory to be able to perform random seeks within a large file quickly.

Currently, each Store machine uses XFS, an extent based file system. XFS has two main advantages for Haystack.

1. the blockmaps for several contiguous large files can be small enough to be stored in main memory.

2. XFS provides efficient file preallocation, mitigating fragmentation and reining in how large block maps can grow.

3.4.6 Recovery from failures

To proactively find Store machines that are having problems, we maintain a background task, dubbed **pitchfork**, that periodically checks the health of each Store machine. Pitchfork remotely tests the connection to each Store machine, checks the availability of each volume file, and attempts to read data from the Store machine. If pitchfork determines that a Store machine consistently fails these health checks then pitchfork automatically marks all logical volumes that reside on that Store machine as read-only. We manually address the underlying cause for the failed checks offline.

Once diagnosed, we may be able to fix the problem quickly. Occasionally, the situation requires a more heavy-handed bulk sync operation in which we reset the data of a Store machine using the volume files supplied by a replica. Bulk syncs happen rarely (a few each month) and are simple albeit slow to carry out. The main bottleneck is that the amount of data to be bulk synced is often orders of magnitude greater than the speed of the NIC on each Store machine, resulting in hours for mean time to recovery. We are actively exploring techniques to address this constraint.

3.5 Optimizations

3.5.1 Compaction

A Store machine compacts a volume file by copying needles into a new file while skipping any duplicate or deleted entries. During compaction, deletes go to both files. Once this procedure reaches the end of the file, it blocks any further modifications to the volume and atomically swaps the files and in-memory structures.

3.5.2 Saving more memory

3.5.3 Batch upload

4 Problems

5 References