Lecture 10 : Cloud Storages - Dynamo, BigTable, Haystack

Dr. Bibhas Ghoshal

Assistant Professor

Department of Information Technology

Indian Institute of Information Technology Allahabad

Amamzon’s Dynamo

DynamoDB is a NoSQL database service for latency-sensitive applications that need consistent access at any scale.

- Fully-managed database service designed to provide an always-on experience (responsive even when nodes fail).

- Supports both document and key-value store (distributed) models and has been used for mobile, web, gaming, IoT, advertising, real-time analytics, and other applications
  - Map a key to a blob of in-structured data, stored across multiple nodes

- Stores data on SSDs to support latency-sensitive applications; typical requests take milliseconds to complete.
  - highly scalable (throughput scales with increasing nodes)

- Allows developers to specify the throughput capacity required for specific tables within their database using the provisioned throughput feature to deliver predictable performance at any scale

- Weak consistency (eventual consistency): ***GET*** may not always return the latest value ***PUT*** in the past

  No atomicity, isolation, or consistency (ACID of RDBMS)
  GET may also return multiple conflicting values
  Suitable for applications that can tolerate inconsistencies (e.g., shopping cart)
  Building block for many Amazon services (S3, DynamoDB)

System Architecture

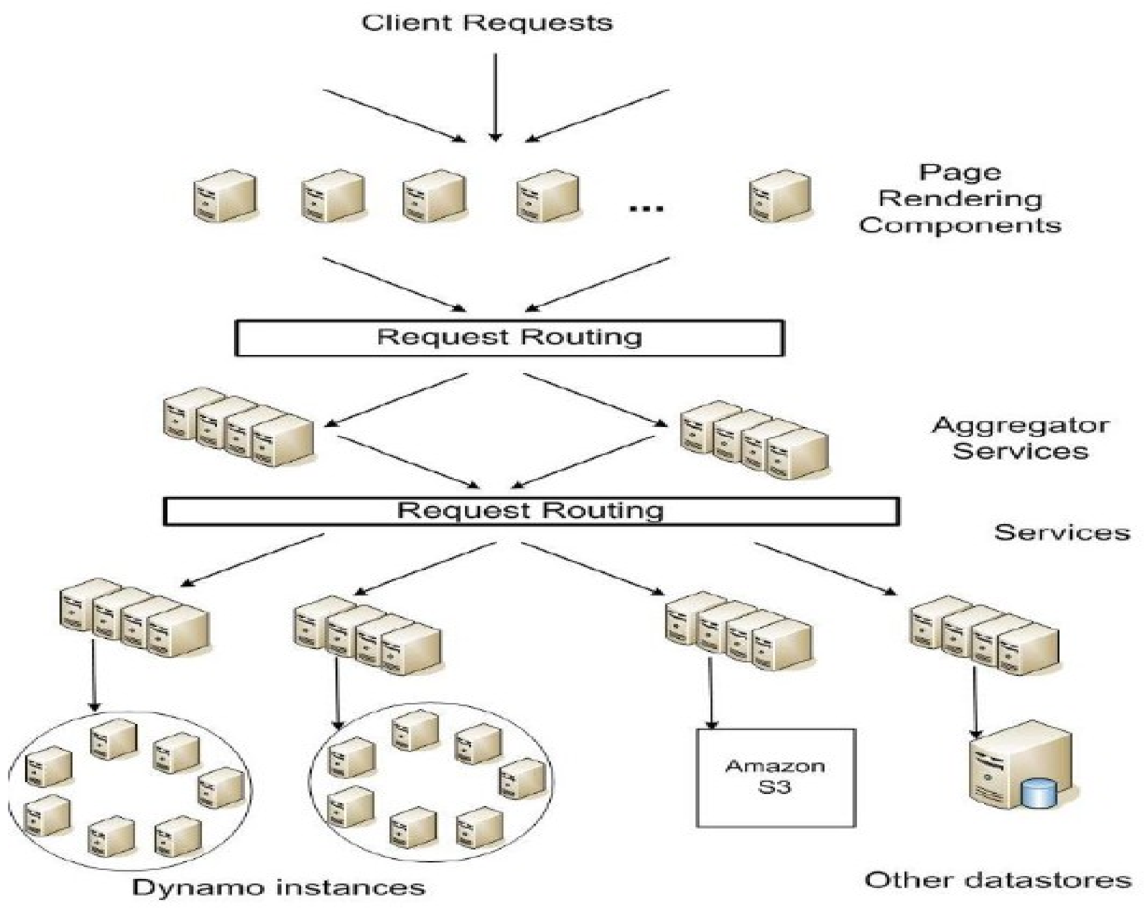


Figure 1: Service-oriented architecture of Amazon's platform

Dynamo DB Design Concerns and Solutions

- High availability is the primary concern. Updates should not be rejected even in the wake of network partitions or server failures.
- Deliver predictive performance in addition to reliability and scalability.
- The services supported have stringent latency requirements and this precludes supporting ACID properties.
- Strong consistency and high data availability cannot be achieved simultaneously.
- Availability can be increased by optimistic replication, allowing changes to propagate to replicas in the background, while disconnected work is tolerated.
- In traditional data stores writes may be rejected if the data store cannot reach all, or a majority of the replicas at a given time.
- Rather than implementing conflict resolution during writes and keeping the read complexity simple, Dynamo increases the complexity of conflict resolution of read operations.

Techniques to Achieve Design Objective

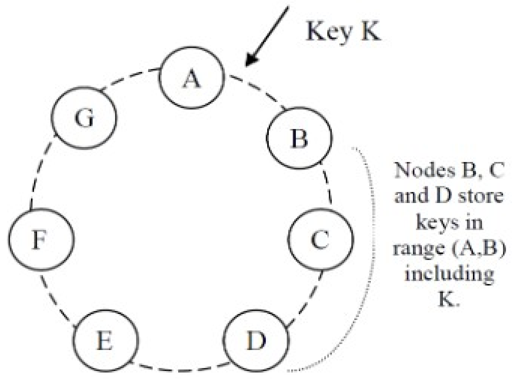
- Incremental scalability ensured by consistent hashing.
- High write availability based on the use of vector clocks with reconciliation.
- Handling temporary failures using sloppy quorum and hinted handoff. This provides high availability and durability guarantees when some of the replicas are not available
- Permanent failure recovery based on anti-entropy and Merkle trees. The technique synchronizes divergent replicas in background.
- Gossip-based membership protocol and failure detection for membership and failure detection. The advantage of this technique is that it preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.

Key Idea of Dynamo

- Dynamo partitions the keys over the set of nodes using consistent hashing
  - Every key is stored at a subset N of the total nodes (“preference list” of a key)
- Shared-nothing architecture: each replica independently stores state
  - System can scale by adding more nodes
- Put operation: the key is written to a subset W of the N nodes
  - Succeeds even if some subset of nodes are unavailable
- Get operation: the key is read back from some subset R of the N nodes
  - Eventual consistency: get may not return latest put
  - Multiple values can be returned, application has to reconcile
- Dynamo chooses R,W,N such that R+W > N, so that the latest value can be returned most of the times
  - Quorum protocol
  - R,W chosen to be less than N in order to achieve good latency

Server Organization

- The servers of the DynamoDB service are organized as a ring. Ring nodes B, C, and D store keys in range (A,B), including the key k.



- A data item identified by a key is assigned to a storage server by hashing the data item key to yield its position on the ring, and then walking the ring clockwise to find the first node with a position larger than the position of the item :
- A key is stored at the first N nodes which succeed the hash of the key in the circular ring (preference list)

- A storage server is responsible for the ring region between itself and its predecessor in the ring.

- First node on the list is the coordinator for the key
  - ***GET/PUT*** operations at all nodes managed by coordinator

Eventual Consistency

This strategy allows updates to be propagated to all replicas asynchronously.

- A versioning system allows multiple versions of a data object to be present in the data store at the same time. The result of each modification is a new and immutable version of data.
- New versions often subsume the older ones and the system can use syntactic reconciliation to determine the authoritative version.
- The system uses vector clocks, lists of (node, counter) pairs, to capture causality of each version of a data object.
- When a client updates an object, it must specify the version it is updating by passing the context it obtained from an earlier read operation, which contains the vector clock information.

Eventual Consistency

This strategy allows updates to be propagated to all replicas asynchronously – In case of ***PUT*** , coordinator does not wait for confirmation from all W nodes before sending a reply to the client

- A versioning system allows multiple versions of a data object to be present in the data store at the same time. The result of each modification is a new and immutable version of data. - A get after a put can find multiple versions of the key at different nodes
- New versions often subsume the older ones and the system can use syntactic reconciliation to determine the authoritative version.
- The system uses vector clocks, lists of (node, counter) pairs, to capture causality of each version of a data object.
- When a client updates an object, it must specify the version it is updating by passing the context it obtained from an earlier read operation, which contains the vector clock information.

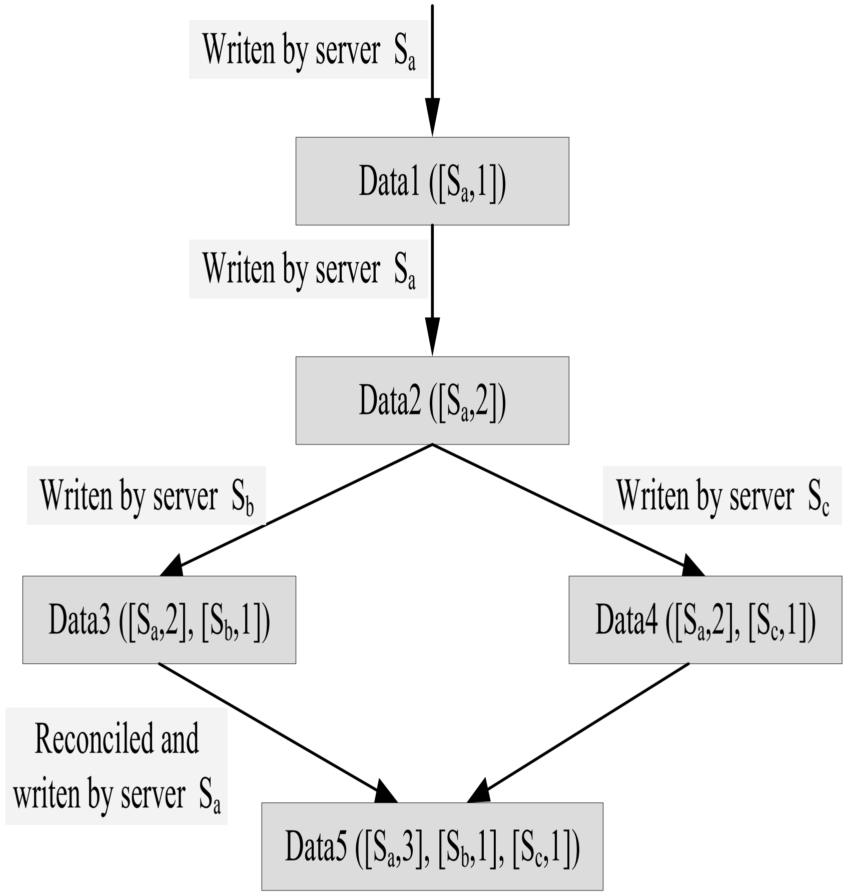
Versioning mechanism : Vector clocks

Since multiple versions of a key-value pair can exist, need some version number to track values

- Dynamo uses the idea of vector clocks to version the key-value pairs
- Vector clock is a set of (node, count) pairs, where the count is incremented locally at every node
- Every node that handles a key will add/increment its entry in the vector clock
- Vector clock version number associated with value (also called “context”) is returned with every get to the client, and the client sends it along with its next ***PUT*** request
  - object, context = get(key)
  - put(key, context, object)
- Suppose there are three nodes X, Y, Z handling a key
  - Suppose client gets a value from X with vector clock [(X, nx), (Y, ny), (Z, nz)]
  - Next put at X will increment the vector clock to [(X, nx+1), (Y, ny), (Z, nz)]
  - If put done at Y instead, vector clock will be [(X, nx), (Y, ny+1), (Z, nz)]

Versioning mechanism : Sequence of events Time Evolution

- Data is written by server S_a and the object Data1 with the associated clock $[S_a,1]$ is created.
- The same server S_a writes again and the object Data2 with the associated clock $[S_a,2]$ is created. Data2 is a descendent of Data1 and over-writes it. There may be replicas of Data1 at servers that have not yet seen Data2.
- The same client updates the object and server S_b handles the request; a new object, Data3, and its associated clock [(S_{a}, 2), (S_{b}, 1)] are created.
- A different client reads Data2 and then tries to update it, and this time server S_c handles her request. A new object, Data4, a descendent of Data2, with version clock is [(S_{a}, 2), (S_{c}, 1)] is created.
- Upon receiving Data4 and its clock, a server aware of Data1 or Data2 could determine, that both are overwritten by the new data and can be garbage collected.



Writen by server S_a
Data1 $([S_a,1])$
Writen by server S_a
Data2 $([S_a,2])$
Writen by server S_b
Writen by server S_c
Data3 $([S_a,2], [S_b,1])$
Data4 $([S_a,2], [S_c,1])$
Reconciled and writen by server S_a
Data5 $([S_a,3], [S_b,1], [S_c,1])$

Google's BigTable

Semi-structured data store: does not support full relational database model, simpler data format

- Borrows ideas from parallel and in-memory databases

- Tables of rows, columns (strings). Each cell also has timestamp. Maps row key, column key and timestamp to a value (array of bytes)

$$(\text{row:string}, \text{column:string}, \text{time:int64}) \rightarrow \text{string}$$

- Widely used by many systems at Google
  - Both batch processing and real time applications
  - Clients can control whether data on disk or in memory
- High availability, scalability, performance

Data Model

Rows: atomic unit of reading and writing data

- Rows sorted alphabetically (users should pick row names suitably)
- Group of adjacent rows is called a tablet (unit of distributing data to machines)

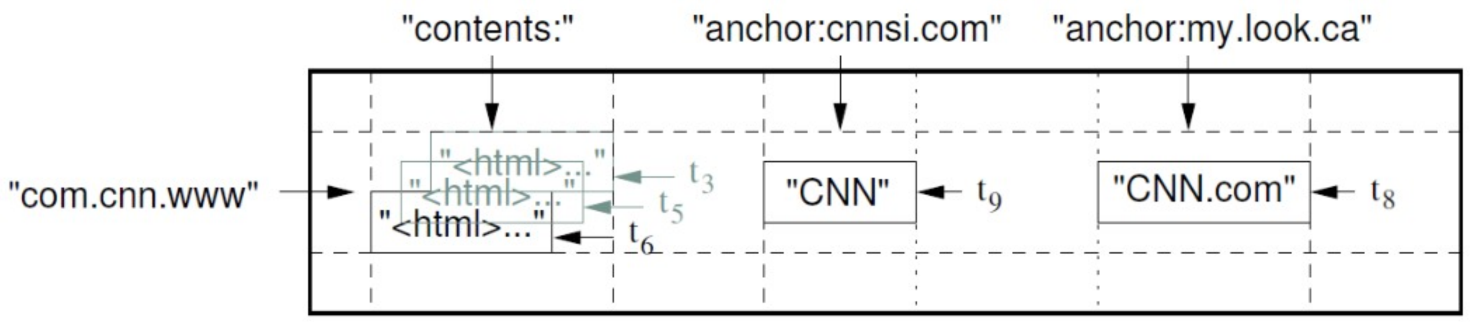
Columns: grouped into small no. of families

- All columns in a family store similar type of data, treated similarly
- Family is granularity for specifying access control, locality (disk vs memory) etc.

Timestamp of a cell acts as a version number and is provided by clients

- Last N versions stored

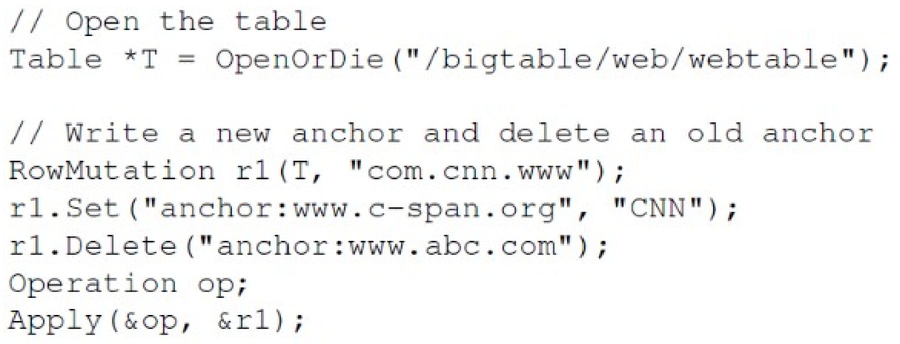
- Example Web table shown below



Big Table API

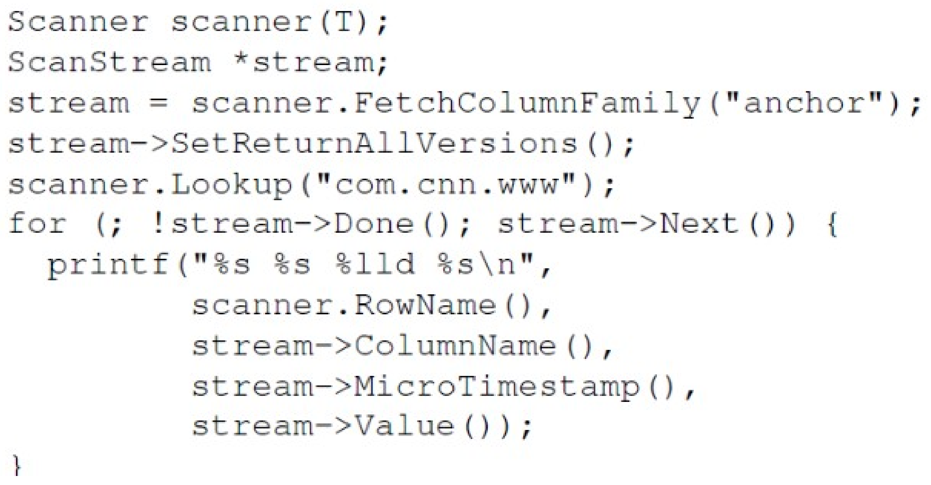
Bigtable's API allows users to create tables and manipulate various cells

- Examples of reading and writing tables



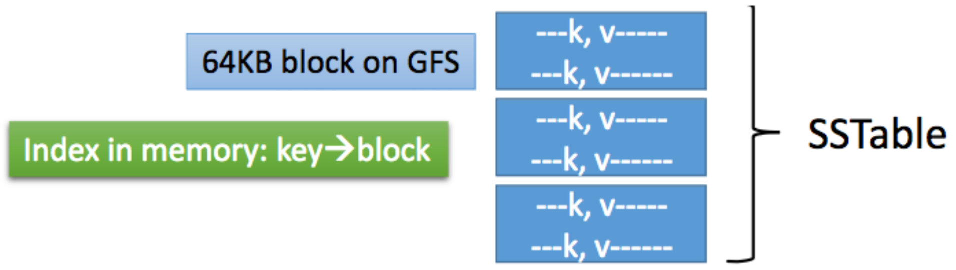
```
// Open the table
Table *T = OpenOrDie("/bigtable/web/webtable");

// Write a new anchor and delete an old anchor
RowMutation r1(T, "com.cnn.www");
r1.Set("anchor:www.c-span.org", "CNN");
r1.Delete("anchor:www.abc.com");
Operation op;
Apply(&op, &r1);
```



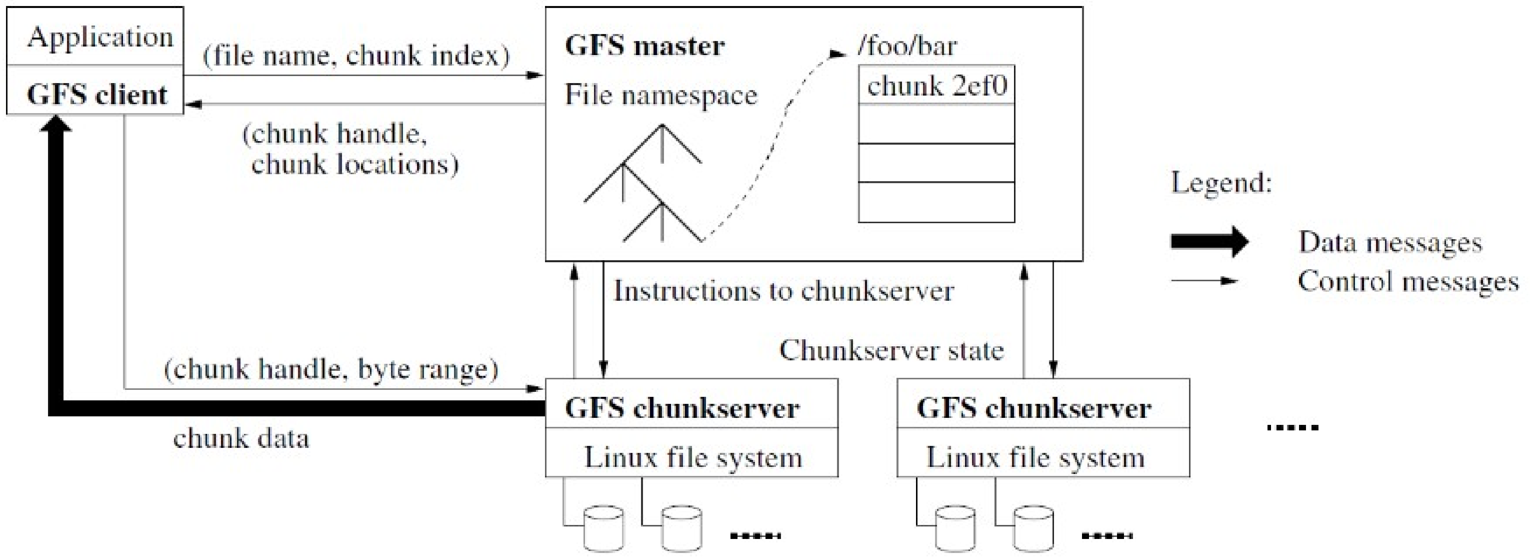
```
Scanner scanner(T);
ScanStream *stream;
stream = scanner.FetchColumnFamily("anchor");
stream->SetReturnAllVersions();
scanner.Lookup("com.cnn.www");
for (; !stream->Done(); stream->Next()) {
  printf("%s %s %lld %s\n",
          scanner.RowName(),
          stream->ColumnName(),
          stream->MicroTimestamp(),
          stream->Value());
}
```

Building Blocks



- Leverages two existing systems of Google: SSTable and Chubby
- SSTable: file format used to store Bigtable data internally
  - Immutable map of key-value pairs, stored on a sequence of blocks (64KB) on disk.
- Blocks stored on GFS (Google File System)
  - Can lookup a value using key, or iterate over all key-value pairs
  - Index of SSTable maps keys to disk blocks, loaded into memory when SSTable opened
  - Lookup only needs single disk access: lookup key in index to find block location, then access disk block
  - More efficient than storing files in other formats on regular filesystems

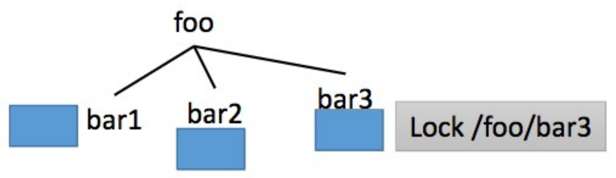
Building Blocks



Google File System: distributed file system on commodity hardware

- Designed to efficiently store a small number of large files (not POSIX API)
- GFS cluster has one master and multiple chunk servers (Linux machines)
- File divided into fixed size chunks, chunks replicated at multiple chunk servers
- Chunks stored at chunk servers on local disk, identified by a unique handle
- Master stores chunk handle chunk server mapping

Building Blocks



- Chubby: distributed lock service / distributed directory service
    - Exposes a namespace of directories and files
    - Users of chubby can acquire a lock on some directory/file and read/write its contents
- How is it useful?
    - Suppose an application is built to run at one node, needs to migrate to multiple nodes for fault tolerance. One way is to change app logic to run on multiple replicas, with some leader election/consensus algorithm among replicas – hard work!
    - Easier way: ask all replicas to lock a certain file using Chubby, whoever gets the lock is master. Master shares data with replicas via Chubby files
- Internally, Chubby implements Paxos across its 5 replicas, in order to consistently maintain replicated state of the lock namespace
    - Chubby runs Paxos, so that other apps do not have to reimplement the logic
- Clients maintain sessions with Chubby servers and send requests to acquire/release locks
    - If a session breaks (client fails), all locks held by a client are released.

Big Table Implementation

Components of the system:

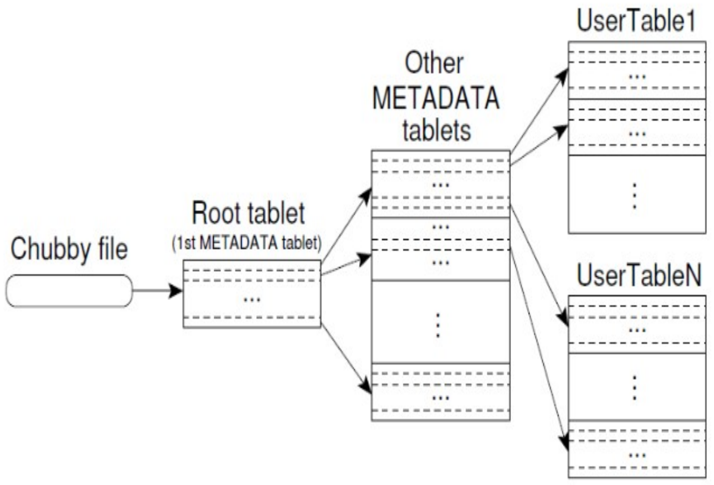
- Tablet servers: store few hundred tablets (group of rows of a Big table)
- Master: distributes tablets to tablet servers, load balancing, handles failures of tablet servers, creation/updation of table schema.
- Clients: read/write tablets at tablet servers

- How does master track tablet servers?
  - Every tablet servers create its own unique file on Chubby
  - Master periodically looks at these files to find list of healthy tablet servers, assigns each tablet to one tablet server
  - If tablet server fails, Chubby connection lost, file is deleted, master reassigns its tablets to other servers.
  - If master fails, it scans list of tablet servers, queries them for tablets, and reconstructs tablet assignment

Locating Tablets

Tablet locations stored in metadata tablets

- Locations of all metadata tablets stored in a root tablet
- Root tablet published in Chubby by master
- Clients cache locations of tablets: no need to query master
- No need to traverse the metadata tables
- Assume 1KB location information, 128 MB tablets
  Each tablet stores 128K = 2^17 locations
  Maximum filesize = 2^34 tablets = 2^61 bytes



Inside a Tablet Server

Persistent data is stored on GFS

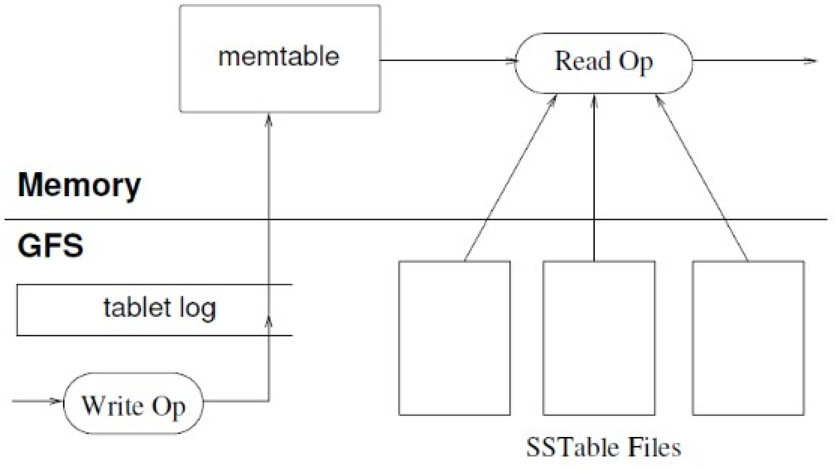
- GFS handles replication, so tablet can be just stored at one tablet server

- Each tablet server has multiple immutable SSTables and commit logs in GFS and a mem table in memory
  - List of all SSTables and commit logs of tablet server stored in metadata in Chubby

- Write operation:
  - Changes first written to a commit log on GFS
  - Recently committed writes are in memtable in memory
  - Old data is not deleted from SSTable, only deletion record written

- Read operation :

  Reads from merged view of mem table and SSTables
  - Latest value of a row is chosen from merged view
  - Easy to merge as all tables are sorted



Inside a Tablet Server

Why immutable SSTable? Simplifies design considerably

- Can be accessed concurrently without locks during reads and writes
- Only mem table needs concurrency control

- Compactions
  - Once enough data accumulates, memtable compacted into SSTable and stored in GFS
  - Periodically, multiple SSTables merged to create fewer SSTables (handling deletions)
- Separate locality groups created for each column family
  - Column families stored together in SSTables
  - SSTable of a column group can be specified to be on disk or in-memory
  - Compression format can be specified for a column group
- Caching at tablet server to improve performance
  - Recently read key-value pairs from SSTable are cached
  - Recently read blocks of SSTable are cached

Facebook’s Haystack

Some cloud storage systems are optimized for specific applications

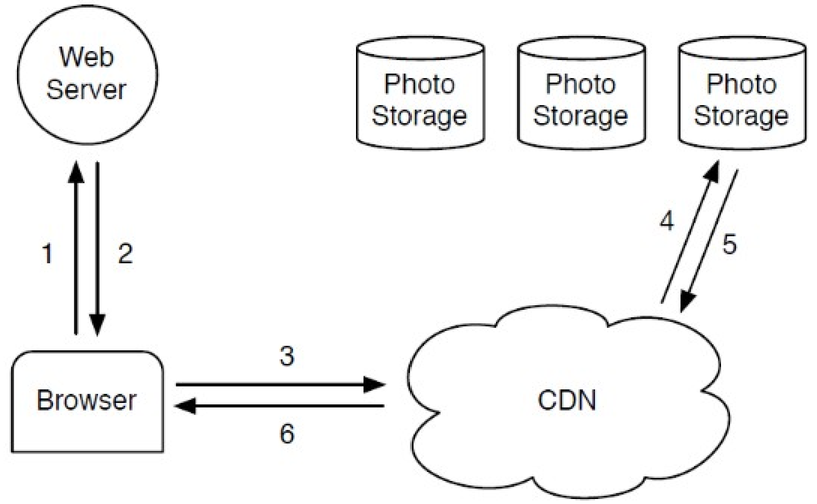
- Facebook’s Haystack is optimized for photo storage

- Why not store photos as regular files on a POSIX-compliant filesystem?
  - Many attributes like permissions are meaningless
  - Lot of metadata accesses (inodes) before actual photo access
  - App specific knowledge: photos are written once, read often, rarely modified or deleted
  - High throughput, low latency, fault tolerance, with cost-effectiveness

Design of Photo/Object Storage

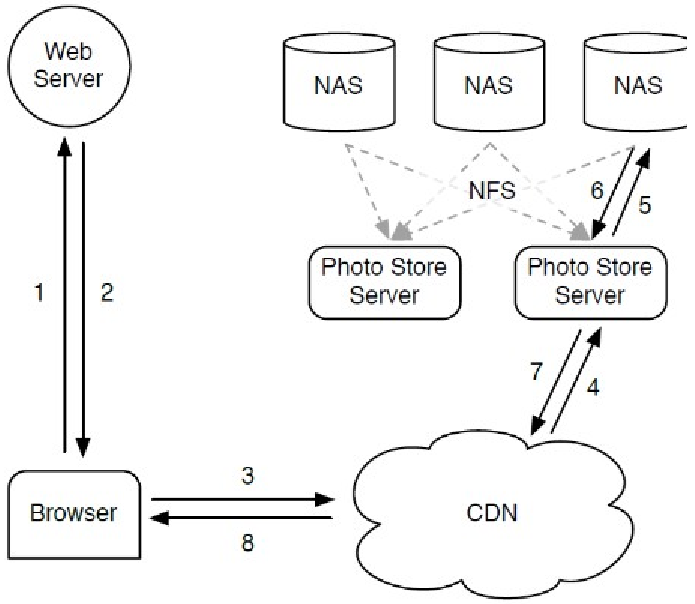
Photos and other read-only objects are served from Content Delivery Networks (CDNs), e.g., Akamai

- DNS redirects to geographically closest CDN servers
- If object cached in CDN, served directly from CDN
- Else, fetch object from original storage and serve, cache
- CDNs improve performance only for the hottest objects found in cache
- Photos have a "long tail": unpopular photos form significant part of traffic
- Need to optimize photo storage even if using CDN cache



NFS Based Photo/Object Storage

Each photo stored as a separate file on a commercial NAS (Network Attached Storage) box, served by NFS
• At least 3 disk accesses to read a photo from filesystem
• Get inode number of file by reading parent directory blocks), read inode block, then fetch actual file
• Large directories spread over multiple blocks
incur even higher overhead
• Can cache inodes in memory to save disk accesses, but too much memory consumed to store all inodes of all files



Motivation

Ideally, access photo directly on disk, without multiple disk accesses

- Metadata (inode) to locate photo on disk should be in memory
- However, caching all inodes for even unpopular photos is not possible

- Existing systems do not have the right “RAM-to-disk” ratio
  - Each photo as a separate file, each inode occupied ~100 bytes in memory
  - Too much memory for metadata in general purpose filesystems
- Goal: reduce metadata per photo, so that all metadata in memory, only one disk access even for unpopular photos
- Key idea: new filesystem, store multiple photos in large files, minimal metadata per photo
  - Redesigning filesystem is better than buying more NAS appliances / web servers / CDN storage

Haystack Architecture

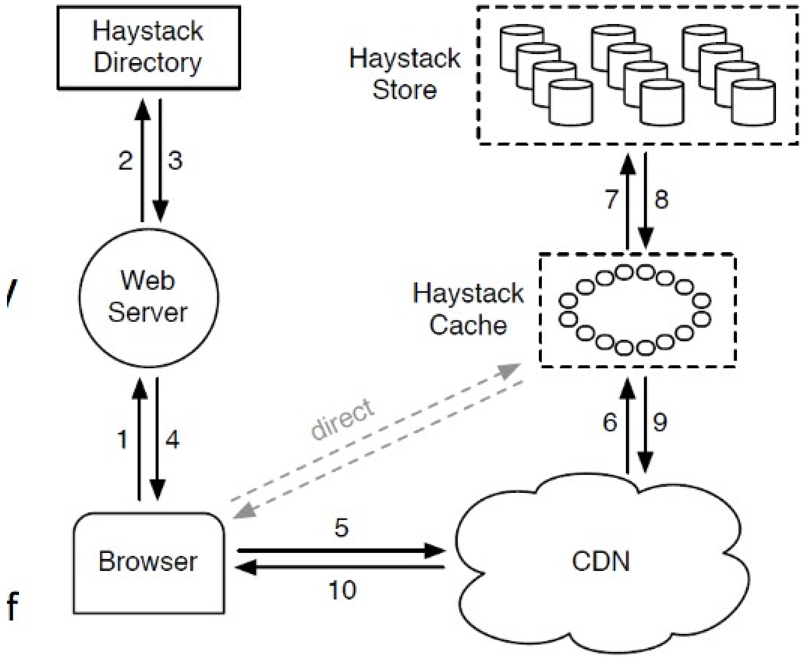
3 components: Store, Cache, Directory
- Store has the actual photos
- Each server has many physical volumes (disks) which are organized into logical volumes

- Cache caches popular content that is not already cached in CDNs
- Directory maintains location mapping (which CDN/cache/store/logical volume may have photo)
  - When user requests photo from Facebook's webserver, it looks up directory
  - Directory returns a URL which encodes the location of the photo:

CDN/Cache/Store/logical volume info

http://⟨CDN⟩/⟨Cache⟩/⟨Machine id⟩/⟨Logical volume, Photo⟩

- Can check in CDN first, or directly go to cache
- Balances load across store machines



Upload a Photo

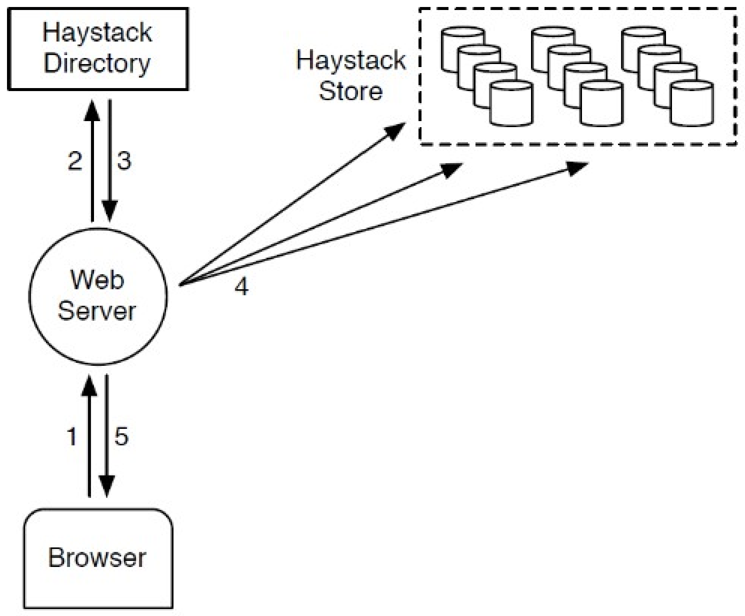
Upload path:

• Photo goes to webserver, which looks up Directory

• Directory returns the location of the Store server and logical volume where photo to be stored

• A logical volume is replicated at multiple physical volumes for resiliency

• Web server uploads photos at the multiple locations of a logical volume



Store Server Architecture

Each Store server has multiple physical volumes/disks

- Physical volume is a large file (~100GB) with millions of photos
- Each physical volume belongs to one of the logical volumes

- Logical volume = collection of different physical volumes on different servers
  - When a photo stored on a logical volume, it is replicated at all physical volumes of the logical volume, for resiliency
  - Directory has all info on physical and logical volumes on all store servers

- Photo identified by Store machine ID, logical volume, photo identifier/key
  - Go to server, find physical volume associated with logical volume, lookup photo
- New machine added to store: write-enabled, accepts uploads
  - Once capacity is full, moves to read-only mode, only serves photos
  - Cache mostly caches data from write-enabled store machines, because the most recently uploaded photos are frequently accessed by users

Store Server : Disk Layout

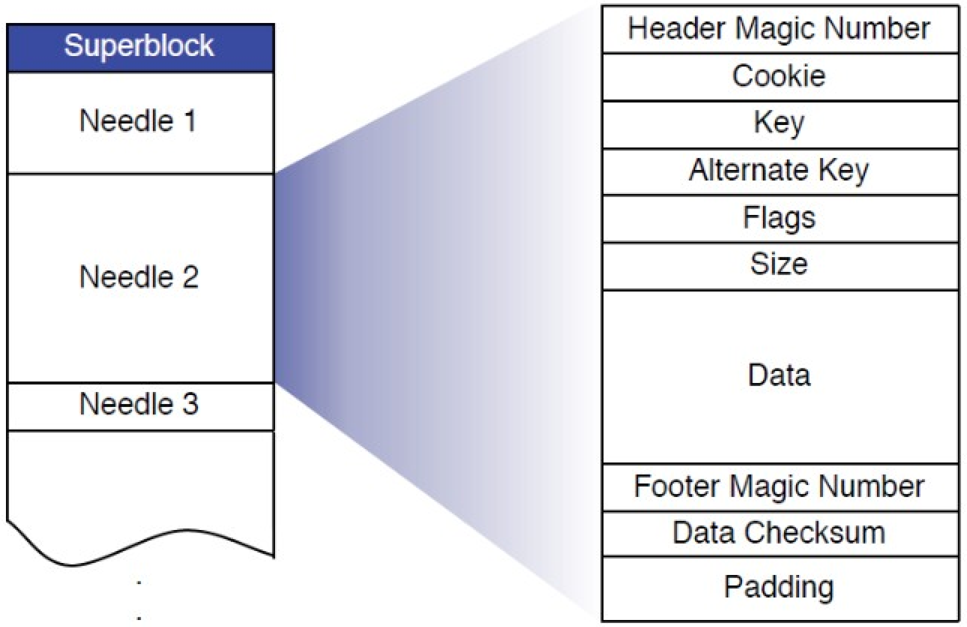
Each physical volume has a Super block and a set of “needles”

- A single file with all photos

- Needle = photo + all its metadata (key, alternate key, size, etc.)

- Alternate key is a way to distinguish multiple versions of a photo (e.g., different Resolutions)

- Large file stored on disk using existing filesystems (XFS)



Store Server : In Memory Data Structures

Store server has open file descriptor for each physical volume

- In-memory index mapping photo key/alternate key to offset within disk
  - Lower overhead than full-fledged inodes

- Read request: lookup photo’s key in index, find disk offset, read data from disk
  - Achieved goal of one disk access per photo

- Write request of new photo: appended to disk at end, index updated
- Modification/deletion of existing photo (rare): new copy appended at end, index updated to point to latest version
  - Modifications (e.g., rotations) have same key and alternate key
  - Old data not overwritten on disk for modifications or deletions, instead updated entry (or deletion record) is appended to disk
  - Periodic compactions of disk files to delete stale entries

Updating index file

Where is index stored? In theory, no need to store index on disk, reconstructed from disk data on booting

- If two entries on disk for same photo key (e.g., deletion or modification), index points to latest entry
- However, may take a long time for large disks

- Periodically checkpoint index into a file on disk for quick bootup:
  - Index file written to disk asynchronously after appending actual data to disk
  - If system crashes after updating actual data but before updating index, index file on disk may be stale
  - For example, we can have orphans (photos on disk without entry in index)
  - During bootup, start with index file, see latest entry in index, all disk recordsafter that are read and incorporated into index