Finding a needle in Haystack:
Facebook’s photo storage

OSDI 2010

Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, Peter Vajgel
Facebook.com

An Overview Presented in CSE-291 (Cloud Computing), Fall 2016
Gregory Kesden
Overview

• Facebook is the world’s largest photosharing site
• As of 2010:
  • 260 billion images
  • 20 petabytes of data
  • 1 billion photos, 60 terabytes uploaded each week
  • Over 1 million images/second served at peak
  • News feed and albums are 98% of photo requests
• Images are written once, read often, never modified, and rarely deleted
Why Not Use a Traditional Filesystem?

- No need for most metadata (directory tree, owner, group, etc)
  - Wastes space
  - More importantly, slows access to read, check, and use it
  - Turned out to be bottleneck
- What cost? Seems small?
  - Multiplied a lot
  - Think about steps: Map name to inode (name cache, directory files), read inode, read data
  - Latency is in access, itself, not tiny transfer
Haystack Goals

• High throughput, low latency
• Fault-tolerance
• Cost-effective (It is hugely scaled, right?)
• Simple (Means matures to robustness quickly, low operational cost)
Typical Design

Figure 1: Typical Design
“Original” Facebook NFS-based Design

Figure 2: NFS-based Design
Lessons Learned from “Original”

• CDNs serve “hottest” photos, e.g. profile pictures, but don’t help with “Long tail” of requests for older photos generated by such a large volume site
  • Significant amount of traffic, hitting backing store
  • Too many possibilities, too few used to keep in memory cache of any kind, not just via CDN

• Surprising complexity
  • Directories of thousands of images ran into metadata data inefficiencies in NAS, ~10 access/image
  • Even when optimized to hundreds of images/directory, still took 3 access: metadata, inode, file
Why Go Custom?

• Needed better RAM:Disk ratio
• Unachievable, because would need too much RAM
• Had to reduce demand for RAM by reducing metadata
Reality and Goal

• Reality: Can’t keep all files in memory, or enough for long-tail
• Achievable Goal: Shrink metadata so it can fit in memory
• Result: 1 disk access per photo, for the photo, itself (not metadata)
Figure 3: Serving a photo
Photo URL

- http://<CDN>/<Cache>/<Machine id>/<Logical volume, Photo>
- Specifies steps to retrieving the photos
  - CDN Looks up <Logical volume, Photo>. If hit, great. If not,
  - CDN strips <CDN> component, and asks the Cache. If Cache hits, great. If not,
  - Cache strips <Cache> component and asks back-end Haystack Store machine
  - If not in CDN just starts at second step.
Photo Upload

- Request goes to Web server
- Web server requests a write-enabled logical volume from the Haystack Directory
- Web server assigns unique ID to photo and uploads it to each physical volume associated with logical volume
Haystack Directory

• Functions:
  • Logical volume to physical volumes mapping
  • Load balances writes across logical volumes and reads across physical volumes
  • Determines if the request should be handed by CDN or Cache
  • Makes volumes read-only, if full, or for operational reasons (Machine-level granularity)
  • Removes failed physical volumes, replaces with new Store
  • Replicated database with memcache
Haystack Cache

- Distributed hash table with photo ID as key
- Cache photo iff
  - Request is from end user (not CDN) – CDN much bigger than cache. Miss there, unlikely to hit in smaller Cache.
  - Volume is write-enabled
    - Volumes perform better when reading or writing, but not mix, so doing one or the other is helpful
    - Shelter reads, letting focus on writes (No need to shelter, once volume is full – no more writes)
    - Could pro-actively push newly uploaded files into cache.
Haystack Store

- Store needs logical volume id and offset (and size)
  - This needs to quick to get, given the photo id – no disk operations
- Keeps open file descriptors for each physical volume (preloaded fd cache)
- Keeps in-memory mapping of photo ids to fs metadata (file, offset, size)
- Needle represents a file stored within Haystack
  - In memory mapping from <photoid, type (size)> to <flags, size, offset>
Needle and Store File

Figure 5: Layout of Haystack Store file

<table>
<thead>
<tr>
<th>Field</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Header</td>
<td>Magic number used for recovery</td>
</tr>
<tr>
<td>Cookie</td>
<td>Random number to mitigate brute force lookups</td>
</tr>
<tr>
<td>Key</td>
<td>64-bit photo id</td>
</tr>
<tr>
<td>Alternate key</td>
<td>32-bit supplemental id</td>
</tr>
<tr>
<td>Flags</td>
<td>Signifies deleted status</td>
</tr>
<tr>
<td>Size</td>
<td>Data size</td>
</tr>
<tr>
<td>Data</td>
<td>The actual photo data</td>
</tr>
<tr>
<td>Footer</td>
<td>Magic number for recovery</td>
</tr>
<tr>
<td>Data Checksum</td>
<td>Used to check integrity</td>
</tr>
<tr>
<td>Padding</td>
<td>Total needle size is aligned to 8 bytes</td>
</tr>
</tbody>
</table>

Table 1: Explanation of fields in a needle
Reads from Store

• Cache machine requests <logical volume id, key, alternate key, cookie>
• Cookie is random number assigned/maintained by Directory upon upload.
  • Prevents brute-force lookups via photo ids
  • Store machines looks this up in in-memory metadata, if not deleted
    • Seeks to offset in volume file and reads entire needle from disk
    • Verifies cookie and data integrity
    • Returns photo to cache
Photo Write

• Web server provides <logical volume id, key, type (size), cookie, data> to store machines (all associated with logical volume)

• Store machines *synchronously appends* needle to physical volume and updates mapping
  • The append makes this much happier
  • But, if files are updated, e.g. rotated, needle can’t be changed, new one must be appended – if multiple, greatest offset wins. (Directory can update for logical volumes)
Delete

• Just a delete flag – long live those party photos!
Index File

- Asynchronously updated checkpoint of in-memory data structures, in event of reboot
  - Possible to reconstruct, but much data would need to be crunched
  - Can be missing recent files and/or delete flags
- Upon reboot
  - Load checkpoint
  - Find last needle
  - Add needles after that from volume file
  - Restore checkpoint
- Store machines re-verify deleted flag after read form storage, in case index file was stale
Host Filesystem

- Store machines should use file system that:
  - Requires little memory for random seeks within a large file
  - E.g., blockmaps vs B-trees for logical to physical block mapping
Recovering From Failure

- Failure detection
  - Proactively test Stores: Connected? Each volume available? Each volume readable?
    - Fail? Mark logical volumes on store read only and fix.
    - In worst case, copy over from other replicas (slow = hours)
Optimizations

• Compaction
  • Copies over used needles, ignoring deleted ones, locks, and atomically swaps
  • 25% of photos deleted over course of a year, more likely to be recent ones

• Batch Uploads
  • Such as when whole albums uploaded
  • Improves performance via large, sequential writes
Cache Hit Rate
Why Did We Talk About Haystack

- All the fun bits
  - Classical Browser-Server-Directory-CDN-Cache-Store Layering
- Simple, Easy Example, By Design (Simple = Robust Fast)
- Optimizes for use cases
- Memory-Disk Trade
- Focus on fitting metadata into memory
- Real-world storage concerns