finding a needle in a Haystack

Facebook's photo storage

presented by:
Angelos Sakellaropoulos, Christos Mantas
Images, billions of them

- 65 billion photos (x4 different sizes = $260 \times 10^{12}$)
- more than 20 petabytes of data (total, for now)
- current growth rate: 220M photos per week (25TB weekly)
- At the peak: 550,000 images served per second
- biggest photo sharing site in the world
- Significant challenge for the storage infrastructure
Design Goals

- **High throughput and low latency**
  - Ignored requests are bad user experience
  - CDNs (like Akamai) are expensive

- **Fault tolerance**
  - Large scale availability needs redundancy (Datacenters spread in distinct geographical locations)

- **Cost effectiveness**
  - In Haystack, each TB costs ~28% less and processes ~4x more reads per second than NAS

- **Simplicity**
  - Keep it simple to implement, deploy and maintain

- **Data read often, never modified, rarely deleted**
Stated Contributions

- Design of Haystack
- Learned lessons in building and scaling an inexpensive(?), reliable and available photo storage system
- A characterization of the requests made to Facebook’s photo sharing app.
NFS based Storage Design
Step-by-Step ("Traditional")

1. GET request for the webpage to webserver
2. ->returns the markup (html only -incl links)
3. Request for the image to (external) CDN
4. request to FB's Photo Store Server for the image (by the CDN, if not in its cache)
5. Store Server requests the image file from the NAS machine via NFS.
6. multiple IOPS occur to retrieve and return the image
7-8 image is returned back
Disk ops on a POSIX FS (NFS)

- File metadata (permissions etc.) must be fetched from disk to read the file itself.

- Several disk operations for a single photo
  Steps 5 and 6 are really a few operations:
    i. read directory entry to get the inode No
    ii. read inode (100s of bytes)
    iii. read the file itself

- So, limiting factor for read throughput
Observation: excessive number disk ops for metadata lookups

Insight: Metadata could be looked up from RAM

Result: Higher performance with less cost

So why not just cache it in RAM? (or use a CDN)
The Long Tail
(is why one cannot just use cache)

- CDNs are good in serving a few "hot" pictures
- For a social networking website this will not do
- Requests are often for less popular, older content (pictures), aka "The Long Tail"
- CDNs can't serve those cheaply
- This also means that caching metadata (let alone pictures) is **not cost-effective** because one would need to keep all i-nodes in RAM (and i-nodes are too big to allow that)
Haystack, the big picture: Store - Directory - Cache

Haystack Directory

Web Server

Browser

Haystack Store

Haystack Cache

CDN

1 4

2 3

5 10

6 9

7 8

direct
Enter "the Needle"

- A file per picture means an i-node's worth of metadata - too much
- **So let's pack a lot of pictures (needles) in each (large) file**
  - log-structure approach
- This big file is called the "physical volume" (100GB) (a.k.a "Haystack File")
- the "logical volume" is an abstraction for replicas stored in different machines
finding a Needle in Haystack

• in each Store Machine, files stored as: /hay/haystack_<logical volume ID>
• very few i-nodes (physical files) per Store Machine
• A photo can be accessed quickly using
  ○ the ID of the volume
  ○ the offset at which the photo resides
• claimed: read data operation ~1 IOP (except corner-cases)
Anatomy of a Volume
Caching Metadata hay-style

per Volume, Store Machines need to keep in RAM:

- its file descriptor (i-node)
- a data structure that maps:
  \{key, alt_key\}->$\{flags, size, offset\}
  - key: ID of the user's image
  - alternate key: images type (for different sizes)
  - flags: deleted status
  - offset, size: the picture's offset in the volume & its size
  - reconstructed on reboot

- 32+ bytes per photo (8 bytes/ scaled image vs. 536B inode)
Haystack Store: Interface

(Kept it simple)

- **Read:** Request for \{volumeID, key, alt_key, cookie\}
  - lookup metadata (in RAM). If not deleted, seek in file and read needle. Verify cookie, integrity. Return.

- **Write:** web server provides:
  - \{volumeID, key, alt_key, cookie\}
    - append only store (modify Needle by storing a new version with same key. Latest is used)

- **Delete:** synchronously set the "deleted" flag in memory and physical volume
  - later reads return error
  - space lost for the moment
Haystack Store: optimizations

- **Multiwrite (Modify)**
  - Appends needles asynchronously one by one to a Volume
  - Asynchronously append index records to the index file (see later)
  - used instead of "write"

- **Compaction**
  - Online operation, rarely performed
  - Creates a fresh copy of a volume with too many duplicates and deleted needles
The Index file (speeds up reboots)

- Structure parallel to the Volume.
- Used to rebuild memory mappings.
- It is updated asynchronously so that write and delete operations return faster.
- Thus, needles might exist that are not in the index file (orphans):
  - Start from last record of the volume that's indexed and sequentially retrieve the rest of the needles.
- Indexed records might be deleted: examine flag on retrieval.
Haystack Store: Storage

- 2 x quad-core CPUs
- 16 GB-32 GB memory
- Hardware raid controller with 256MB-512MB NVRAM
- 12+1TB SATA drives
- Hatstack deployed on commodity storage blades
  around 10 TB of usable space
- RAID-6 provides adequate redundancy and excellent
  read performance while keeping cost down
- The poor write performance is partially mitigated by the
  RAID controller NVRAM-back cache
- Disk caches are disabled in order to guarantee data
  consistency in the event of crash or power loss
Haystack uses XFS, (an extent-based journaling file system) on top of 10 TB volume because:

(a) The block maps for several contiguous large files can be small enough to be stored in main memory in contrast with block based file systems which may require several I/Os.

(b) XFS provides efficient file pre allocation by aggressively allocating a large chunk of space whenever growing the physical file (1GB extent)
1. Maintains mapping (from logical volumes to physical volumes) used for constructing image URLs and uploading photos
2. Balances writes across logical volumes and reads across physical volumes
3. Determines whether a photo request should be handled by CDN or by the Cache (how?)
4. Identifies volumes which are read-only
   - Machine is marked read-only for operational reasons or when it exhausts its capacity
Haystack Cache

- If it cannot respond, it fetches the photo from the Store and responds either to the user or the CDN
- Functions as an internal CDN
- A newly retrieved photo is cached if
  (a) Request comes directly from a user and not the CDN
      - Post CDN caching is not effective
  (b) Photo is fetched from a user and not the CDN
      - write-enabled Store machines photos are most heavily accessed soon after uploaded
      - Haystack performs better when doing either reads or writes
Haystack, how it goes
Step-by-Step (Haystack)

1. GET request to Web-Server
2. WS looks up photo in Haystack Directory
3. Hay.Dir returns a link to the photo. Format:
   http://[ CDN /] Cache / Machine_id /{ Logical_volume, Photo}
   ● Note the generic load balancing
4. Web Server returns markup with photo links
5. Browser requests photo from CDN (or Cache)
6. CDN falls back to FB's Haystack (or returns)
7. Cache falls back to Haystack Store (or returns)
8. Store returns the photo (single disk op) ...
Haystack Optimizations

- Compaction
  - Infrequent online operation
  - Create a copy of haystack skipping duplicates and deleted photos
  - The pattern of deletes is similar to photo views, young photos are a lot more likely to be deleted
  - Over a year about 25% of the photos got deleted
Haystack Optimizations

- Saving More Memory:
  - Eliminate the need for an in-memory representation of flags by setting the offset to be 0 for deleted photos
  - Store machine does not keep track of cookie values in main memory and instead checks the supplied cookie after reading from the disk
  - With the above techniques store machines reduce their main memory footprints by 20%
Batch upload - Multiwrite (log structured file system idea):
- Disks perform better with large sequential writes instead of small random writes
- Many users upload entire albums to Facebook instead of each picture separately which gives an opportunity for batch uploads
CDF shows that caching at both CDNs and Cache can be very effective for hosting popular content.

The long tail implies that a significant amount of photos cannot be dealt with caching.
## Evaluation (Traffic)

- Haystack responds to 10% of photos viewed.
- Haystack photos written are 12 times the number of photos uploaded.
- Smaller images account for most of the photos viewed.

<table>
<thead>
<tr>
<th>Operations</th>
<th>Daily Counts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Photos Uploaded</td>
<td>~120 Million</td>
</tr>
<tr>
<td>Haystack Photos Written</td>
<td>~1.44 Billion</td>
</tr>
<tr>
<td>Photos Viewed</td>
<td>80-100 Billion</td>
</tr>
<tr>
<td>[Thumbnails]</td>
<td>10.2%</td>
</tr>
<tr>
<td>[Small]</td>
<td>84.4%</td>
</tr>
<tr>
<td>[Medium]</td>
<td>0.2%</td>
</tr>
<tr>
<td>[Large]</td>
<td>5.2%</td>
</tr>
<tr>
<td>Haystack Photos Read</td>
<td>10 Billion</td>
</tr>
</tbody>
</table>

Table 3: Volume of daily photo traffic.
Evaluation (Cache)

- high hit rates approximately 80% because of relatively recent photos
Evaluation (Directory)

- the graph shows the number of multi-write operations seen by 9 different Store machines
- since the lines are indistinguishable, directory balances writes well
Evaluation (Store) synthetic workload

- two benchmarks: Randomio (external) and Haystress (custom built)
- Haystack delivers 85% of the raw throughput of the device while incurring 17% higher latency (workload A: random read of 64KB)
- Multi-writes of 4 and 16 writes improves throughput by 30% and 78% respectively

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Random IO</td>
<td>[ Only Reads ]</td>
<td>902.3</td>
<td>33.2</td>
<td>26.8</td>
<td>-</td>
</tr>
<tr>
<td>Haystress</td>
<td>[ A # Only Reads ]</td>
<td>770.6</td>
<td>38.9</td>
<td>30.2</td>
<td>-</td>
</tr>
<tr>
<td>Haystress</td>
<td>[ B # Only Reads ]</td>
<td>877.8</td>
<td>34.2</td>
<td>28.1</td>
<td>-</td>
</tr>
<tr>
<td>Haystress</td>
<td>[ C # Only Multi-Writes]</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Haystress</td>
<td>[ D # Only Multi-Writes]</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Haystress</td>
<td>[ E # Only Multi-Writes]</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Haystress</td>
<td>[ F # Reads &amp; Multi-Writes]</td>
<td>718.1</td>
<td>41.6</td>
<td>31.6</td>
<td>6099.4</td>
</tr>
<tr>
<td>Haystress</td>
<td>[ G # Reads &amp; Multi-Writes]</td>
<td>692.8</td>
<td>42.8</td>
<td>33.7</td>
<td>7899.7</td>
</tr>
</tbody>
</table>
Evaluation (Store) production workload

- Write operations are always multi-writes
- Write enabled boxes see many more requests than read-only boxes
- As more data gets written to write-enabled boxes, the volume of photos increases resulting in an increase of number of reads
Evaluation (Store) production workload

- Multi-write latency fairly low (1 and 2 ms) and stable (to variable traffic)
- Reads on a read-only box latency fairly stable;
- Reads on write-enabled boxes are a result of three factors
  (a) increase in the number of photos result to an increase of read traffic
  (b) photos are cahed on write-enabled machines
  (c) recently written photos are usually read back
Related Work

● Filesystems
Haystack takes after log structured file systems with the key differences that:
(a) Store machines write their data in such a way that they can efficiently serve reads once they become read-only
(b) the read request rate for older data decreases over time
- there are several papers on how to group metadata and files intelligently (i.e. embedded inodes, hybrid File system):
  - Haystack maintains metadata in main memory and users upload related photos in a bulk

● Object-Data Storage
- similar to the NASD File and Storage Manager
- better read throughput than OBFS which is 1/25 the size of XFS
Related Work

● Managing Metadata
- idea: decoupling the mapping from logical to physical units, clients can calculate the appropriate metadata
- obstacle: less effective implementation on object stores because they lack the semantic groupings of directories
- solution: embed inter-object relations into the object id which is orthogonal to facebook social graph

● Distributed Systems
- Boxwood project high level data structures (use of a variant of B tree) may not have high impact on Haystacks' interface and semantics
- Sinfonia mini-transactions, PNUT's database (from Yahoo) provide much more complicated features and guarantees than Haystack needs
- GFS, Big Table: unclear if they are optimized for photo storage as Haystack
Conclusion

- Haystack presents an object data store containing needles that map to stored objects.
- Storing photos eliminates metadata overhead by aggregating images in a single file and stores each needle's location in an in-memory index.
- Image retrieval with a minimal number of I/O operations.
- Haystack uses cheap storage for a simple, fault tolerant and effective storage system.
- Future work:
  - Software RAID-6
  - Limit dependency on external CDN
  - Push recently uploaded photos into the cache
  - Calculate the metadata on the client side