CLOUD-SCALE INFORMATION RETRIEVAL

Ken Birman, CS5412 Cloud Computing

Styles of cloud computing

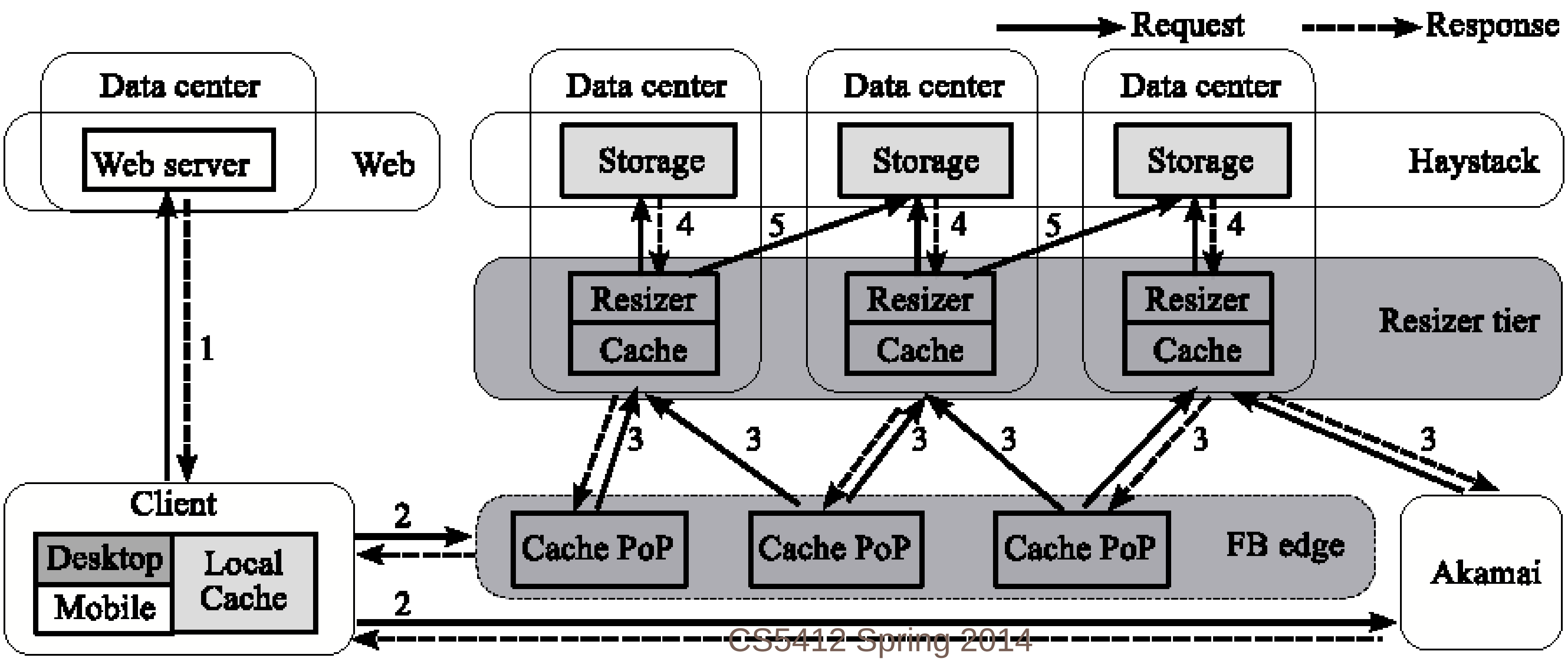
- Think about Facebook…
  - We normally see it in terms of pages that are image-heavy
  - But the tags and comments and likes create "relationships" between objects within the system
  - And FB itself tries to be very smart about what it shows you in terms of notifications, stuff on your wall, timeline, etc…
- How do they actually get data to users with such impressive real-time properties? (often << 100ms!)

Facebook image "stack"

- Role is to serve images (photos, videos) for FB's hundreds of millions of active users
  - About 80B large binary objects ("blob") / day
  - FB has a huge number of big and small data centers
    - "Point of presense" or PoP: some FB owned equipment normally near the user
    - Akamai: A company FB contracts with that caches images
    - FB resizer service: caches but also resizes images
    - Haystack: inside data centers, has the actual pictures (a massive file system)

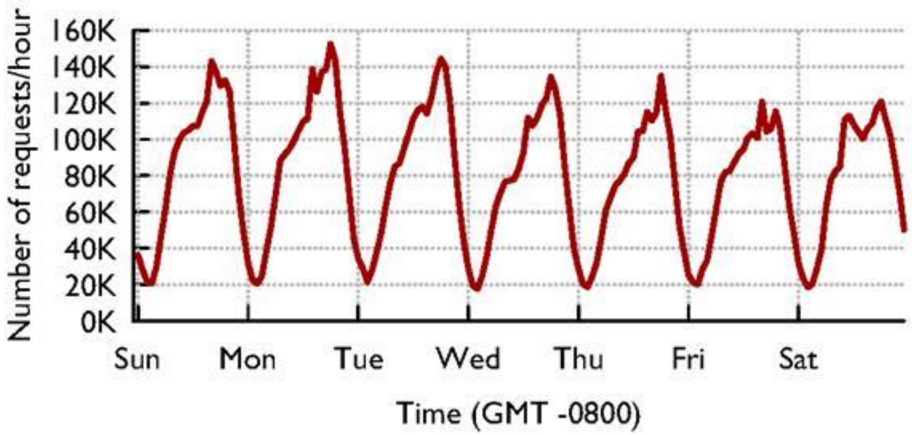
Facebook "architecture"

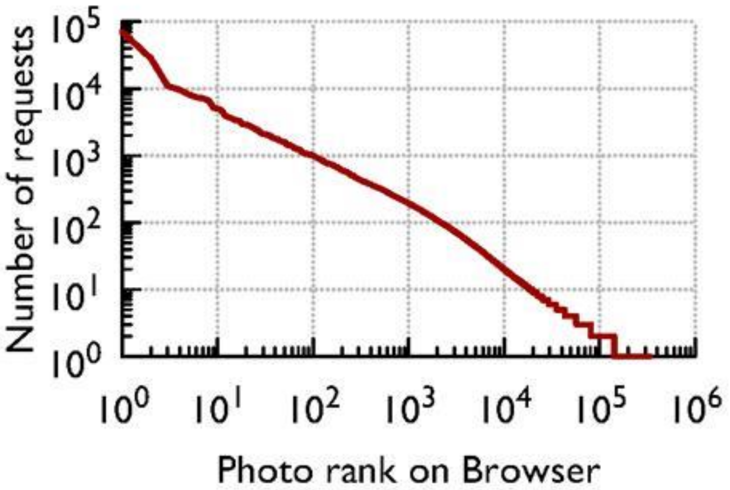
- Think of Facebook as a giant distributed HashMap
  - Key: photo URL (id, size, hints about where to find it...)
  - Value: the blob itself



Facebook traffic for a week

- Client activity varies daily....





- ... and different photos have very different popularity statistics

Observations

- There are huge daily, weekly, seasonal and regional variations in load, but on the other hand the peak loads turn out to be "similar" over reasonably long periods like a year or two
  - Whew! FB only needs to reinvent itself every few years
  - Can plan for the worst-case peak loads…
- And during any short period, some images are way more popular than others: Caching should help

Facebook's goals?

- Get those photos to you rapidly
- Do it cheaply
- Build an easily scalable infrastructure
  - With more users, just build more data centers
- ... they do this using ideas we've seen in cs2110!

Best ways to cache this data?

- Core idea: Build a *distributed photo cache* (like a HashMap, indexed by photo URL)
- Core issue: We could cache data at various places
  - On the client computer itself, near the browser
  - In the PoP
  - In the Resizer layer
  - In front of Haystack
- Where's the best place to cache images?
  - Answer depends on image popularity...

Distributed Hash Tables

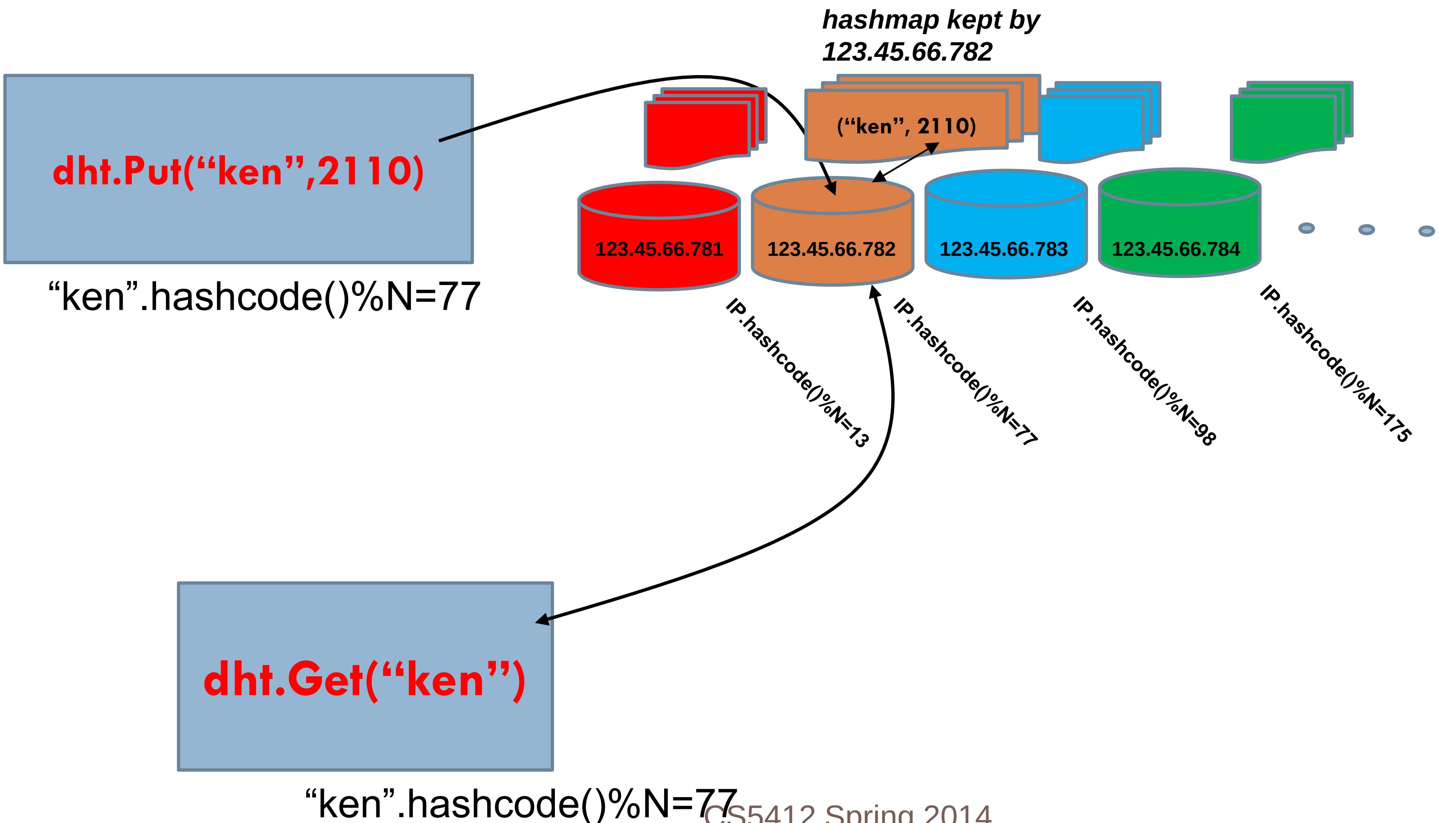
- It is easy for a program on biscuit.cs.cornell.edu to send a message to a program on "jam.cs.cornell.edu"
  - Each program sets up a "network socket
  - Each machine has an IP address, you can look them up and programs can do that too via a simple Java utility
  - Pick a "port number" (this part is a bit of a hack)
  - Build the message (must be in binary format)
  - Java utils has a request

Distributed Hash Tables

- It is easy for a program on biscuit.cs.cornell.edu to send a message to a program on "jam.cs.cornell.edu"
- ... so, given a key and a value
  1. Hash the key
  2. Find the server that "owns" the hashed value
  3. Store the key,value pair in a "local" HashMap there

- To get a value, ask the right server to look up key

Distributed Hash Tables

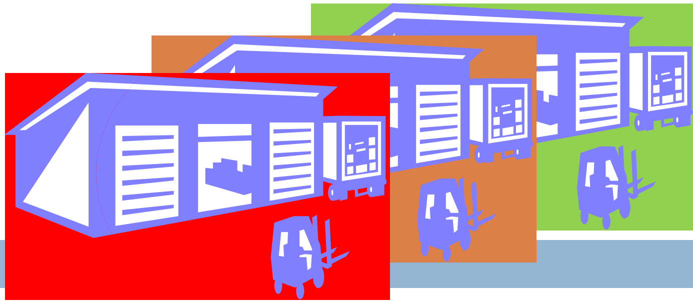


How should we build this DHT?

- DHTs and related solutions seen so far in CS5412
  - Chord, Pastry, CAN, Kelips
  - MemCached, BitTorrent

- They differ in terms of the underlying assumptions
  - Can we safely assume we know which machines will run the DHT?
    - For a P2P situation, applications come and go at will
    - For FB, DHT would run "inside" FB owned data centers, so they can just keep a table listing the active machines…

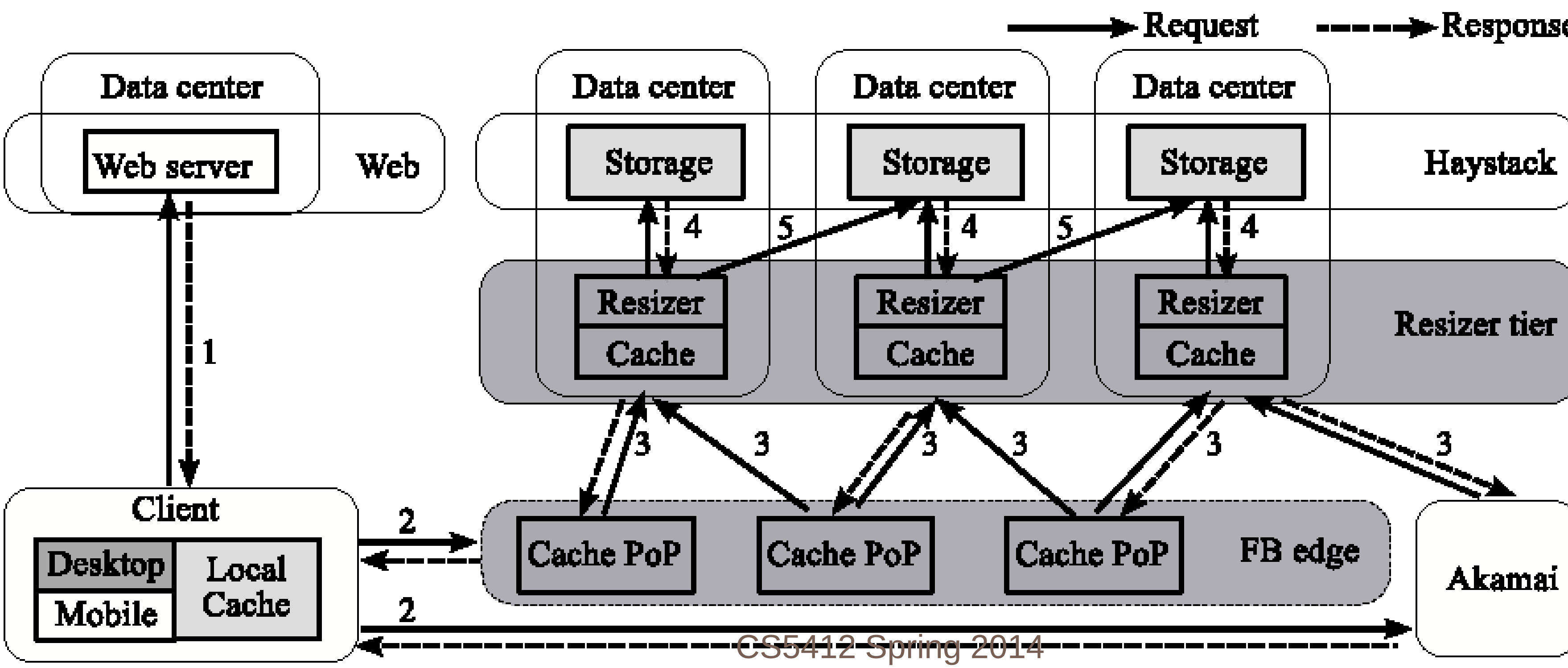
FB DHT approach



- DHT is actually split into many DHT subsystems
  - Each subsystem lives in some FB data center, and there are plenty of those (think of perhaps 50 in the USA)
  - In fact these are really side by side clusters: when FB builds a data center they usually have several nearby buildings each with a data center in it, combined into a kind of regional data center
  - They do this to give "containment" (floods, fires) and also so that they can do service and upgrades without shutting things down (e.g. they shut down 1 of 5…)

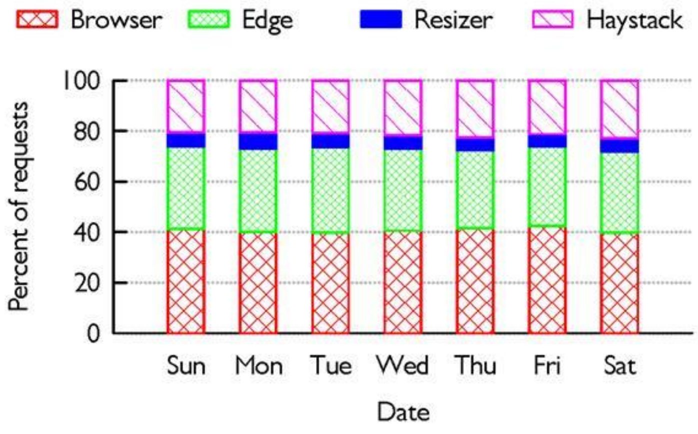
Facebook "architecture"

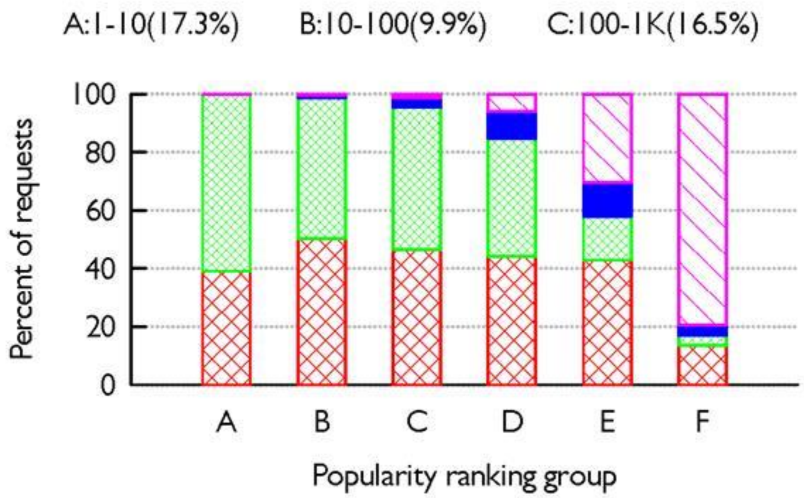
- Think of Facebook as a giant distributed HashMap
  - Key: photo URL (id, size, hints about where to find it...)
  - Value: the blob itself



Facebook cache effectiveness

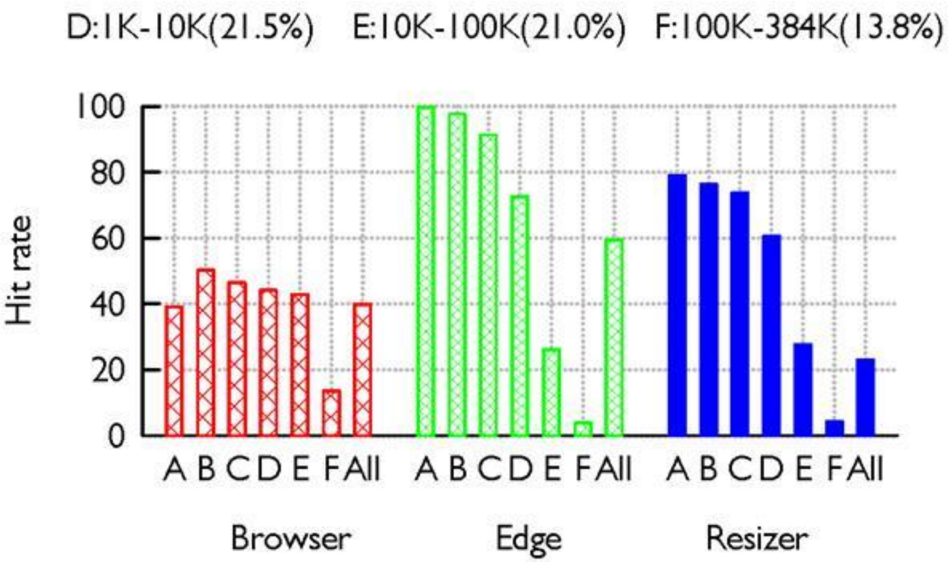
- Existing caches are very effective…
- … but different layers are more effective for images with different popularity ranks



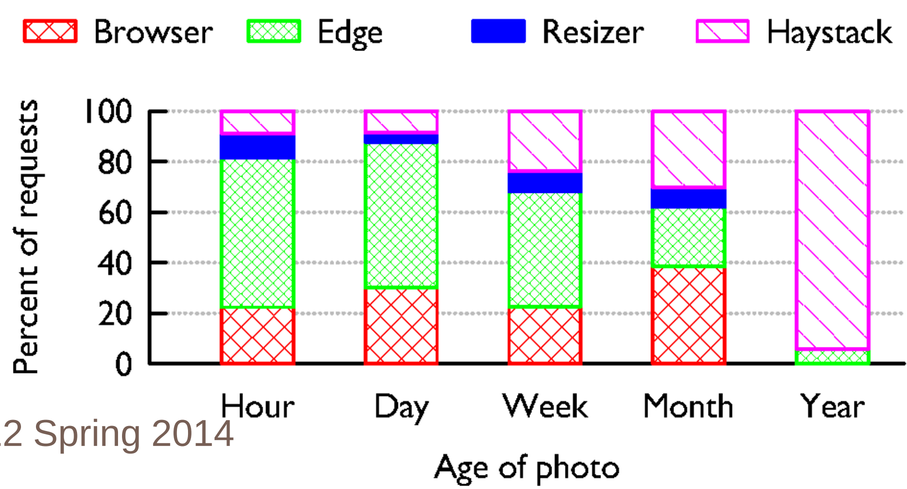


Facebook cache effectiveness

- Each layer should “specialize” in different content.

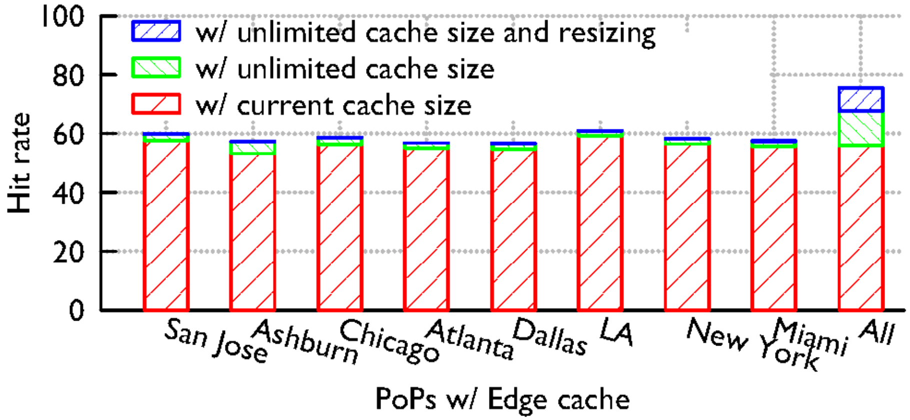


- Photo age strongly predicts effectiveness of caching

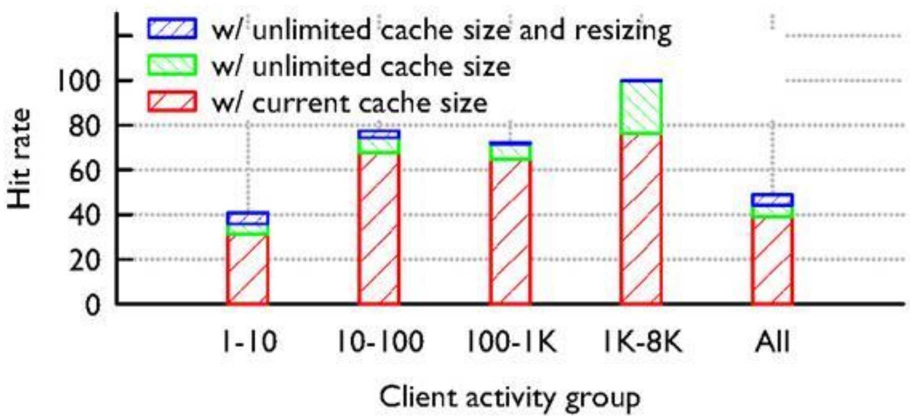


Hypothetical changes to caching?

- We looked at the idea of having Facebook caches collaborate at national scale…

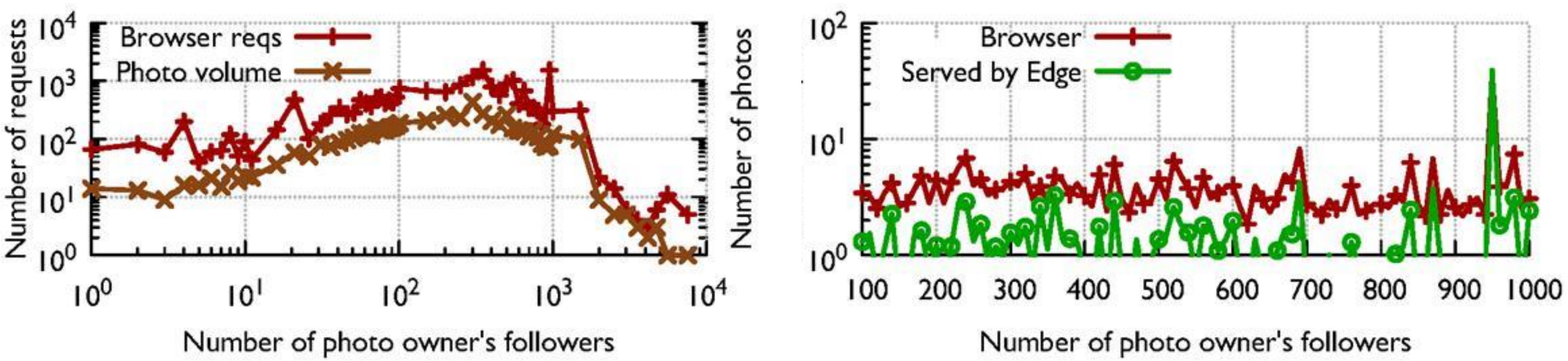


- … and also at how to vary caching based on the "busyness" of the client



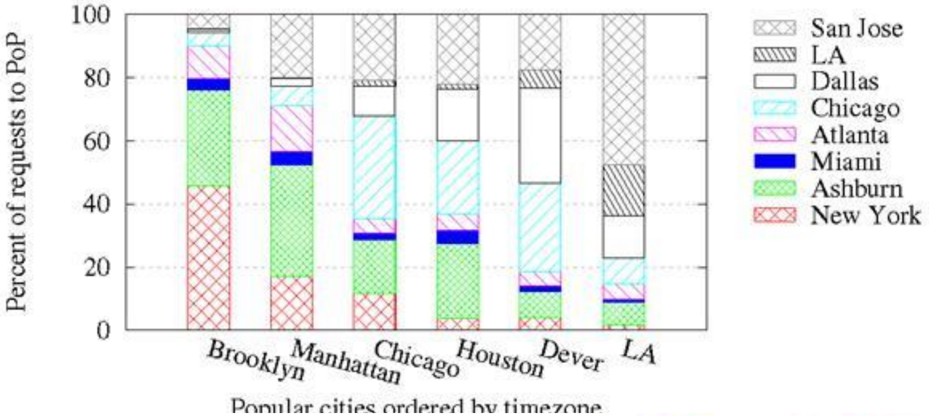
Social networking effect?

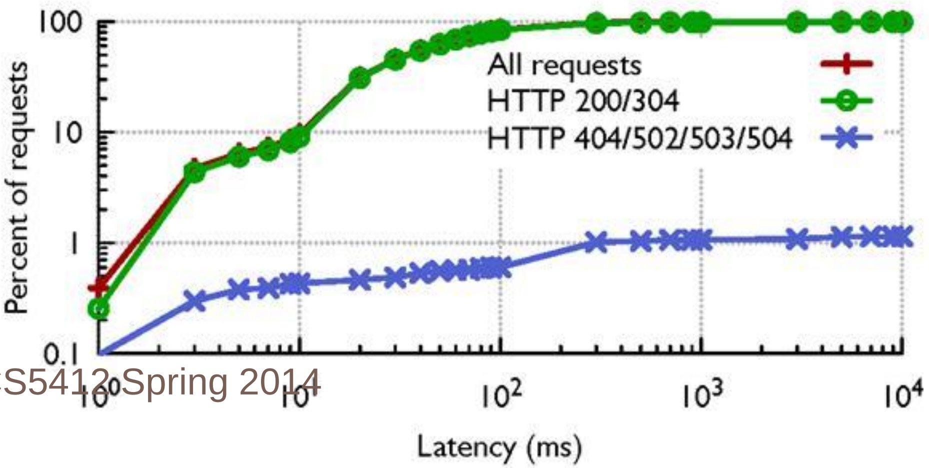
- Hypothesis: caching will work best for photos posted by famous people with zillions of followers
- Actual finding: *not really*



Locality?

- Hypothesis: FB probably serves photos from close to where you are sitting
- Finding: *Not really…*
- … *just the same, if the photo exists, it finds it quickly*





Can one conclude anything?

- Learning what patterns of access arise, and how effective it is to cache given kinds of data at various layers, we can customize cache strategies

- Each layer can look at an image and ask "should I keep a cached copy of this, or not?"

- Smart decisions $\Rightarrow$ Facebook is more effective!

Strategy varies by layer

- Browser should cache less popular content but not bother to cache the very popular stuff
- Akamai/PoP layer should cache the most popular images, etc...

- We also discovered that some layers should "cooperatively" cache even over huge distances
  - Our study discovered that if this were done in the resizer layer, cache hit rates could rise 35%!

Overall picture in cloud computing

- Facebook example illustrates a style of working
  - Identify high-value problems that matter to the community because of the popularity of the service, the cost of operating it, the speed achieved, etc
  - Ask how best to solve those problems, ideally using experiments to gain insight
  - Then build better solutions

- Let's look at another example of this pattern

Caching for TAO

- Facebook recently introduced a new kind of database that they use to track groups
  - Your friends
  - The photos in which a user is tagged
  - People who like Sarah Palin
  - People who like Selina Gomez
  - People who like Justin Beiber
  - People who think Selina and Justin were a great couple
  - People who think Sarah Palin and Justin should be a couple

How is TAO used?

- All sorts of FB operations require the system to
  - Pull up some form of data
  - Then search TAO for a group of things somehow related to that data
  - Then pull up fingernails from that group of things, etc

- So TAO works hard, and needs to deal with all sorts of heavy loads
  - Can one cache TAO data? Actually an open question

How FB does it now

- They create a bank of maybe 1000 TAO servers in each data center
- Incoming queries always of the form “get group associated with this *key*”
- They use consistent hashing to hash key to some server, and then the server looks it up and returns the data. For big groups they use *indirection* and return a pointer to the data plus a few items

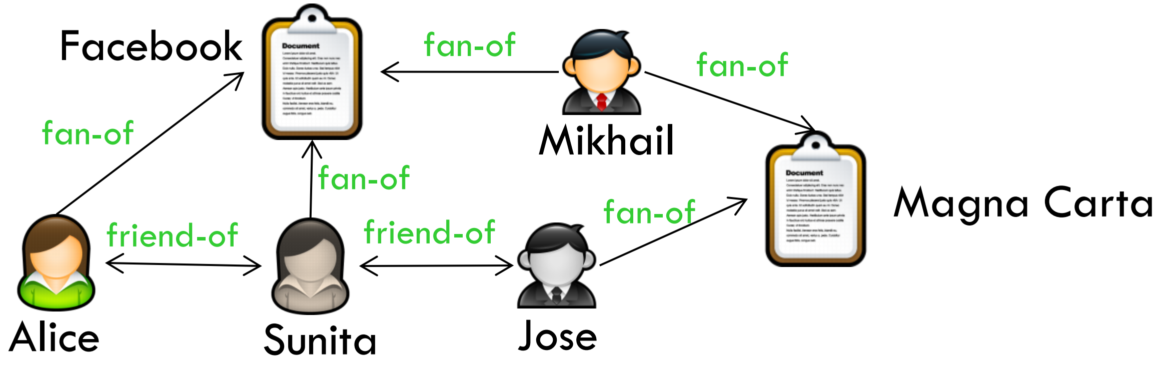
Challenges

- TAO has very high update rates
  - Millions of events per second
  - They use it internally too, to track items you looked at, that you clicked on, sequences of clicks, whether you returned to the prior page or continued deeper…
  - So TAO sees updates at a rate even higher than the total click rate for all of FBs users (billions, but only hundreds of millions are online at a time, and only some of them do rapid clicks… and of course people playing games and so forth don't get tracked this way)

Goals for TAO [Slides from a FB talk given at Upenn in 2012]

- Provide a data store with a graph abstraction (vertexes and edges), not keys+values
- Optimize heavily for reads
  - More than 2 orders of magnitude more reads than writes!
- Explicitly favor efficiency and availability over consistency
  - Slightly stale data is often okay (for Facebook)
  - Communication between data centers in different regions is expensive

Thinking about related objects



- We can represent related objects as a labeled, directed graph
- Entities are typically represented as nodes; relationships are typically edges
  - Nodes all have IDs, and possibly other properties
  - Edges typically have values, possibly IDs and other properties

TAO's data model

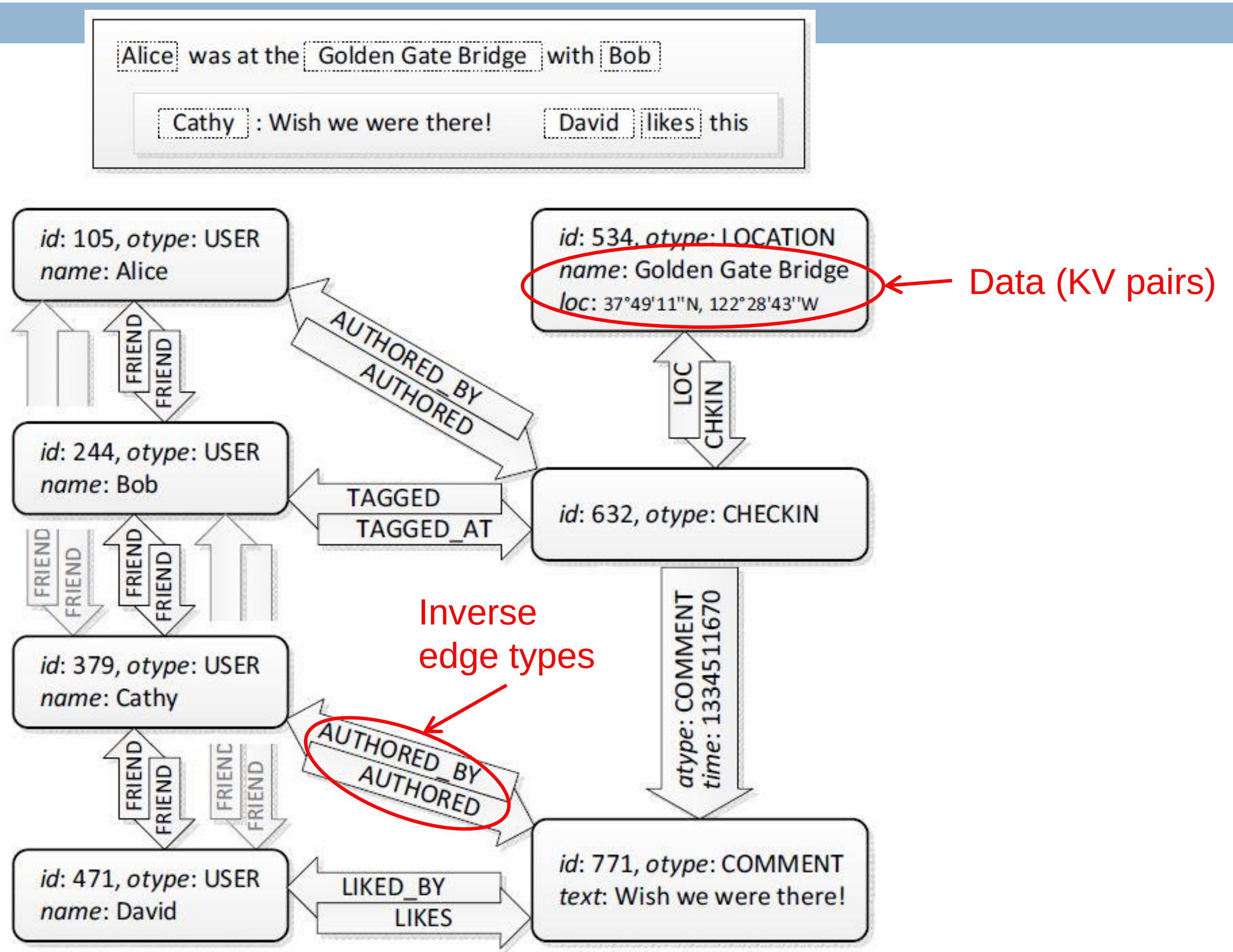
- Facebook's data model is exactly like that!
  - Focuses on people, actions, and relationships
  - These are represented as vertexes and edges in a graph
- Example: Alice visits a landmark with Bob
  - Alice 'checks in' with her mobile phone
  - Alice 'tags' Bob to indicate that he is with her
  - Cathy added a comment
  - David 'liked' the comment

vertexes and edges in the graph

TAO's data model and API

- TAO "objects" (vertexes)
  - 64-bit integer ID (id)
  - Object type (otype)
  - Data, in the form of key-value pairs
- Object API: allocate, retrieve, update, delete
- TAO "associations" (edges)
  - Source object ID (id1)
  - Association type (atype)
  - Destination object ID (id2)
  - 32-bit timestamp
  - Data, in the form of key-value pairs
- Association API: add, delete, change type
- Associations are unidirectional
  - But edges often come in pairs (each edge type has an 'inverse type' for the reverse edge)

Example: Encoding in TAO



Association queries in TAO

- TAO is not a general graph database
  - Has a few specific (Facebook-relevant) queries 'baked into it'
  - Common query: Given object and association type, return an association list (all the outgoing edges of that type)
    - Example: Find all the comments for a given checkin
  - Optimized based on knowledge of Facebook's workload
    - Example: Most queries focus on the newest items (posts, etc.)
    - There is creation-time locality → can optimize for that!
- Queries on association lists:
  - assoc_get(id1, atype, id2set, t_low, t_high)
  - assoc_count(id1, atype)
  - assoc_range(id1, atype, pos, limit) ← "cursor"
  - assoc_time_range(id1, atype, high, low, limit)

TAO's storage layer

- Objects and associations are stored in mySQL
- But what about scalability?
  - Facebook's graph is far too large for any single mySQL DB!!
- Solution: Data is divided into logical shards
  - Each object ID contains a shard ID
  - Associations are stored in the shard of their source object
  - Shards are small enough to fit into a single mySQL instance!
  - A common trick for achieving scalability
  - What is the 'price to pay' for sharding?

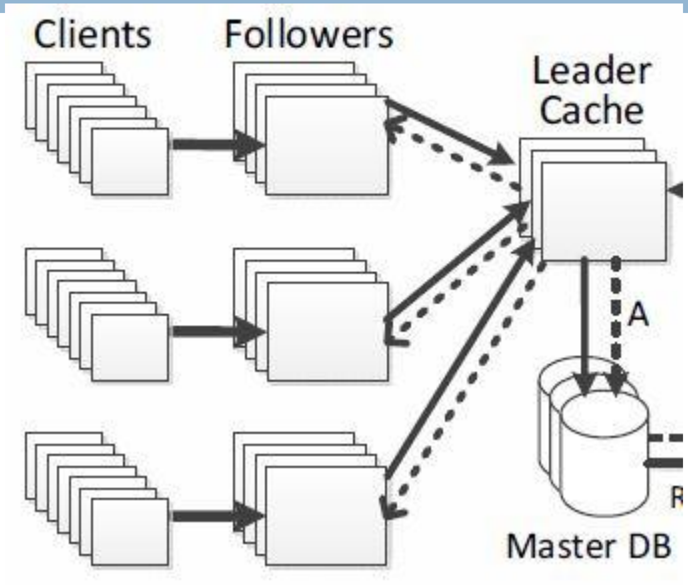
Caching in TAO (1/2)

- Problem: Hitting mySQL is very expensive
  - But most of the requests are read requests anyway!
  - Let's try to serve these from a cache
- TAO's cache is organized into tiers
  - A tier consists of multiple cache servers (number can vary)
  - Sharding is used again here → each server in a tier is responsible for a certain subset of the objects+associations
  - Together, the servers in a tier can serve any request!
  - Clients directly talk to the appropriate cache server
    - Avoids bottlenecks!
  - In-memory cache for objects, associations, and association counts (!)

Caching in TAO (2/2)

- How does the cache work?
  - New entries filled on demand
  - When cache is full, least recently used (LRU) object is evicted
  - Cache is "smart": If it knows that an object had zero associ-ations of some type, it knows how to answer a range query
    - Could this have been done in Memcached? If so, how? If not, why not?
- What about write requests?
  - Need to go to the database (write-through)
  - But what if we're writing a bidirectonal edge?
    - This may be stored in a different shard → need to contact that shard!
  - What if a failure happens while we're writing such an edge?
    - You might think that there are transactions and atomicity...
    - ... but in fact, they simply leave the 'hanging edges' in place (why?)
    - Asynchronous repair job takes care of them eventually

Leaders and followers



- How many machines should be in a tier?
  - Too many is problematic: More prone to hot spots, etc.
- Solution: Add another level of hierarchy
  - Each shard can have multiple cache tiers: one leader, and multiple followers
  - The leader talks directly to the mySQL database
  - Followers talk to the leader
  - Clients can only interact with followers
  - Leader can protect the database from 'thundering herds'

Leaders/followers and consistency

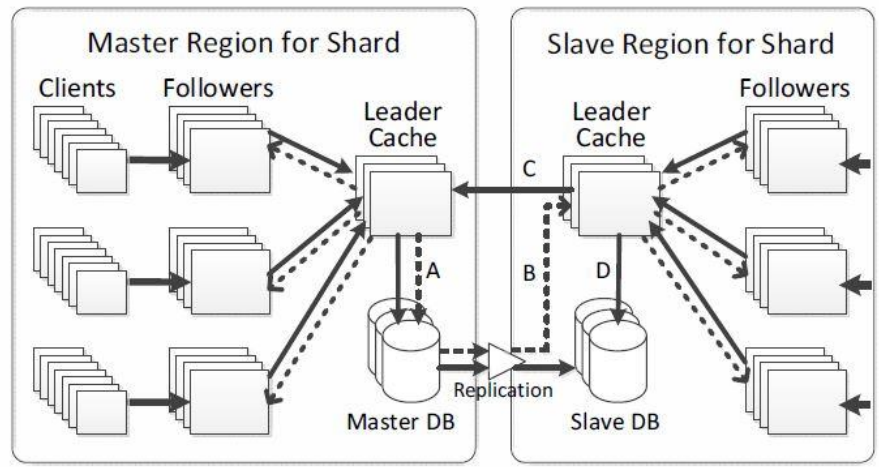
- What happens now when a client writes?
  - Follower sends write to the leader, who forwards to the DB
  - Does this ensure consistency? No!
- Need to tell the other followers about it!
  - Write to an object → Leader tells followers to invalidate any cached copies they might have of that object
  - Write to an association → Don't want to invalidate. Why?
    - Followers might have to throw away long association lists!
  - Solution: Leader sends a 'refill message' to followers
    - If follower had cached that association, it asks the leader for an update
  - What kind of consistency does this provide?

Scaling geographically

- Facebook is a global service. Does this work?
- No - laws of physics are in the way!
  - Long propagation delays, e.g., between Asia and U.S.
  - What tricks do we know that could help with this?

Scaling geographically

- Idea: Divide data centers into regions; have one full replica of the data in each region



  - What could be a problem with this approach?
  - Again, consistency!
  - Solution: One region has the 'master' database; other regions forward their writes to the master
  - Database replication makes sure that the 'slave' databases eventually learn of all writes; plus invalidation messages, just like with the leaders and followers

Handling failures

- What if the master database fails?
  - Can promote another region's database to be the master
  - But what about writes that were in progress during switc
  - Wha
  - TAO

shard, and is automatically switched to recover from the failure of a database. Writes that fail during the switch are reported to the client as failed, and are not retried. Note that since each cache hosts multiple shards, a server

Consistency in more detail

- What is the overall level of consistency?
  - During normal operation: Eventual consistency (why?)
  - Refills and invalidations are delivered 'eventually' (typical delay is less than one second)
  - Within a tier: Read-after-write (why?)
- When faults occur, consistency can degrade
  - In some situations, clients can even observe values 'go back in time'!
  - How bad is this (for Facebook specifically / in general)?
- Is eventual consistency always 'good enough'?
  - No - there are a few operations on Facebook that need stronger consistency (which ones?)
  - TAO reads can be marked 'critical' ; such reads are handled directly by the master.

Fault handling in more detail

- General principle: Best-effort recovery
  - Preserve availability and performance, not consistency!
- Database failures: Choose a new master
  - Might happen during maintenance, after crashes, repl. lag
- Leader failures: Replacement leader
  - Route around the faulty leader if possible (e.g., go to DB)
- Refill/invalidation failures: Queue messages
  - If leader fails permanently, need to invalidate cache for the entire shard
- Follower failures: Failover to other followers
  - The other followers jointly assume responsibility for handling the failed follower's requests

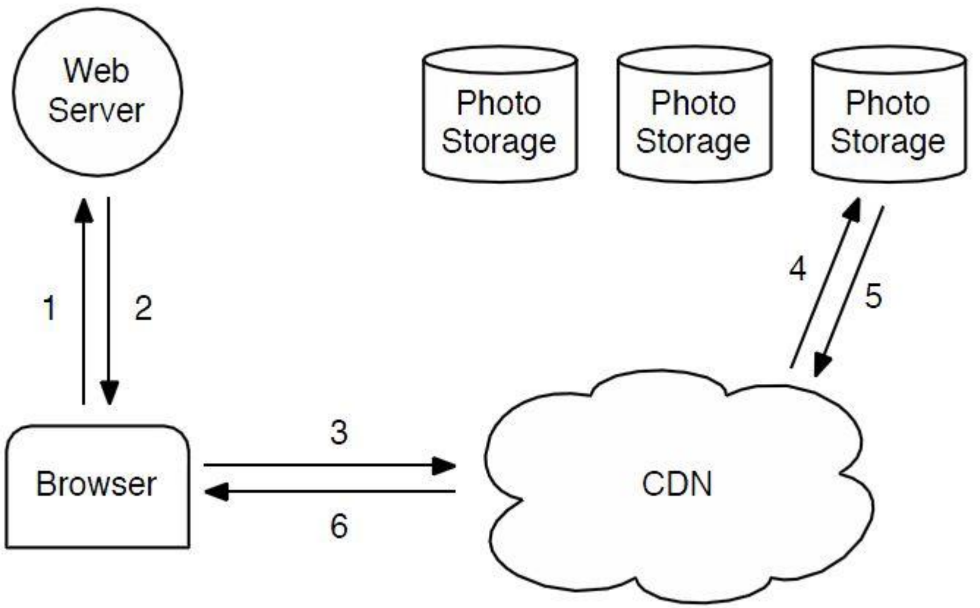
Production deployment at Facebook

- Impressive performance
  - Handles 1 billion reads/sec and 1 million writes/sec!
- Reads dominate massively
  - Only 0.2% of requests involve a write
- Most edge queries have zero results
  - 45% of assoc_count calls return 0...
  - but there is a heavy tail: 1% return >500,000! (why?)
- Cache hit rate is very high
  - Overall, 96.4%!

TAO Summary

- The data model really does matter!
  - KV pairs are nice and generic, but you sometimes can get better performance by telling the storage system more about the kind of data you are storing in it ($\rightarrow$ optimizations!)
- Several useful scaling techniques
  - "Sharding" of databases and cache tiers (not invented at Facebook, but put to great use)
  - Primary-backup replication to scale geographically
- Interesting perspective on consistency
  - On the one hand, quite a bit of complexity & hard work to do well in the common case (truly "best effort")
  - But also, a willingness to accept eventual consistency (or worse!) during failures, or when the cost would be high

HayStack Storage Layer

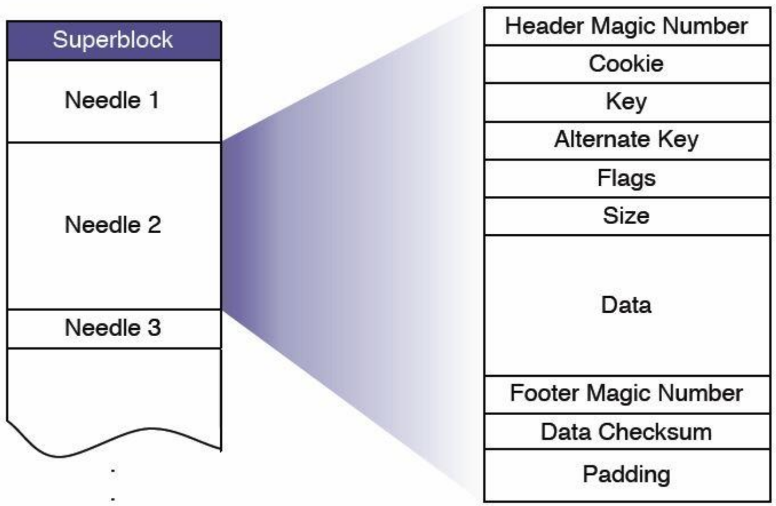


- Facebook stores a huge number of images
  - In 2010, over 260 billion (~20PB of data)
  - One billion (~60TB) new uploads each week
- How to serve requests for these images?
  - Typical approach: Use a CDN (and Facebook does do that)

Haystack challenges

- Very long tail: People often click around and access very rarely seen photos
- Disk I/O is costly
  - Haystack goal: one seek and one read per photo
- Standard file systems are way too costly and inefficient
  - Haystack response: Store images and data in long "strips" (actually called "volumes")
  - Photo isn't a file; it is in a strip at off=xxxx len=yyyy

Haystack: The Store (1/2)



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

- Volumes are simply very large files (~100GB)
  - Few of them needed → In-memory data structures small
- Structure of each file:
  - A header, followed by a number of 'needles' (images)
  - Cookies included to prevent guessing attacks
  - Writes simply append to the file; deletes simply set a flag

Haystack: The Store (2/2)

- Store machines have an in-memory index
  - Maps photo IDs to offsets in the large files
- What to do when the machine is rebooted?
  - Option #1: Rebuild from reading the files front-to-back
    - Is this a good idea?
  - Option #2: Periodically write the index to disk
- What if the index on disk is stale?
  - File remembers where the last needle was appended
  - Server can start reading from there
  - Might still have missed some deletions - but the server can 'lazily' update that when someone requests the deleted img

Recovery from failures

- Lots of failures to worry about
  - Faulty hard disks, defective controllers, bad motherboards...
- Pitchfork service scans for faulty machines
  - Periodically tests connection to each machine
  - Tries to read some data, etc.
  - If any of this fails, logical (!) volumes are marked read-only
    - Admins need to look into, and fix, the underlying cause
- Bulk sync service can restore the full state
  - ... by copying it from another replica
  - Rarely needed

How well does it work?

- How much metadata does it use?
  - Only about 12 bytes per image (in memory)
  - Comparison: XFS inode alone is 536 bytes!
  - More performance data in the paper

- Cache hit rates: Approx. 80%

Summary

- Different perspective from TAO's
  - Presence of "long tail" → caching won't help as much
- Interesting (and unexpected) bottleneck
  - To get really good scalability, you need to understand your system at all levels!
- In theory, constants don't matter - but in practice, they do!
  - Shrinking the metadata made a big difference to them, even though it is 'just' a 'constant factor'
  - Don't (exclusively) think about systems in terms of big-O notations!