efficient storage of billions of photos

The Photos application is one of Facebook's most popular features. Up to date, users have uploaded over 15 billion photos which makes Facebook the biggest photo sharing website. For each uploaded photo, Facebook generates and stores four images of different sizes, which translates to a total of 60 billion images and 1.5PB of storage. The current growth rate is 220 million new photos per week, which translates to 25TB of additional storage consumed weekly. At the peak there are 550,000 images served per second. These numbers pose a significant challenge for the Facebook photo storage infrastructure.

The old photo infrastructure consisted of several tiers:

- Upload tier receives users' photo uploads, scales the original images and saves them on the NFS storage tier.
- Photo serving tier receives HTTP requests for photo images and serves them from the NFS storage tier.
- NFS storage tier built on top of commercial storage appliances.

Since each image is stored in its own file, there is an enormous amount of metadata 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, resulting in multiple I/O operations per photo upload or read request. The whole photo serving infrastructure is bottlenecked on the high metadata overhead of the NFS storage tier, which is one of the reasons why Facebook relies heavily on CDNs to serve photos. Two additional optimizations were deployed in order to mitigate this problem to some degree:

- Cachr: a caching server tier caching smaller Facebook "profile" images.
- NFS file handle cache - deployed on the photo serving tier eliminates some of the NFS storage tier metadata overhead

The new photo infrastructure merges the photo serving tier and storage tier into one physical tier. It implements a HTTP based photo server which stores photos in a generic object store called Haystack. The main requirement for the new tier was to eliminate any unnecessary metadata overhead for photo read operations, so that each read I/O operation was only reading actual photo data (instead of filesystem metadata). Haystack can be broken down into these functional layers -

- HTTP server
- Photo Store
- Haystack Object Store
- Filesystem
- Storage

In the following sections we look closely at each of the functional layers from the bottom up.Haystack is deployed on top of commodity storage blades. The typical hardware configuration of a 2U storage blade is –

- 2 x quad-core CPUs
- 16GB – 32GB memory
- hardware raid controller with 256MB – 512MB of NVRAM cache
- 12+ 1TB SATA drives

Each storage blade provides around 10TB of usable space, configured as a RAID-6 partition managed by the hardware RAID controller. RAID-6 provides adequate redundancy and excellent read performance while keeping the storage cost down. The poor write performance is partially mitigated by the RAID controller NVRAM write-back cache. Since the reads are mostly random, the NVRAM cache is fully reserved for writes. The disk caches are disabled in order to guarantee data consistency in the event of a crash or a power loss.Haystack object stores are implemented on top of files stored in a single filesystem created on top of the 10TB volume.

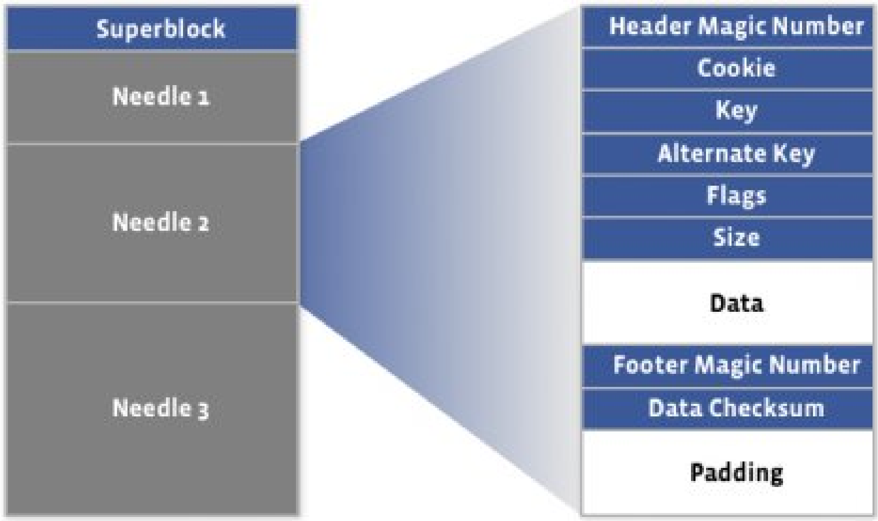
Photo read requests result in read() system calls at known offsets in these files, but in order to execute the reads, the filesystem must first locate the data on the actual physical volume. Each file in the filesystem is represented by a structure called an inode which contains a block map that maps the logical file offset to the physical block offset on the physical volume. For large files, the block map can be quite large depending on the type of the filesystem in use.

Block based filesystems maintain mappings for each logical block, and for large files, this information will not typically fit into the cached inode and is stored in indirect address blocks instead, which must be traversed in order to read the data for a file. There can be several layers of indirection, so a single read could result in several I/Os depending on whether or not the indirect address blocks are cached.

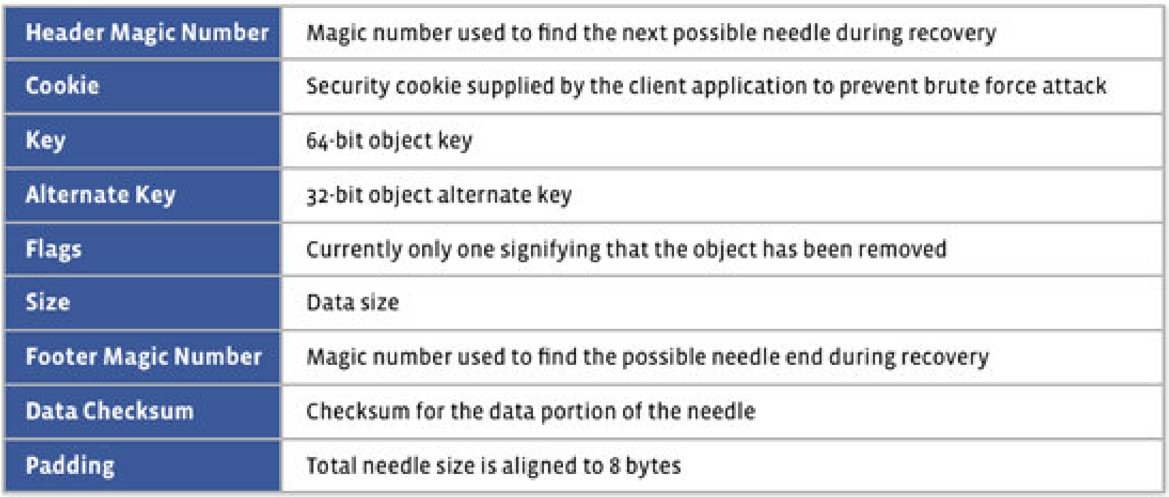
Extent based filesystems maintain mappings only for contiguous ranges of blocks (extents). A block map for a contiguous large file could consist of only one extent which would fit in the inode itself. However, if the file is severely fragmented and its blocks are not contiguous on the underlying volume, its block map can grow large as well. With extent based filesystems, fragmentation can be mitigated by aggressively allocating a large chunk of space whenever growing the physical file.

Currently, the filesystem of choice is XFS, an extent based filesystem providing efficient file preallocation.

Haystack is a simple log structured (append-only) object store containing needles representing the stored objects. A Haystack consists of two files – the actual haystack store file containing the needles, plus an index file. The following figure shows the layout of the haystack store file:

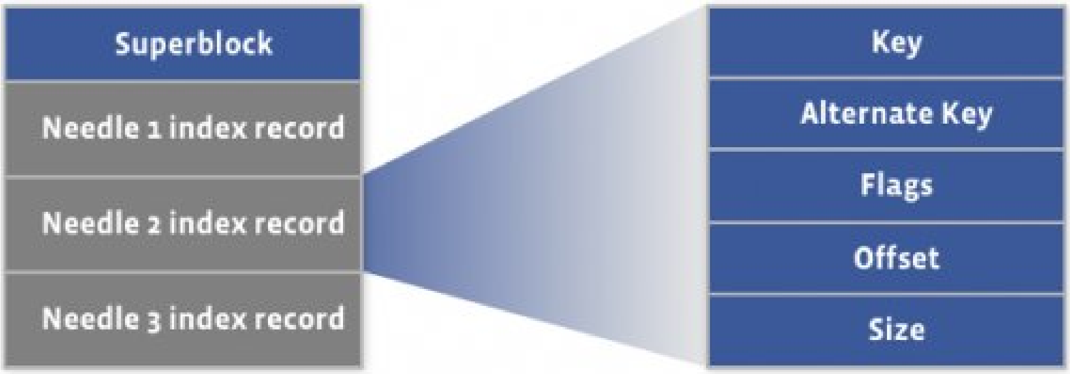


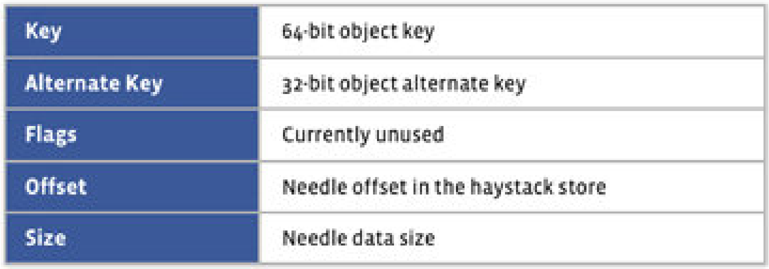
The first 8KB of the haystack store is occupied by the superblock. Immediately following the superblock are needles, with each needle consisting of a header, the data, and a footer:



Header Magic Number	Magic number used to find the next possible needle during recovery
Cookie	Security cookie supplied by the client application to prevent brute force attack
Key	64-bit object key
Alternate Key	32-bit object alternate key
Flags	Currently only one signifying that the object has been removed
Size	Data size
Footer Magic Number	Magic number used to find the possible needle end during recovery
Data Checksum	Checksum for the data portion of the needle
Padding	Total needle size is aligned to 8 bytes

A needle is uniquely identified by its tuple, where the offset is the needle offset in the haystack store. Haystack doesn't put any restriction on the values of the keys, and there can be needles with duplicate keys. Following figure shows the layout of the index file -





Key	64-bit object key
Alternate Key	32-bit object alternate key
Flags	Currently unused
Offset	Needle offset in the haystack store
Size	Needle data size

There is a corresponding index record for each needle in the haystack store file, and the order of the needle index records must match the order of the associated needles in the haystack store file. The index file provides the minimal metadata required to locate a particular needle in the haystack store file. Loading and organizing index records into a data structure for efficient lookup is the responsibility of the Haystack application (Photo Store in our case). The index file is not critical, as it can be rebuilt from the haystack store file if required. The main purpose of the index is to allow quick loading of the needle metadata into memory without traversing the larger Haystack store file, since the index is usually less than 1% the size of the store file.

Haystack Write Operation

A Haystack write operation synchronously appends new needles to the haystack store file. After the needles are committed to the larger Haystack store file, the corresponding index records are then written to the index file. Since the index file is not critical, the index records are written asynchronously for faster performance.

The index file is also periodically flushed to the underlying storage to limit the extent of the recovery operations caused by hardware failures. In the case of a crash or a sudden power loss, the haystack recovery process discards any partial needles in the store and truncates the haystack store file to the last valid needle. Next, it writes missing index records for any trailing orphan needles at the end of the haystack store file.

Haystack doesn’t allow overwrite of an existing needle offset, so if a needle’s data needs to be modified, a new version of it must be written using the same tuple. Applications can then assume that among the needles with duplicate keys, the one with the largest offset is the most recent one.

Haystack Read Operation

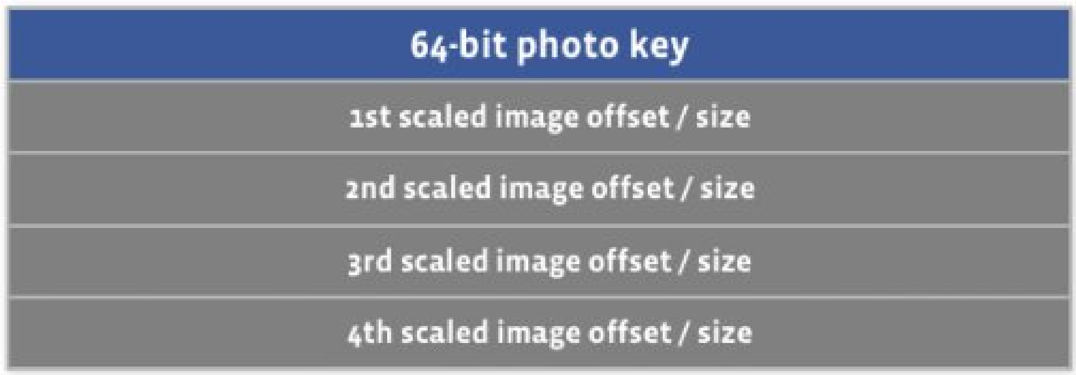
The parameters passed to the haystack read operation include the needle offset, key, alternate key, cookie and the data size. Haystack then adds the header and footer lengths to the data size and reads the whole needle from the file. The read operation succeeds only if the key, alternate key and cookie match the ones passed as arguments, if the data passes checksum validation, and if the needle has not been previously deleted (see below).

Haystack Delete Operation

The delete operation is simple – it marks the needle in the haystack store as deleted by setting a “deleted” bit in the flags field of the needle. However, the associated index record is not modified in any way so an application could end up referencing a deleted needle. A read operation for such a needle will see the “deleted” flag and fail the operation with an appropriate error. The space of a deleted needle is not reclaimed in any way. The only way to reclaim space from deleted needles is to compact the haystack (see below).Photo Store Server is responsible for accepting HTTP requests and translating them to the corresponding Haystack store operations. In order to minimize the number of I/Os required to retrieve photos, the server keeps an in-memory index of all photo offsets in the haystack store file. At startup, the server reads the haystack index file and populates the in-memory index. With hundreds of millions of photos per node (and the number will only grow with larger capacity drives), we need to make sure that the index will fit into the available memory. This is achieved by keeping a minimal amount of metadata in memory, just the information required to locate the images.

When a user uploads a photo, it is assigned a unique 64-bit id. The photo is then scaled down to 4 different sizes. Each scaled image has the same random cookie and 64-bit key, and the logical image size (large, medium, small, thumbnail) is stored in the alternate key. The upload server then calls the photo store server to store all four images in the Haystack.

The in-memory index keeps the following information for each photo:


64-bit photo key
1st scaled image offset / size
2nd scaled image offset / size
3rd scaled image offset / size
4th scaled image offset / size

Haystack uses the open source Google sparse hash data structure to keep the in-memory index small, since it only has 2 bits of overhead per entry.

Photo Store Write/Modify Operation

A write operation writes photos to the haystack and updates the in-memory index with the new entries. If the index already contains records with the same keys then this is a modification of existing photos and only the index records offsets are modified to reflect the location of the new images in the haystack store file. Photo store always assumes that if there are duplicate images (images with the same key) it is the one stored at a larger offset which is valid.

Photo Store Read Operation

The parameters passed to a read operation include haystack id and a photo key, size and cookie. The server performs a lookup in the in-memory index based on the photo key and retrieves the offset of the needle containing the requested image. If found it calls the haystack read operation to get the image. As noted above haystack delete operation doesn't update the haystack index file record. Therefore a freshly populated in-memory index can contain stale entries for the previously deleted photos. Read of a previously deleted photo will fail and the in-memory index is updated to reflect that by setting the offset of the particular image to zero.

Photo Store Delete Operation

After calling the haystack delete operation the in-memory index is updated by setting the image offset to zero signifying that the particular image has been deleted.

Compaction

Compaction is an online operation which reclaims the space used by the deleted and duplicate needles (needles with the same key). It creates a new haystack by copying needles while skipping any duplicate or deleted entries. Once done it swaps the files and in-memory structures.The HTTP framework we use is the simple evhttp server provided with the open source libevent library. We use multiple threads, with each thread being able to serve a single HTTP request at a time. Because our workload is mostly I/O bound, the performance of the HTTP server is not critical.Haystack presents a generic HTTP-based object store containing needles that map to stored opaque objects. Storing photos as needles in the haystack eliminates the metadata overhead by aggregating hundreds of thousands of images in a single haystack store file. This keeps the metadata overhead very small and allows us to store each needle's location in the store file in an in-memory index. This allows retrieval of an image's data in a minimal number of I/O operations, eliminating all unnecessary metadata overhead.

Peter Vajgel, Doug Beaver and Jason Sobel are infrastructure engineers at Facebook.

Excerpted from *Needle in a haystack: efficient storage of billions of photos*
https://www.facebook.com/note.php?note_id=76191543919