Compute and data storage options

- Cloud compute runs on VMs or containers
  - Studied in course so far
- Cloud storage options
  - Relational databases (RDBMS) with strong consistency (ACID properties). Traditional storage option, but harder to scale, lower performance.
  - No-SQL data stores map key to unstructured “blob” of data. Simpler get/put interface using single key, no transaction support. Some form of weaker consistency. Tradeoff stronger guarantees for higher performance.
  - Semi-structured data stores map key to value which has certain column families, but not fixed schema. Useful for certain apps.
  - Other storage frameworks customized for application needs

Cloud storage systems: examples from real-life

- Dynamo: high-performance No-SQL key-value store from Amazon
  - Trade off consistency guarantees for high availability and scalability
- Bigtable: semi-structured data store used at Google
  - Built over two other systems from Google: Chubby (a distributed locking/consensus system) and the Google File System (a distributed file system)
- Haystack: cloud storage system for efficiently storing and retrieving photos at Facebook
- Memcache: distributed cache used at Facebook to cache queries from database servers

Amazon's Dynamo

- Dynamo is a distributed key-value store: simple get/put interface
  - Map a key to a blob of instructured data, stored across multiple nodes
  - Example of No-SQL data store (unlike traditional RDBMS)
- Highly available (responsive even when nodes fail), high performance (high throughput, low average/tail latency), highly scalable (throughput scales with increasing nodes)
- Weak consistency (eventual consistency): a get may not always return the latest value put in the past
  - No atomicity, isolation, or consistency (ACID of RDBMS)
  - A 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)
  - Traditional RDBMS is an overkill for such applications

Dynamo: Amazon's Highly Available Key-value Store
Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels

System architecture

- A service chain of web servers, application servers, data stores
- Aggregator services aggregate data from multiple applications
- Very important to keep even tail latency (e.g., 99.9 percetile latency) low

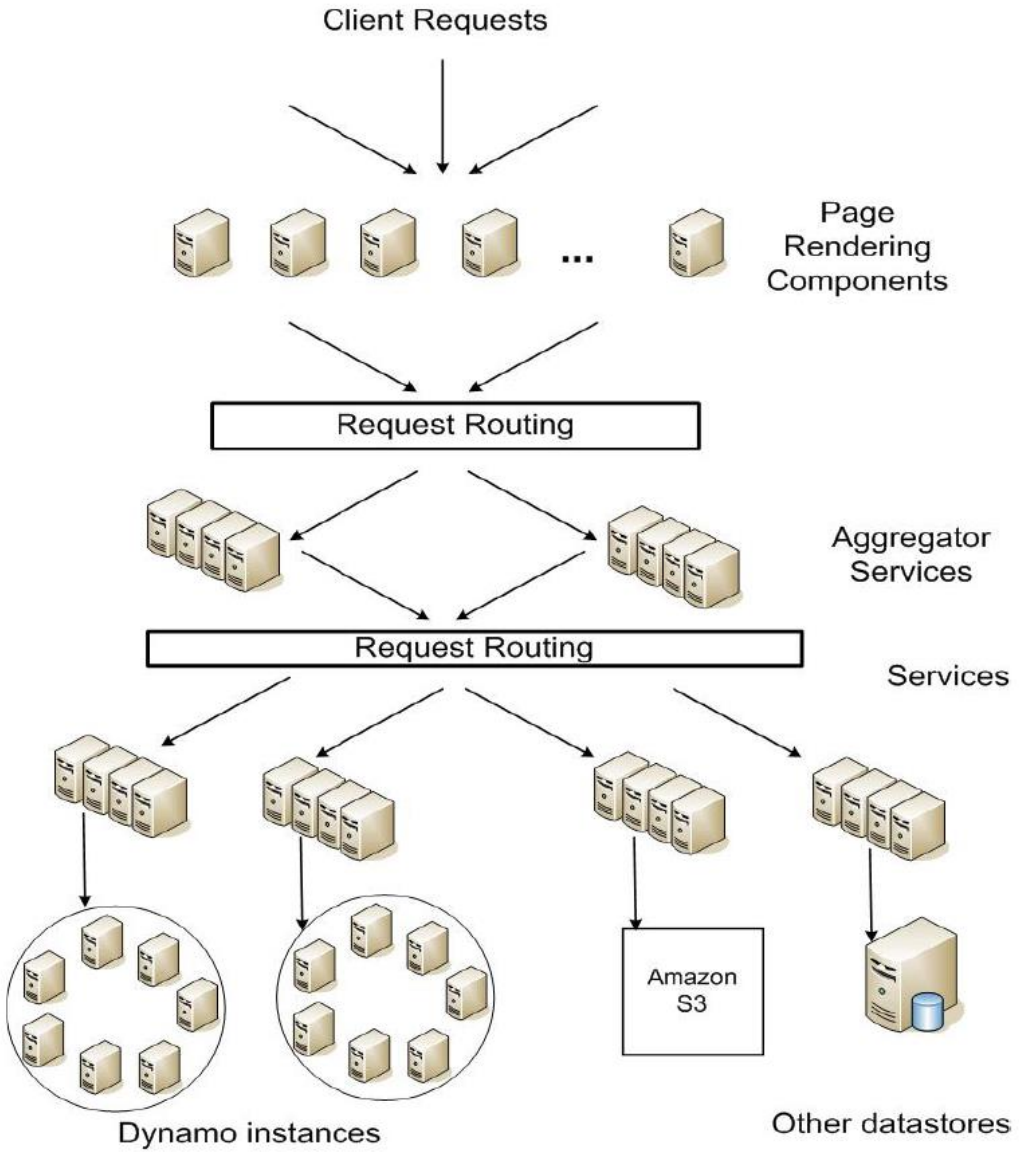


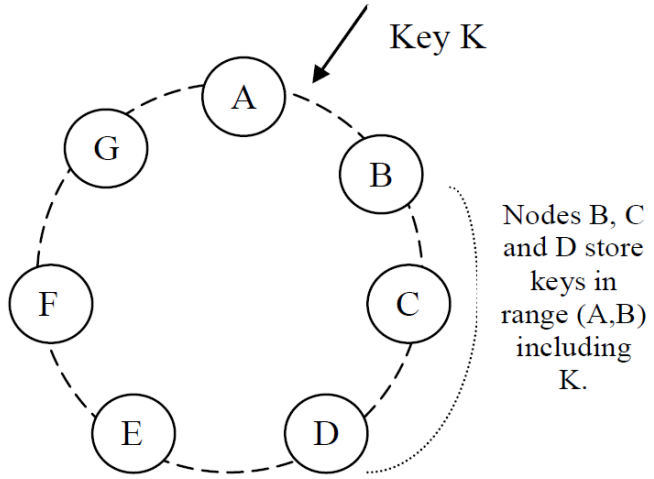
Figure 1: Service-oriented architecture of Amazon's platform

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

Assigning keys to nodes: Consistent Hashing

- Every key is hashed to generate a number in a circular range
- Every node/replica assigned an ID in the same space
- A key is stored at the first N nodes which succeed the hash of the key in the circular ring
  - Called “preference list” of the key
- First node on the list is the coordinator for the key
  - Get/put operations at all nodes managed by coordinator
- For better load balancing, every node is treated as multiple virtual nodes, assigned many positions on list
  - Preference list will contain N distinct physical nodes



Failures and eventual consistency

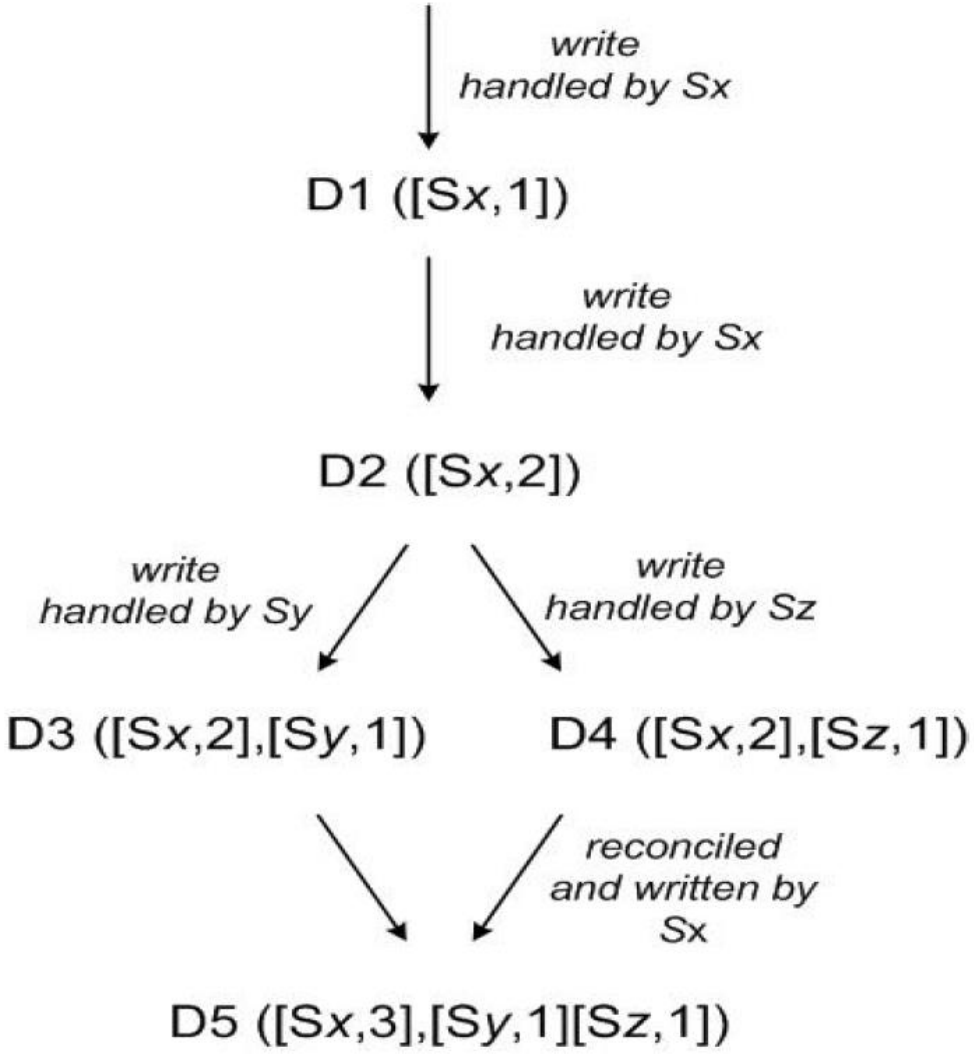
- In cases of node/network failures:
  - Preference list of first N nodes can change with failures, finds first N alive nodes ("sloppy" quorum)
  - Nodes in original preference list will be contacted and updated when they come back
- Put is asynchronous: coordinator does not wait for confirmation from all W nodes before sending a reply to the client
  - In case of failures, a put may not reach all W nodes
- A get after a put can find multiple versions of the key at different nodes
  - Consistency (get returning latest put) is only guaranteed "eventually"
- Why this design? Because one of the goals is to be always writeable
  - System should never turn down a write request from a client
  - Systems with strong consistency will turn down client requests in case of failures, and will only accept requests when they can guarantee consistency

Versioning: 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)]

Example of vector clocks

- A client gets D1, puts D2, Sx is the coodinator
  - D2 directly descends from D1, can overwrite D1
- Client reads D2, Sx is down, so client writes D3 at Sy. Another client reads D2 and writes D4 at Sz in parallel
  - Both D3 and D4 descend from D2
  - D3 and D4 can have conflicting changes, one does not overwrite the other
- On the next get, both D3 and D4 returned to client
  - Node performing get cannot decide which to return
  - Client must reconcile and arrive at new value (D5)
- Next put of D5 at Sx has combined vector clock, indicating reconciliation has happened
  - D5 can overwrite D3 and D4



Summary of key ideas

- Discussed in this lecture:
  - Consistent hashing to partition keys to nodes for scalability
  - "Shared nothing" architecture to scale over multiple nodes
  - Handle temporary failures via sloppy quorum, async writes
  - High availability by settling for weaker consistency guarantees
  - Leave it to application to reconcile inconsistencies using vector clocks
- More details in the paper:
  - Handling permanent failures
  - Membership changes

Table 1: Summary of techniques used in *Dynamo* and their advantages.

Problem	Technique	Advantage
Partitioning	Consistent Hashing	Incremental Scalability
High Availability for writes	Vector clocks with reconciliation during reads	Version size is decoupled from update rates.
Handling temporary failures	Sloppy Quorum and hinted handoff	Provides high availability and durability guarantee when some of the replicas are not available.
Recovering from permanent failures	Anti-entropy using Merkle trees	Synchronizes divergent replicas in the background.
Membership and failure detection	Gossip-based membership protocol and failure detection.	Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.

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)

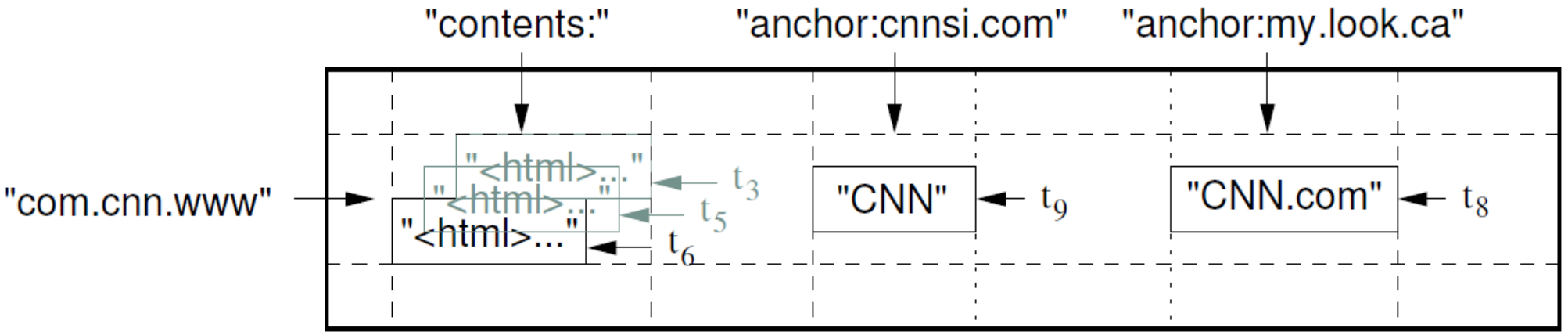
$$(\text{row:string}, \text{column:string}, \text{time:int64}) \rightarrow \text{string}$$

- 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

Bigtable: A Distributed Storage System for Structured Data
Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows
Tushar Chandra, Andrew Fikes, Robert E. Gruber

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 Webtable shown below



Bigtable API

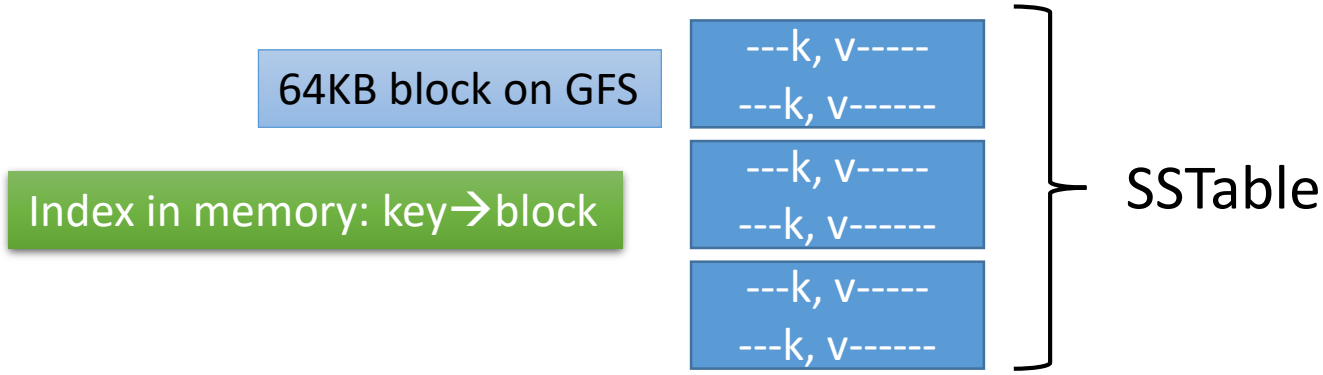
- Bigtable's API allows users to create tables and manipulate various cells
- Examples of reading and writing tables

```
// 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());
}
```

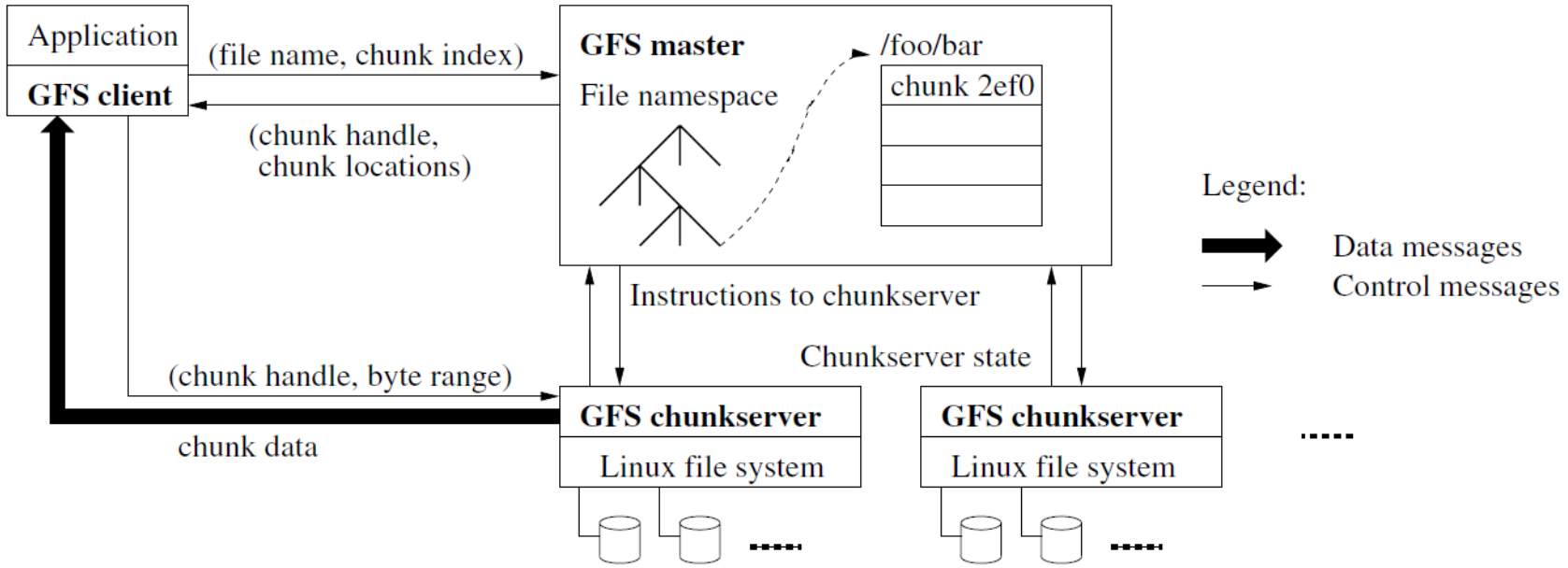
Building blocks (1)



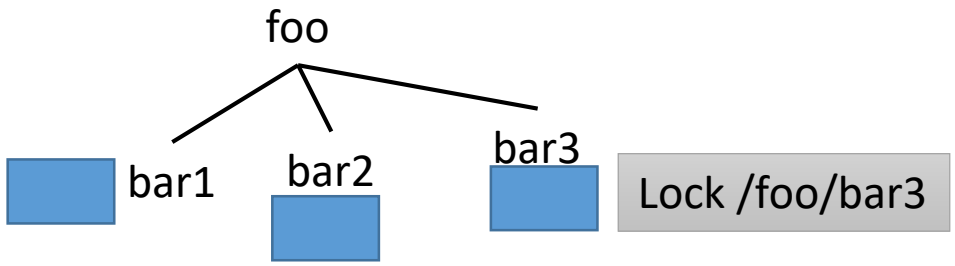
- Leverages two existing systems of Google: SSTable and Chubby
- SSTable: file format used to store Bigtable data internally
  - Immutable map of key-value pairs, stored on a sequence of blocks (64KB) on disk. Blocks stored on GFS (Google File System)
  - Can lookup a value using key, or iterate over 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 (2)

- 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 (3)



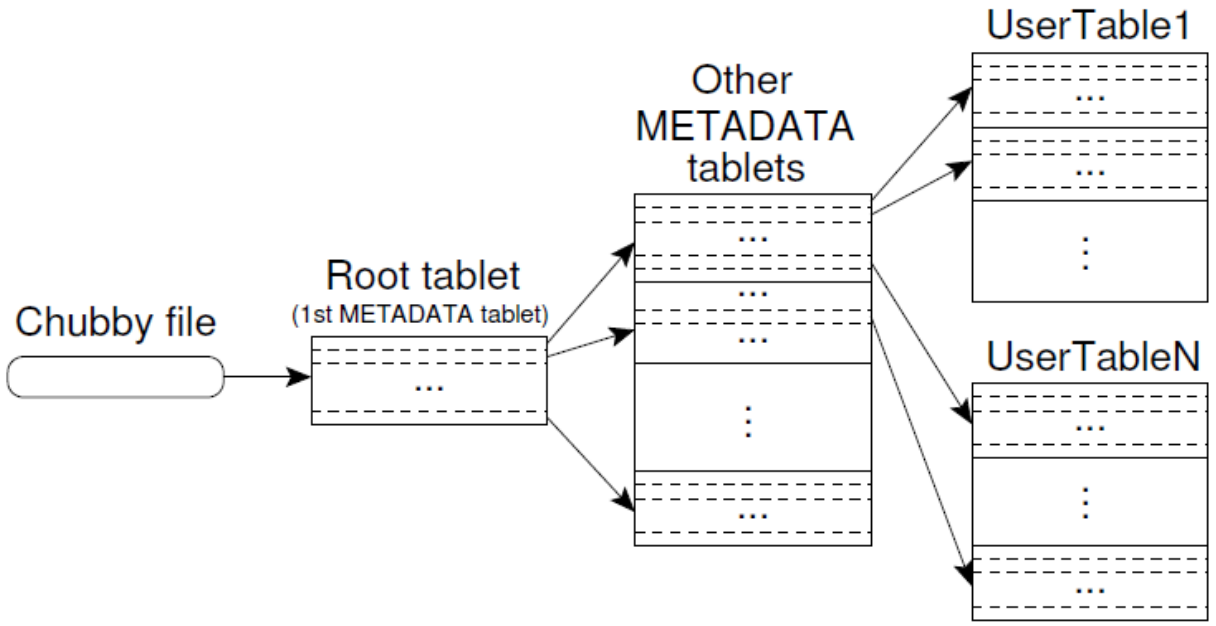
- Chubby: distributed lock service / distributed directory service
  - Exposes a namespace of directories and files
  - Users of chubby can acquire a lock on some directory/file and read/write its contents
- How is it useful?
  - Suppose an application is built to run at one node, needs to migrate to 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.

Bigtable Implementation

- Components of the system:
  - Tablet servers: store few hundred tablets (group of rows of a bigtable)
  - 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

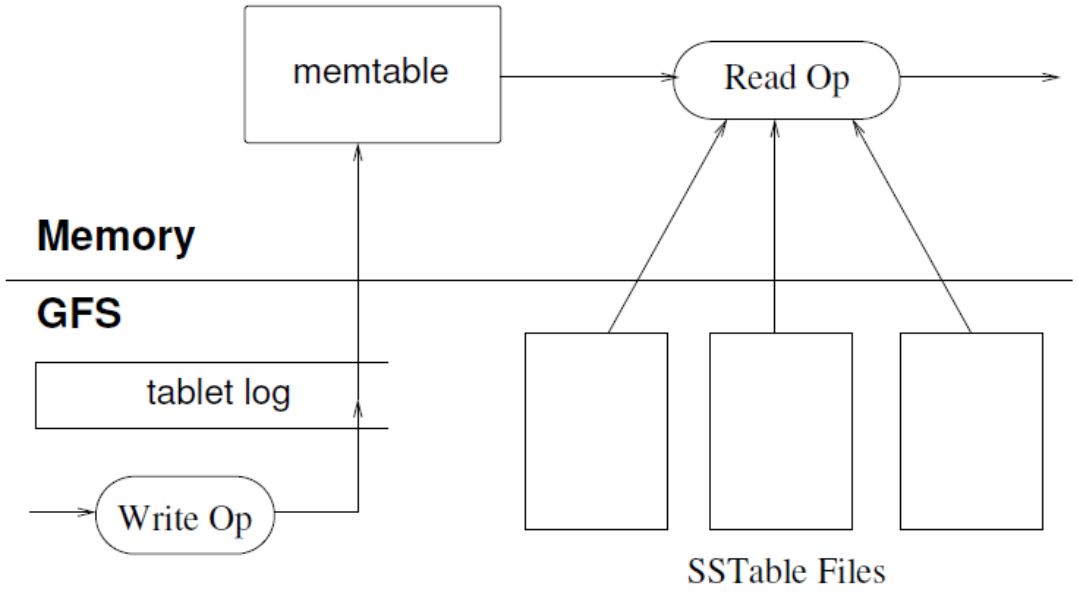
Locating tablets

- 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 (1)

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



Inside a tablet server (2)

- Why immutable SSTable? Simplifies design considerably
  - Can be accessed concurrently without locks during reads and writes
  - Only memtable 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

Summary

- Key ideas:
  - Modified data format (semi structured data) for ease of optimization
    - Simpler than RDBMS, more complex than key-value stores
  - Decentralized master-worker design for scalability: tablet servers directly serve data to clients, master only maintains metadata. System performance scales by adding more tablet servers
  - Reuse existing functionality: SSTable in GFS for distributed storage, Chubby for coordination/consensus
  - Store data as immutable, fixed size SSTables (not general purpose files) for efficiency

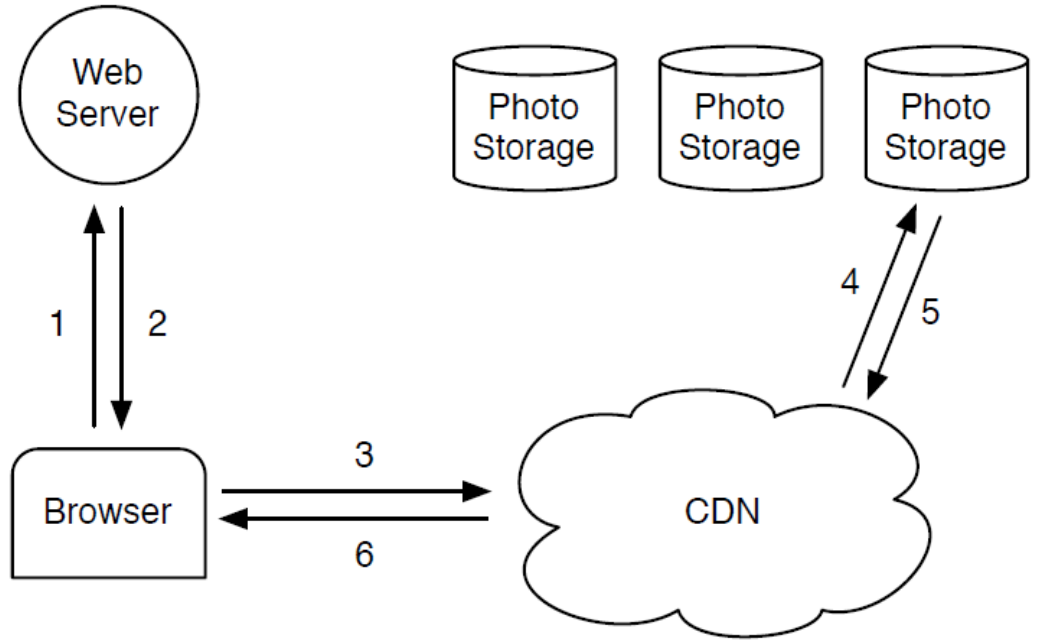
Facebook's photo storage: Haystack

- 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

Finding a needle in Haystack: Facebook's photo storage
Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, Peter Vajgel

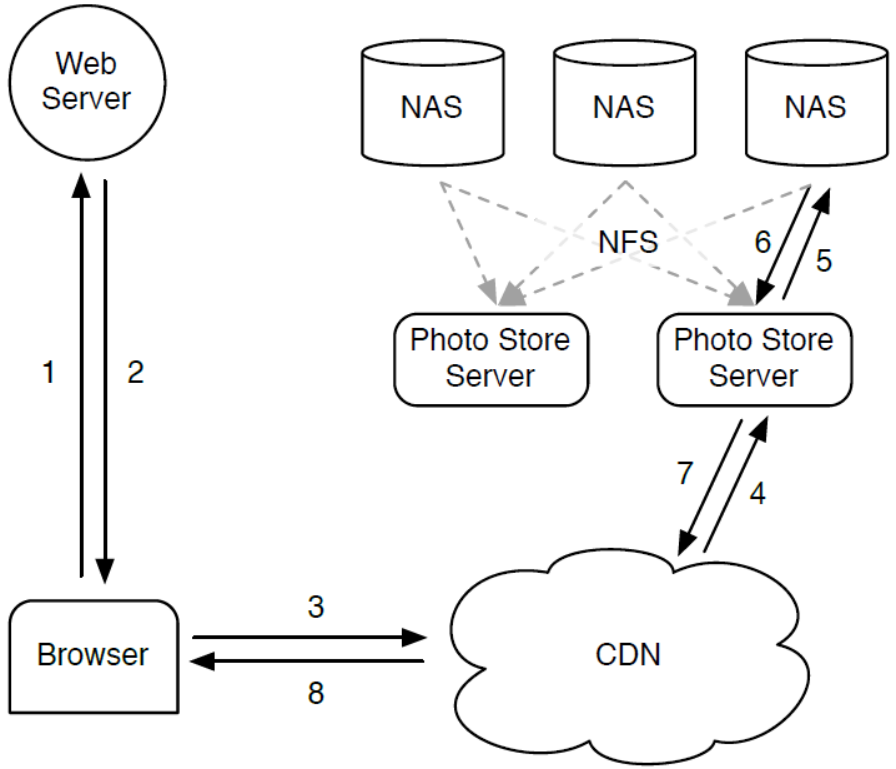
Typical design of photo/object storage

- 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 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 and key idea

- 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

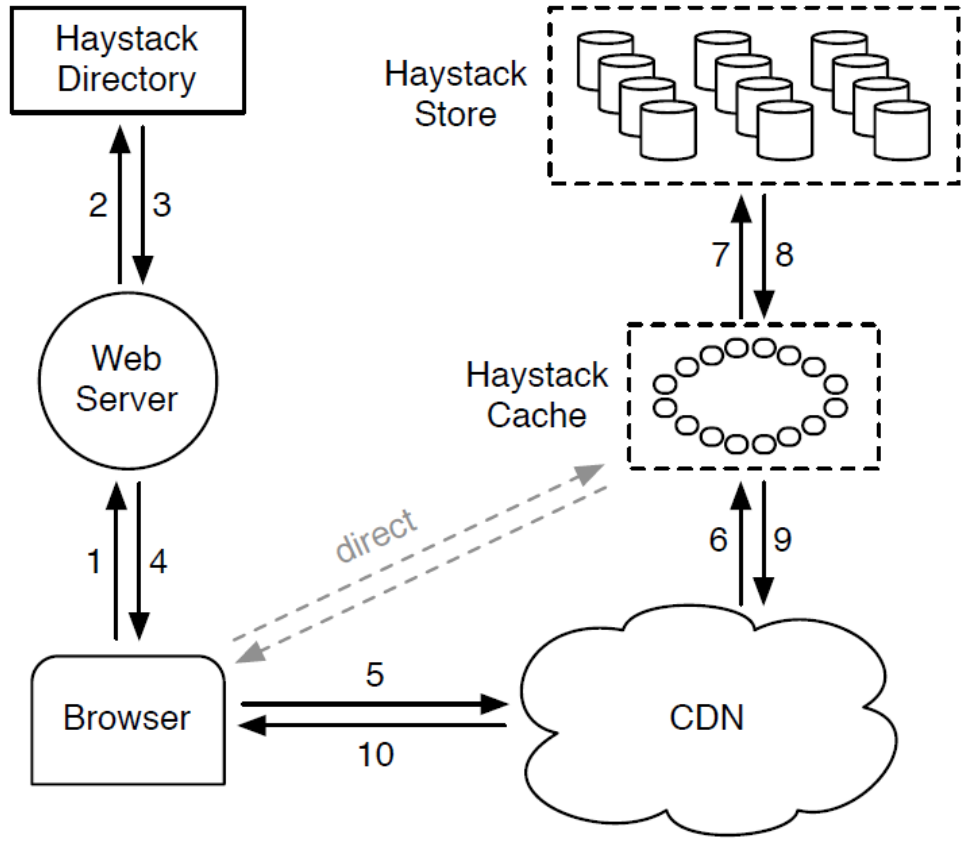


Figure 3: Serving a photo

Photo uploads

- 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

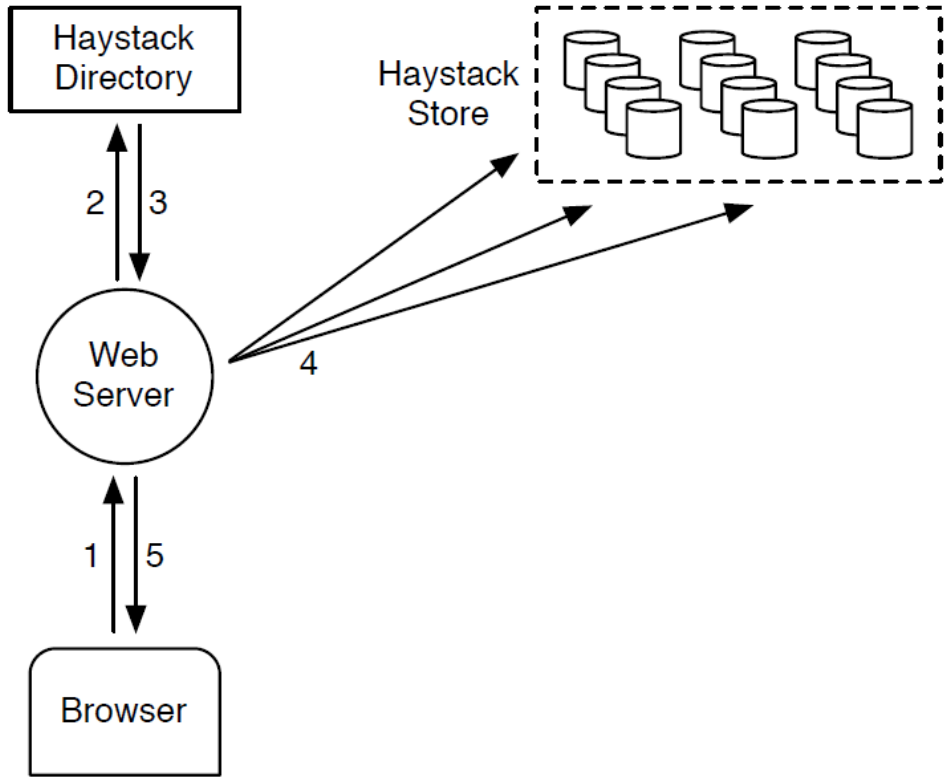


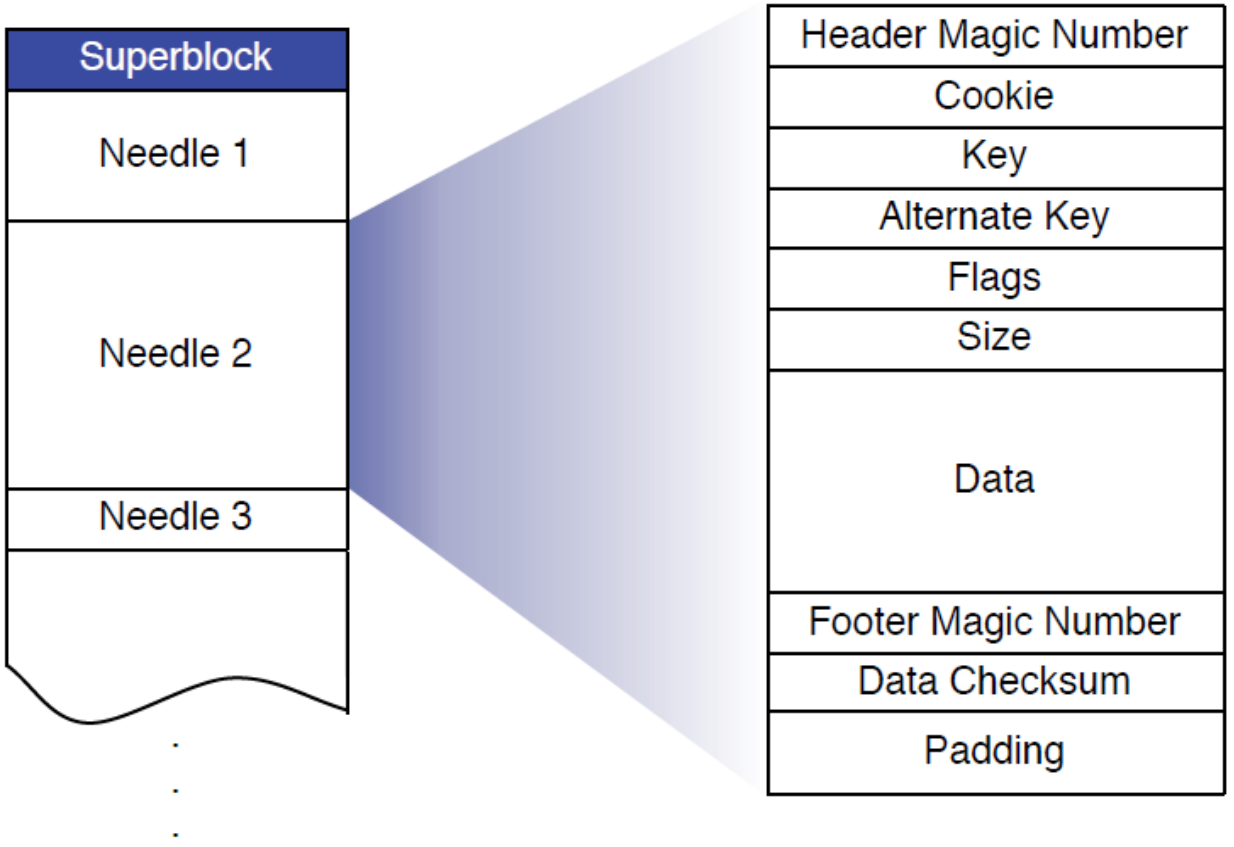
Figure 4: Uploading a photo

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 superblock 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 records after that are read and incorporated into index

Summary

- Application specific knowledge to optimize cloud storage
  - Not optimal to use general purpose filesystem for storing a specific type of files (photos) with specific usage patterns (write once, read multiple times, rarely modified)
  - Efficient disk layout and index design ensures close to one disk access per photo read
  - Good performance: benchmarks in paper show that Haystack achieves 85% of raw disk throughput, and only 17% extra latency (almost close to one disk access per photo)

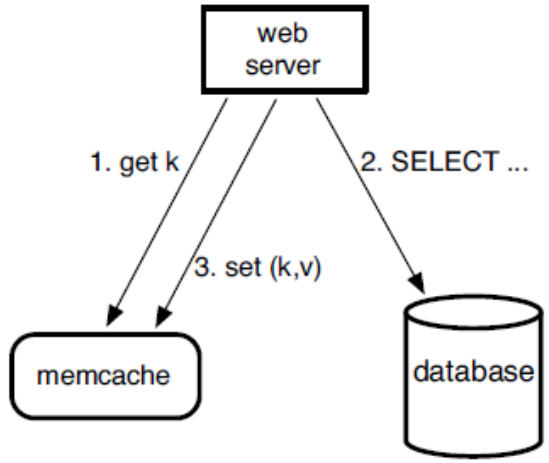
Facebook's Memcache

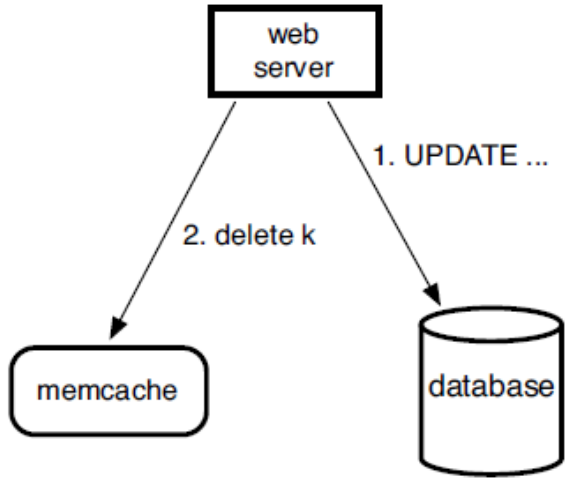
- Facebook starts with a single host version of in-memory key-value store (memcached) and builds a distributed, scalable in-memory caching system (memcache)
  - High performance of billions of requests per second
- Cache sits between web/application servers and backend databases
  - Generic cache that can be used across applications

Scaling Memcache at Facebook
Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, Venkateshwaran Venkataramani

Memcache

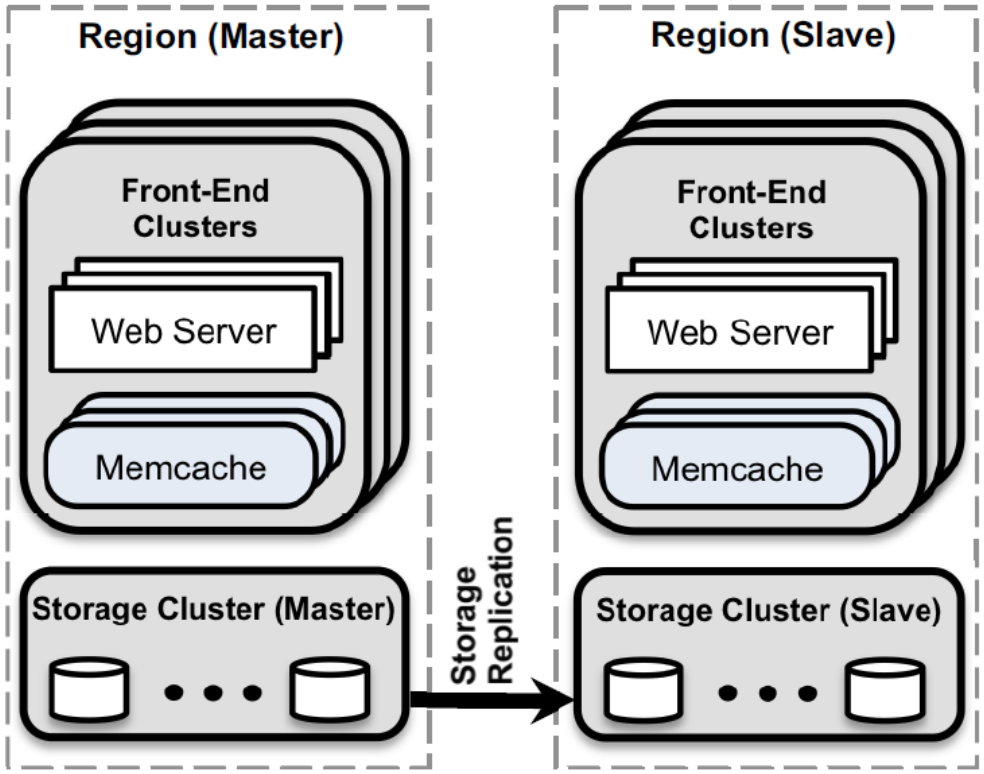
- Demand-filled look-aside cache
  - Cache results of queries to backend databases
- Web server reads data:
  - Look up cache, fetch if available
  - If cache miss, fetch from backend, populate cache
- Web server writes data
  - Write directly to backend
  - Invalidates cache data
- Difference from other No-SQL stores (Dynamo): memcache used for read-heavy workloads, not expected to be persistent/authentic source of data





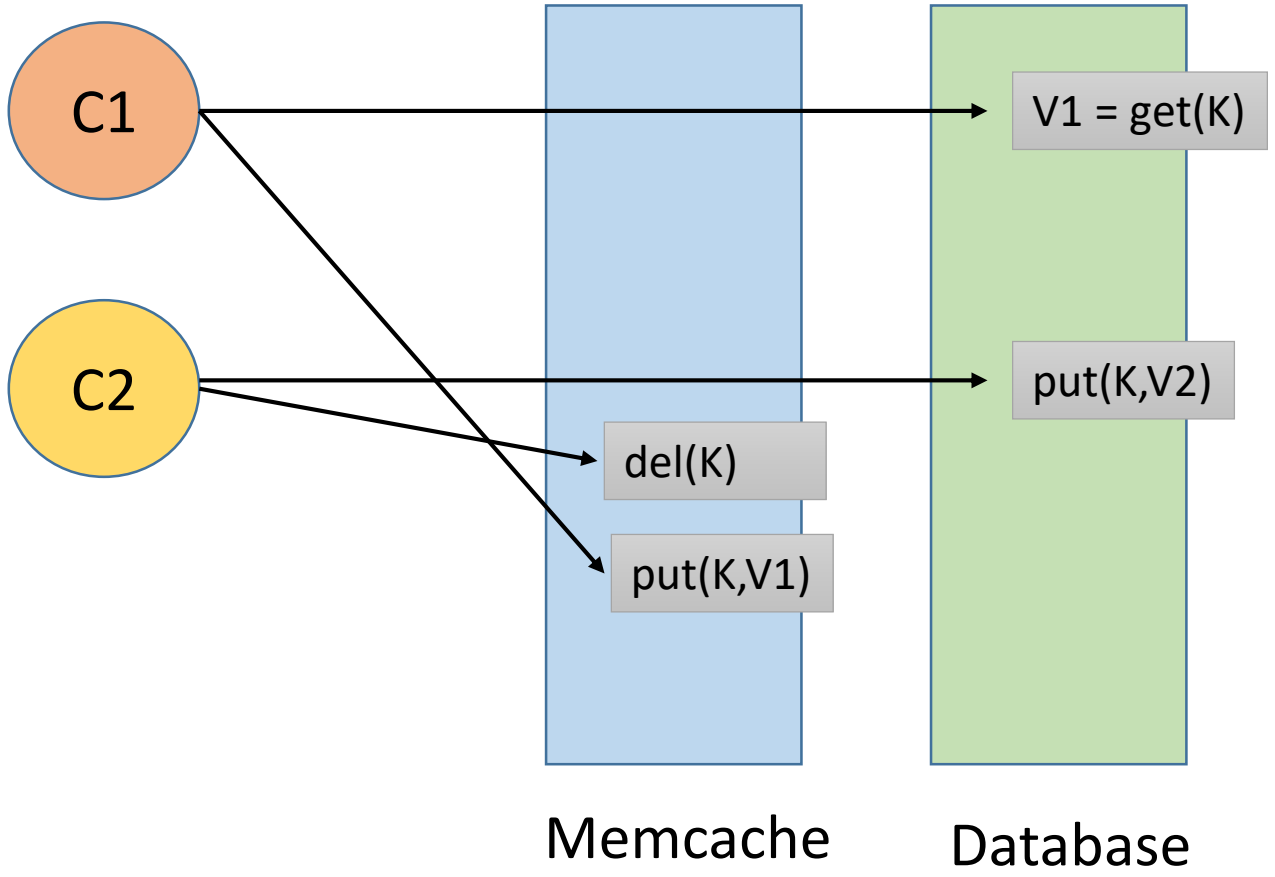
Overall architecture

- Front end clusters:
  - Web server (memcache clients)
  - Memcache (memcached servers)
- Backend storage cluster
  - MySQL databases
- Frontend and backend clusters arranged into regions
  - Region is a failure domain
- Keys divided between memcached servers by consistent hashing
- Web servers (memcache clients) contact the server responsible for a key via a client-side library or a proxy
  - Get requests over UDP (with a sliding window for flow control), put requests over TCP



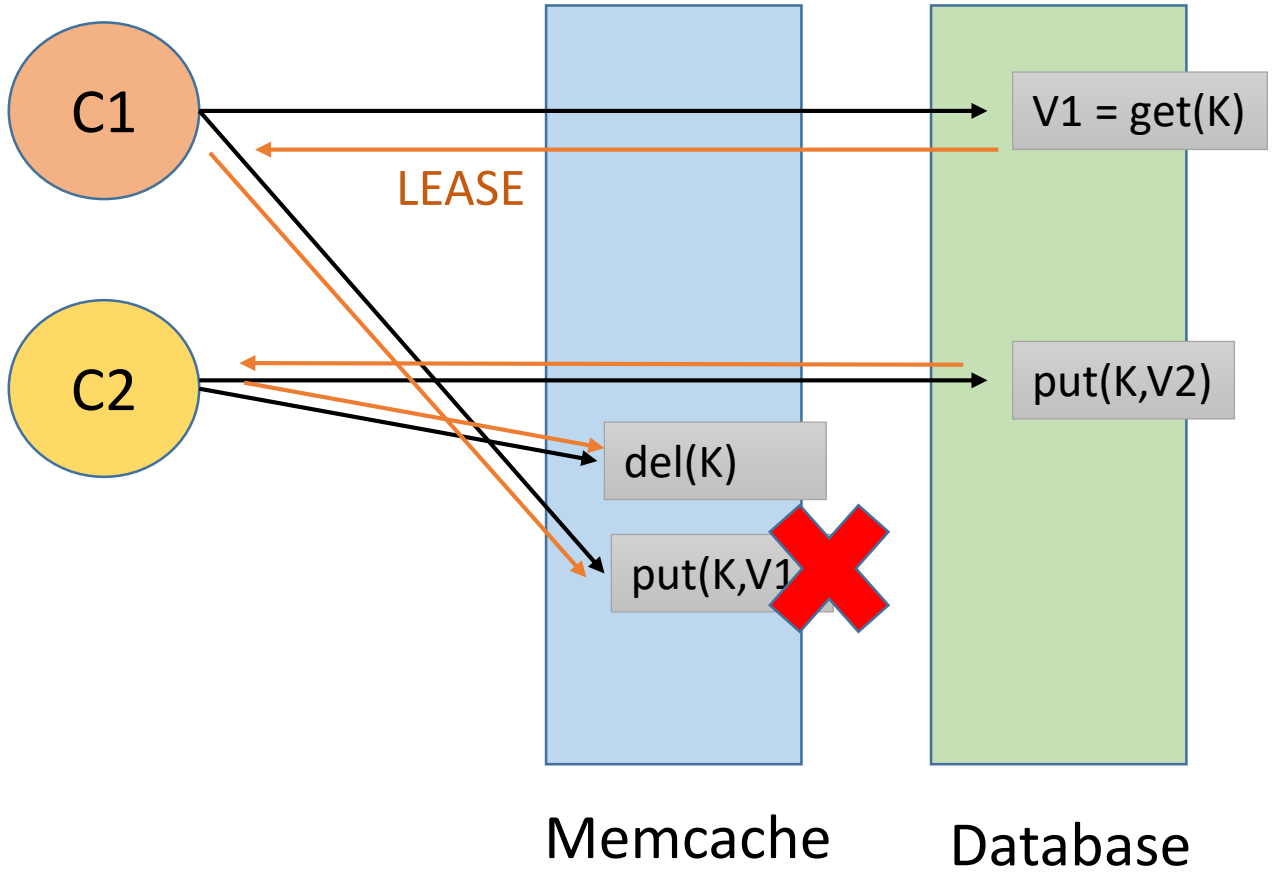
Handling stale values in cache

- C1 gets key K, misses in cache, gets V1 from DB
- C1 puts this value V1 into cache but this put is delayed
- Meanwhile, C2 puts new value V2 into DB and invalidates key K in cache
- Put(K,V1) arrives after del(K)
- Cache contains (K,V1) while DB contains (K,V2)
  - Inconsistent values



Handling stale values in cache: Solution

- When C1 puts value V1 in database, it gets a lease (64-bit version number)
  - This lease to be shown when performing put into cache
- C2 also gets a lease with a higher sequence number
- C1's put wont be accepted at the cache since it has an expired (older) lease
- More mechanisms in the paper on handling consistency across regions (not covered in lecture)



Avoiding the thundering herd

- C0 writes to a key and invalidates cached copy
- C1, C2, .. Cn all perform reads to the key in a short while after invalidation, will miss in cache
- It is enough for one of them to fetch from backend
  - C1 fetches from backend and populates cache
  - C2, ..Cn read from cache – this is ideal scenario
- However, C2, ..Cn may all contact the backend before C1 puts in cache
  - Thundering herd problem
  - Large number of accesses to backend right after cache invalidation
- Fix: C1 is given a lease by backend, but no further leases issues to C2, ..Cn
  - Issue a lease no more than once every 10 seconds
  - C2,..Cn told to wait and check cache later
- Alternately, return stale values for a short time after invalidation (if app permits)

Server “pools”

- Memcache stores different types of keys in different "pools" of servers
  - E.g., some application keys have low churn and long life, others have high churn and short life. If such keys stored together, the high churn keys can replace the low churn keys, which is undesirable
  - Separate key pools for different types of keys avoids impact of one type of application/workload on another
- What happens when servers fail?
  - Permanent failure: remap keys of failed server to another server. Problem: overload at new server (especially if a hot key is remapped)
  - To avoid remapping keys (for transient failures in particular), use a temporary "gutter pool" of servers
  - If a client finds that its assigned server has failed, it gets the key-value pair from the database and puts it into the gutter pool. Other clients also check the gutter pool when they discover the server failure.

Handling server overload

- Suppose caching traffic to server is 1M req/s, server capacity is only 500K req/s. What to do?
- One solution: split keyspace of server. Add another server and give away some keys to new server.
  - Many requests are “multiget”, e.g., client fetches 100 keys together
  - In such cases, both servers will see 1M req/s, but with fewer keys in each get request. Overall load stays same (serving 100 keys or 50 keys incurs similar overhead).
- Another solution: replication. Replicate key-value pairs across two servers
  - Each server gets 500K req/s, each requesting multiple keys
- Replication is better than splitting key space when lot of keys requested together in multiget
  - Splitting key space is not always best solution

Single memcached server optimizations

- Starting point: single memcached server with fixed size hash table
- Automatic expansion of hash table to prevent lookup times from becoming O(n)
- Make server multithreaded with fine-grained locking
- Each thread has a separate UDP port to listen for get requests, to avoid contention
- Slab allocators of various sizes to reduce dynamic memory allocation overheads
  - Adaptive slab sizes to match workload (slabs which are seeing more data will grow bigger)
- Proactively evict short-lived keys instead of waiting for them to be evicted via LRU
  - Short-lived keys stored in a separate transient item cache

Summary

- Techniques to build a caching layer between webservers and backend storage clusters
  - Reuse existing components (memcached server)
  - Add mechanisms to avoid inconsistent results
  - Simple mechanisms instead of stronger guarantees, for better scalability