Case Study 3: Scalable Caching with Memcache

Ashvin Goel

Electrical and Computer Engineering
University of Toronto

Distributed Systems
ECE419

Slides are modified from the original talks by
Rajesh Nishtala and Nathan Bronson

Case study on scaling storage

- Facebook's experience with using in-memory caches to scale storage
- The practical problems that were encountered
- How they were solved
- Tradeoffs between performance and consistency

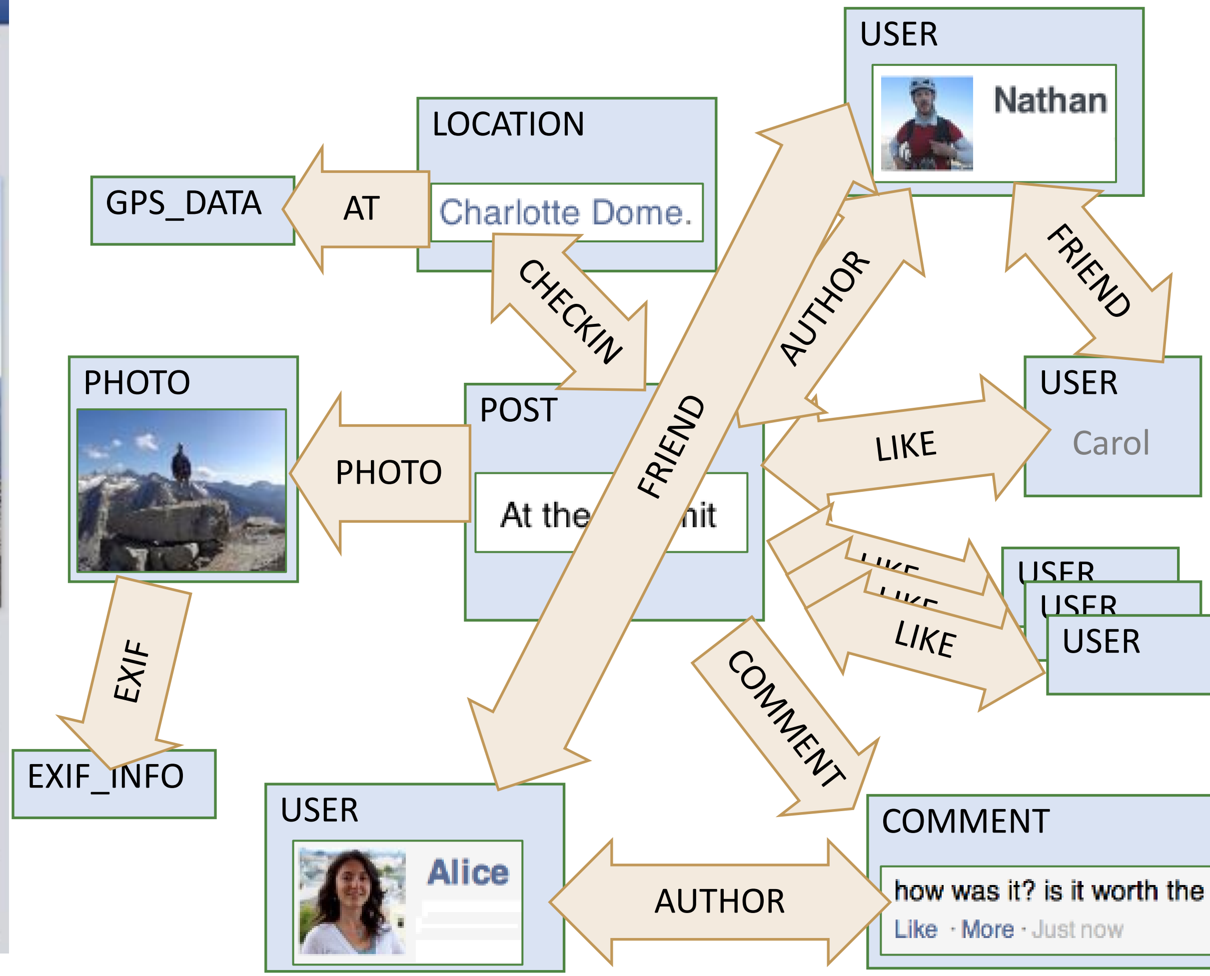
Overview

- Introduction to Facebook storage infrastructure
- One Memcache server
- Memcache servers in a cluster
- Memcache servers in multiple clusters within a region
- Geographically distributed clusters in multiple regions

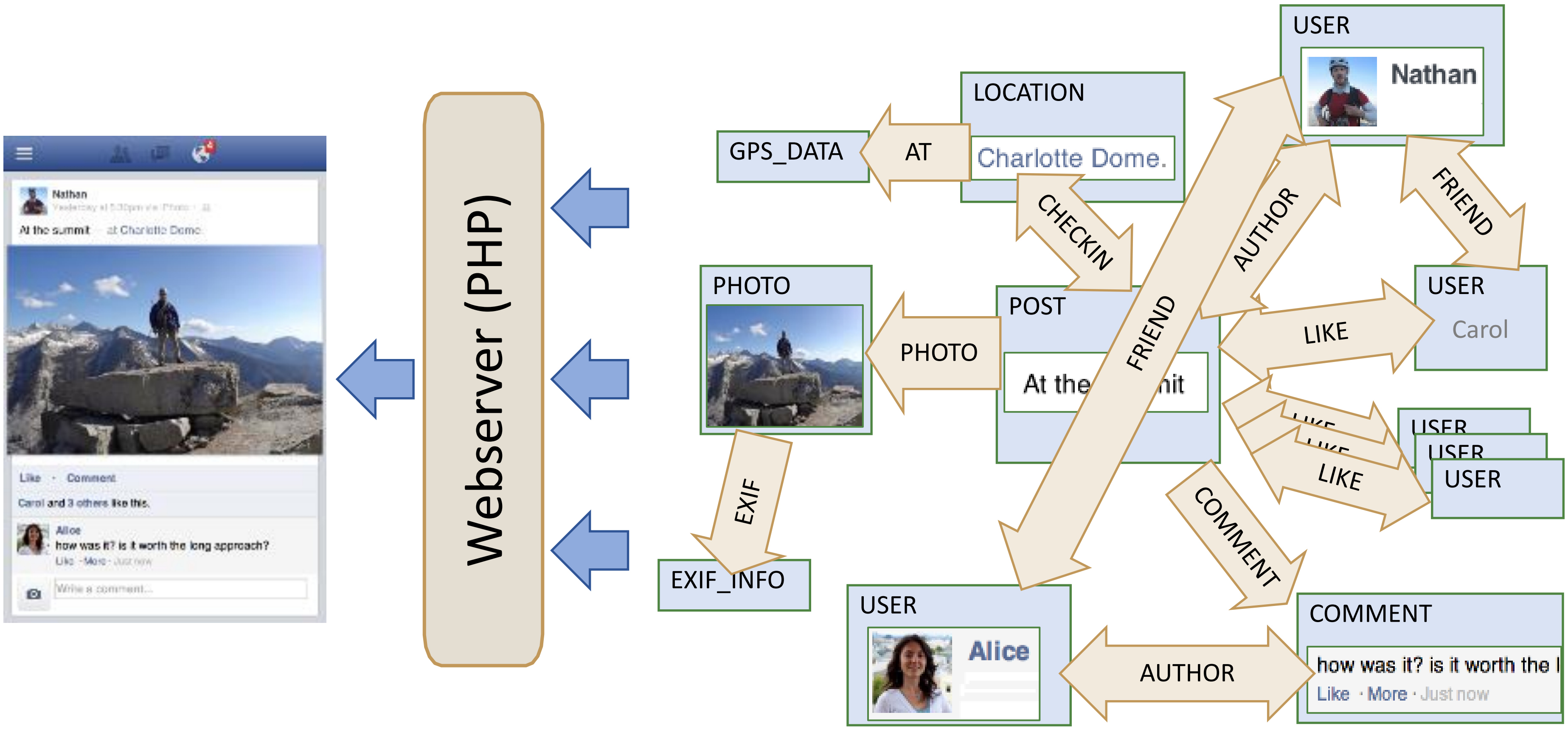
Requirements at Facebook

- Scale to process millions of user requests per second
  - Support heavy read load (over 1 billion reads/sec)
  - Near real-time communication, so tight latency requirements
  - Be able to access and update popular shared content, so hot spots
  - Poor locality for storage accesses
- Scale to petabytes of storage
- Geographically distributed users, multiple data centers

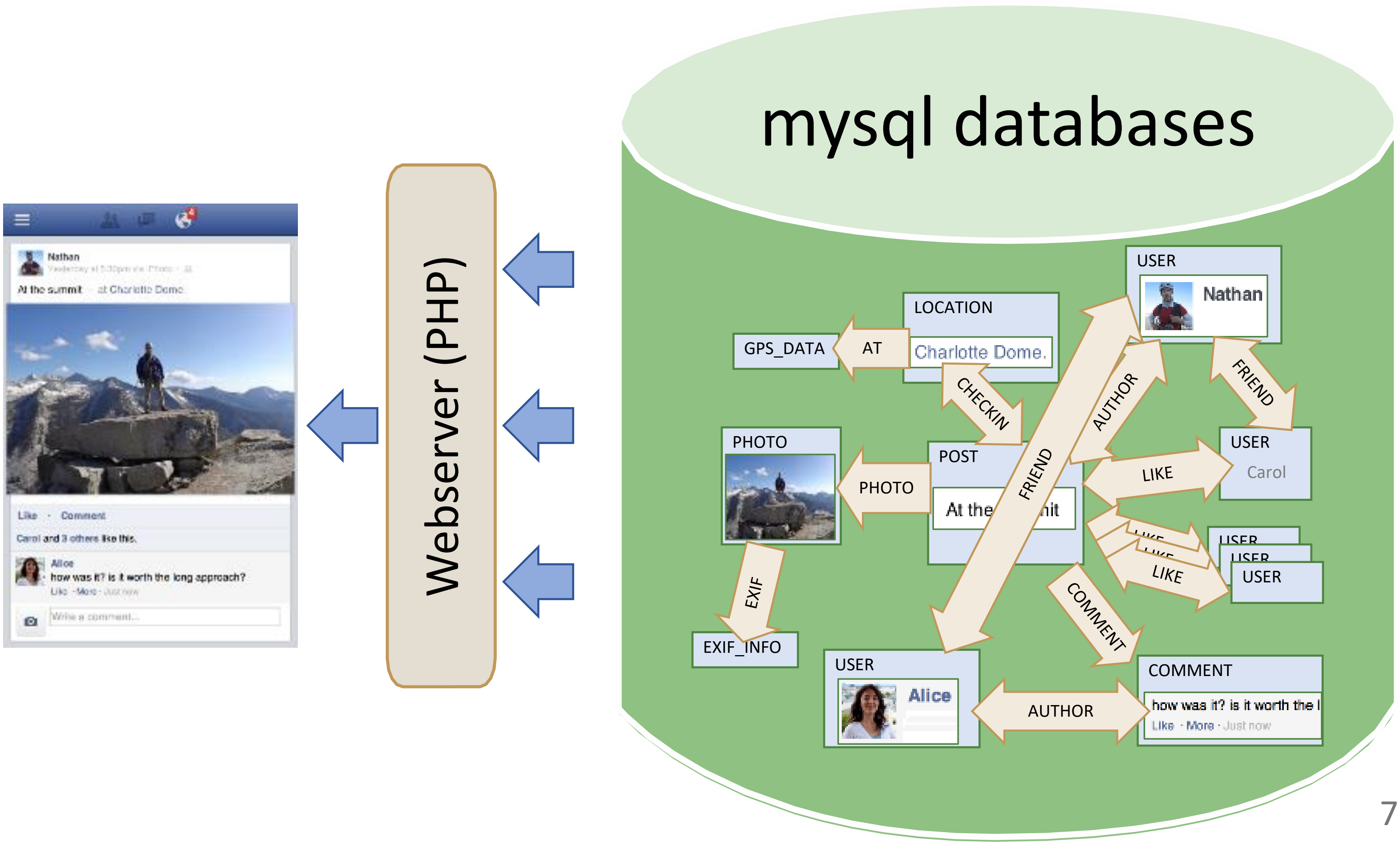
Facebook's social graph



Rendering the social graph

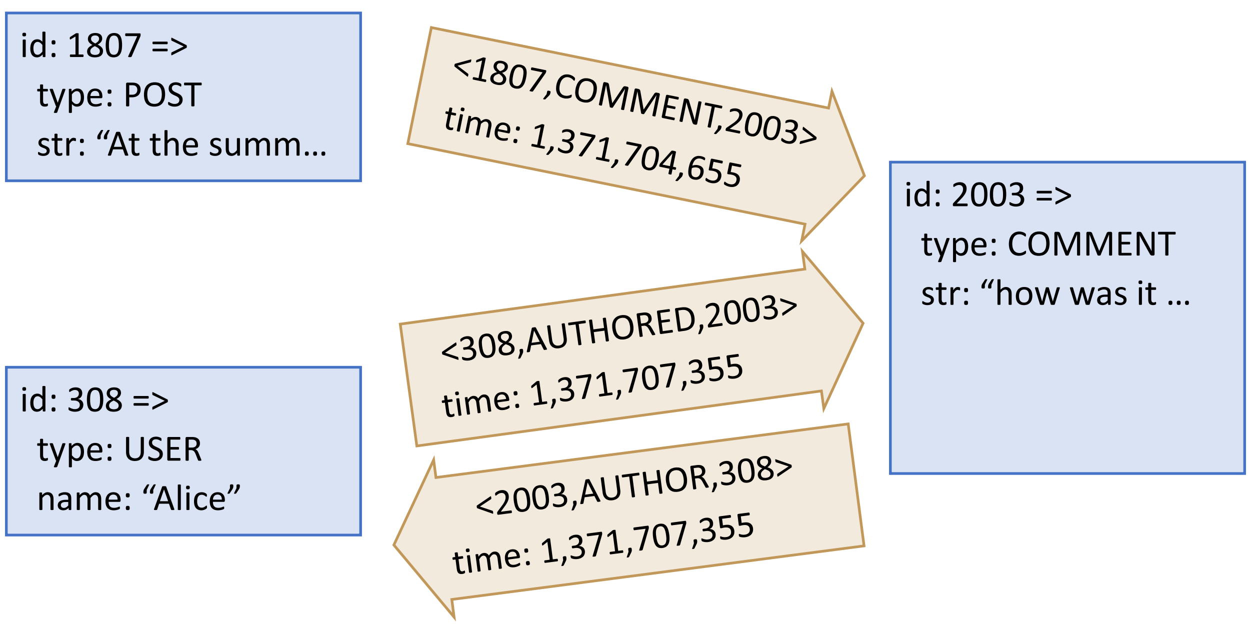


Storing the social graph



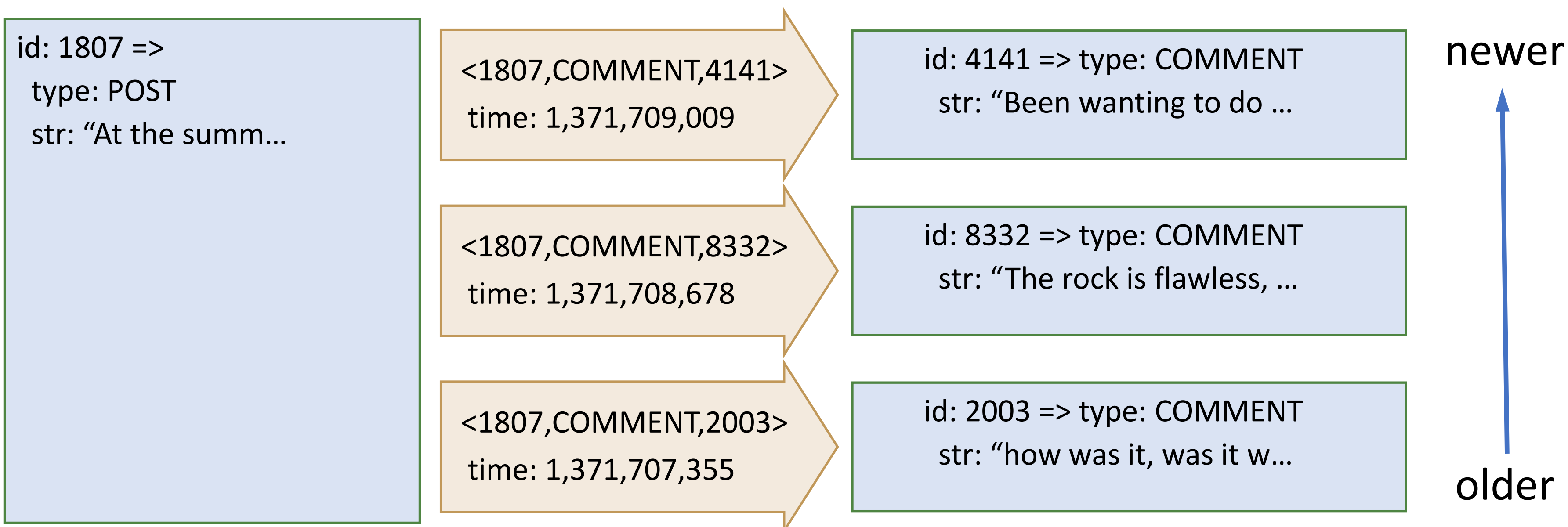
Objects and associations

- Objects, associations stored in separate database tables
  - Objects identified by unique 64-bit IDs
    - Each object has type field and other data fields
  - Associations identified by <id1, type, id2>
    - Associations have a time field and other data fields



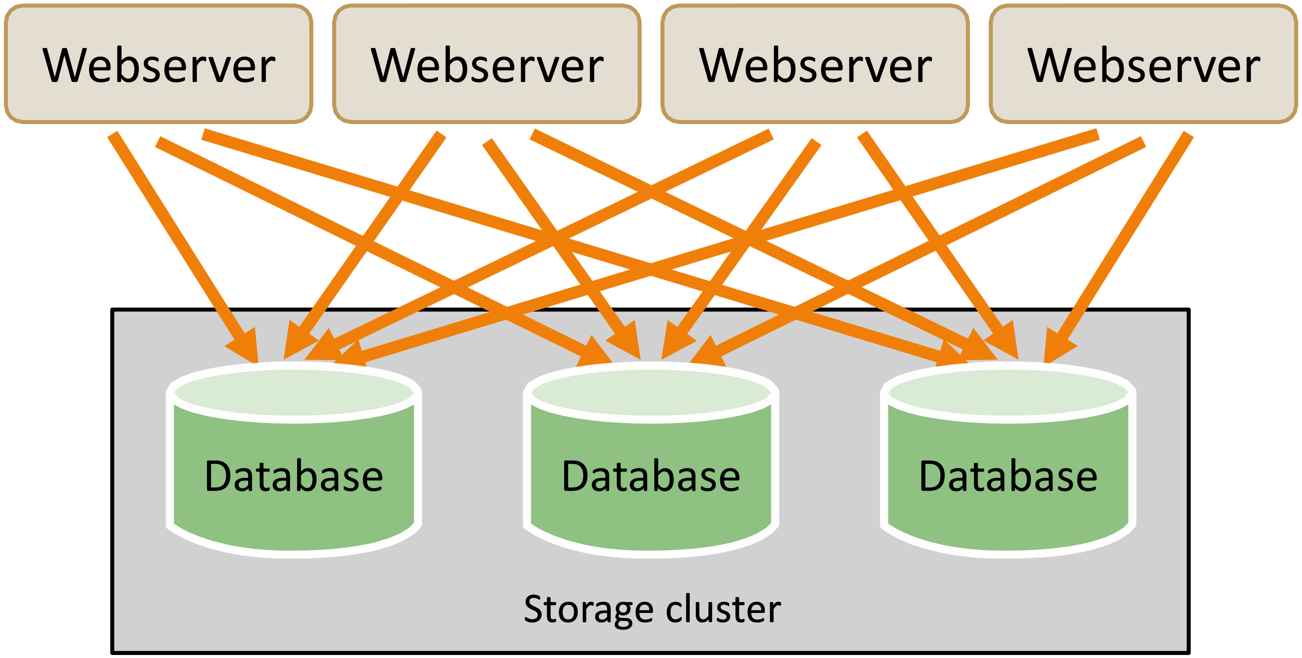
Association lists

- Association queries often require returning a list of associations:
  - assoc_get(1807, COMMENT)
    returns <1807, COMMENT, *>, ordered by time



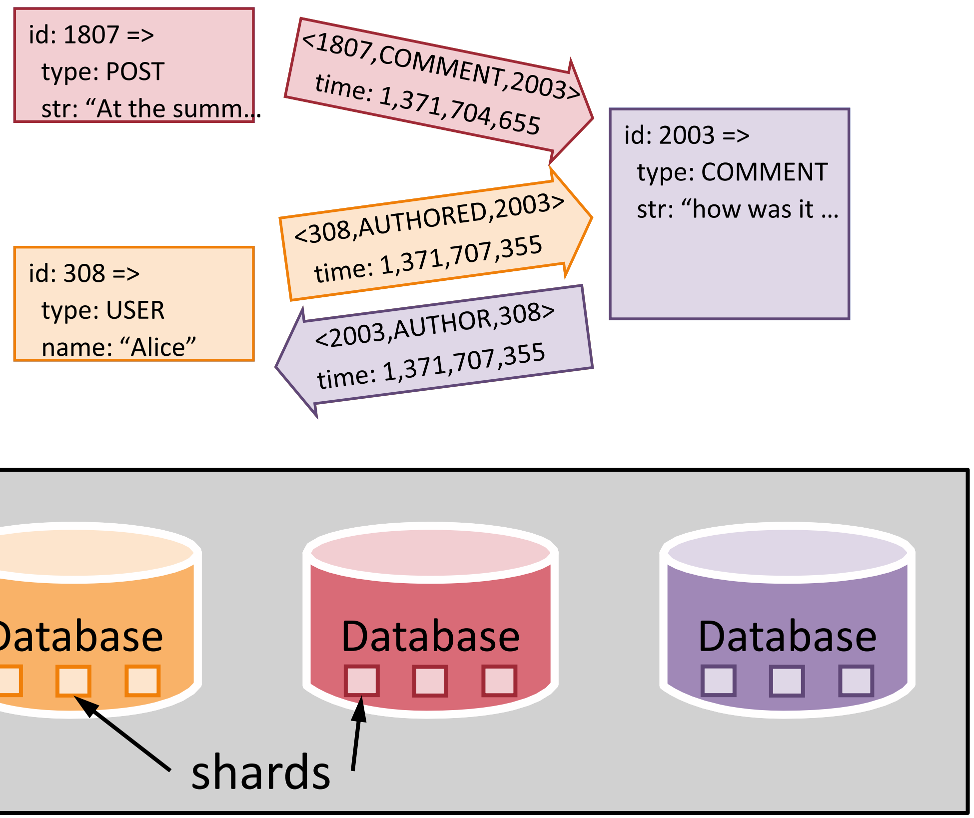
Early days (pre-caching)

- Just a few databases were enough to support the load



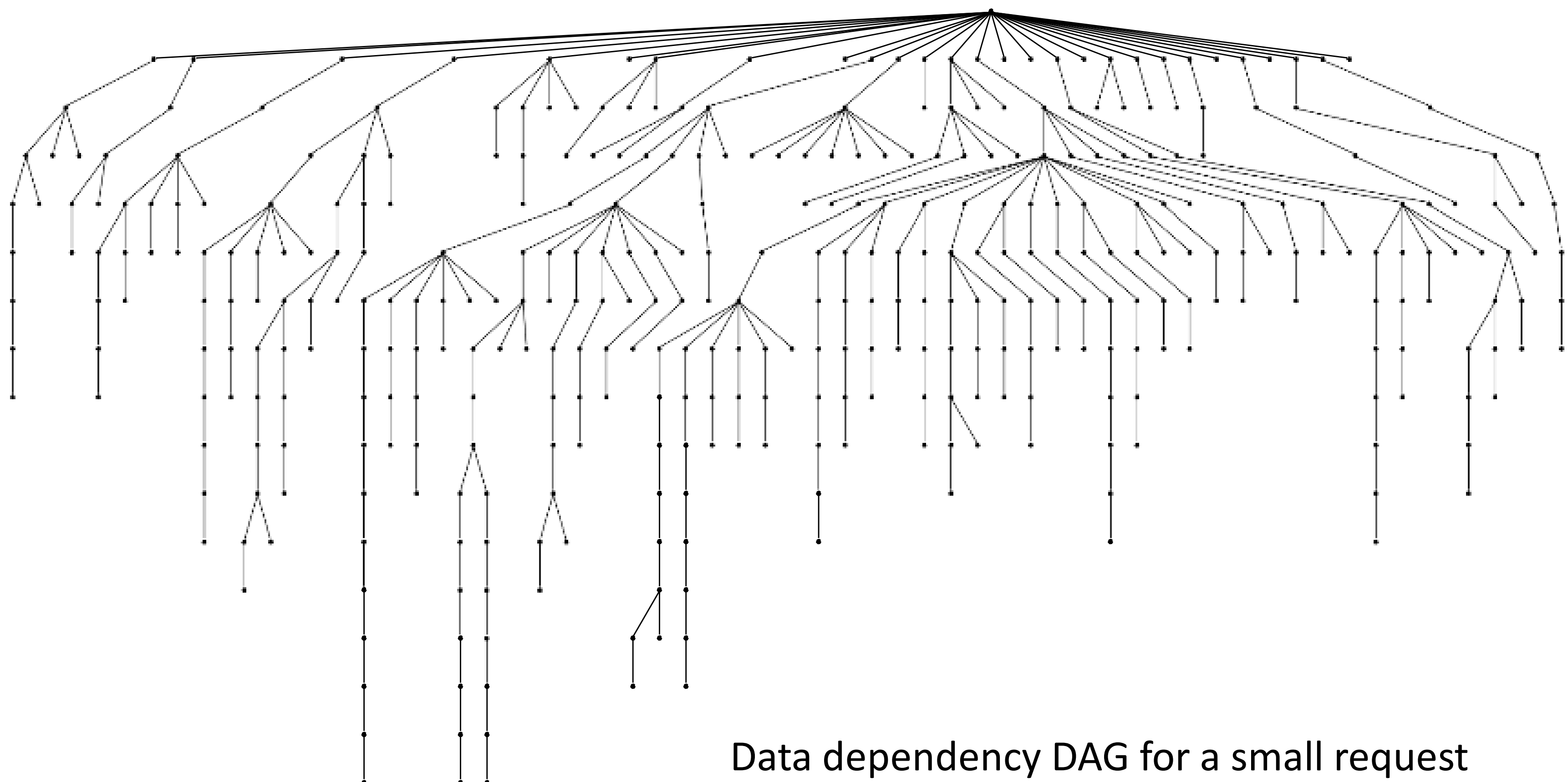
Sharding data across databases

- Data sharded by object id randomly across databases
  - Object and its outgoing associations stored in same shard
  - Association queries for an object served from one shard



Problem

- High fanout and multiple rounds of data fetching
  - Each node issues a database request, poor locality



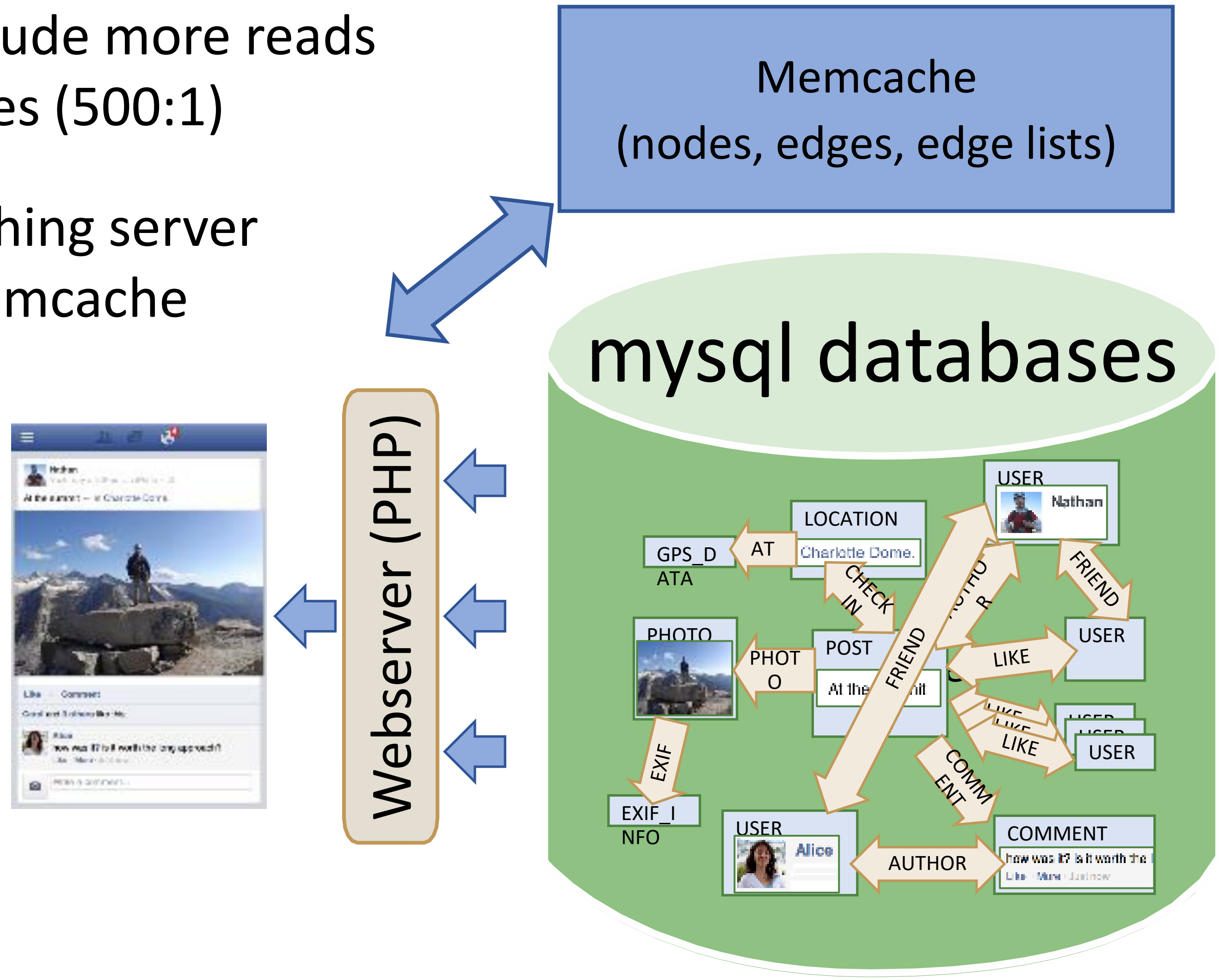
Data dependency DAG for a small request

Scaling Memcache in 4 "easy" steps

0	No Memcache servers (pre-memcache)
1	One Memcache server
2	Memcache servers in a cluster
3	Memcache servers in multiple clusters within a region
4	Geographically distributed clusters in multiple regions

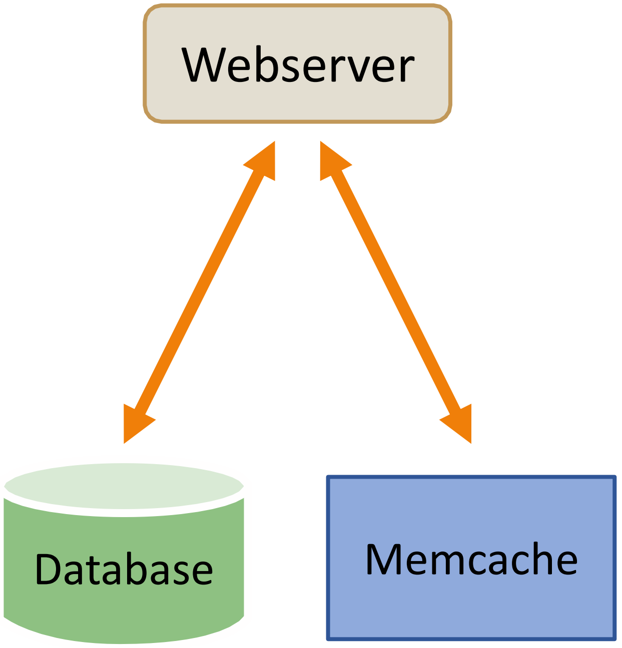
Cache social graph in Memcache

- Facebook has two orders of magnitude more reads than writes (500:1)
- Use a caching server called Memcache



Caching helps read performance

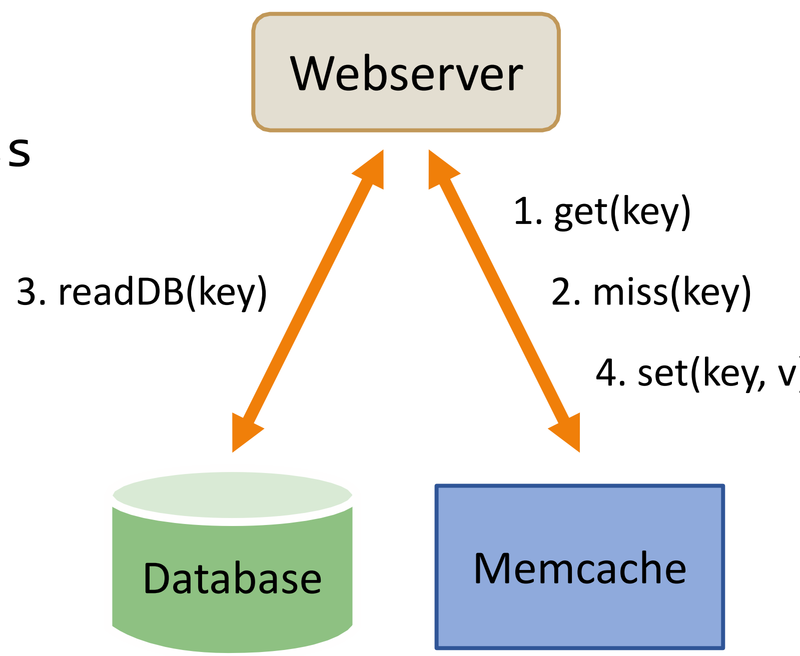
- Memache is a single-node, key-value store that uses a hash table to store key and values
  - Store nodes as keys
  - Stores edge lists as values
- Webserver (client) reads from Memcache
- Reduces load on database



Reading data from Memcache

- Use Memcache as a look-aside cache
  - Avoids any changes to Memcache

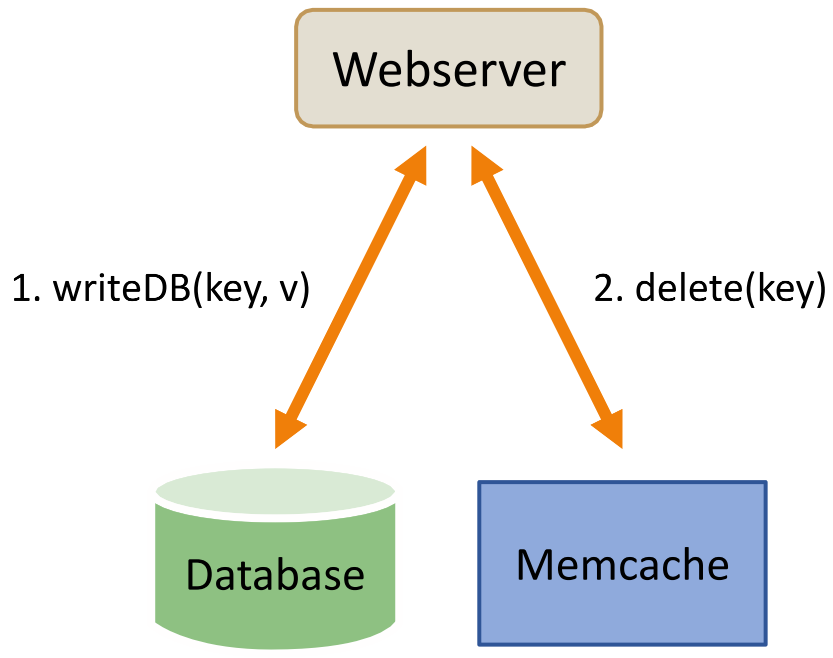
```
read(key)
  v = get(key)
  if (v == NULL) // read miss
    v = readDB(key)
    set(key, v)
```



Handling updates

- On a database update,
  Memcache needs to be synchronized

```
   write(key, v)
1:   writeDB(key, v)
2:   delete(key)
```



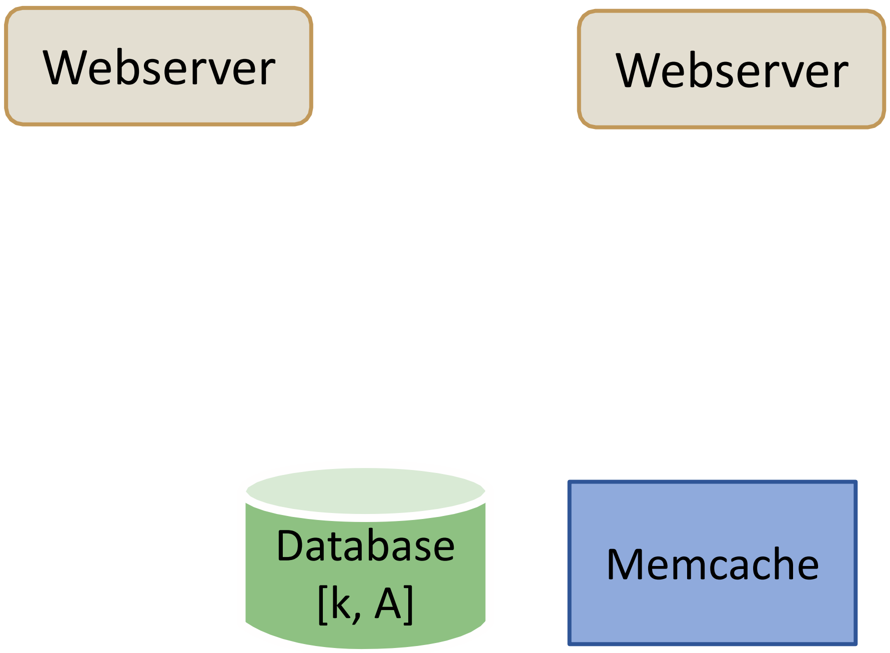
- Webserver uses
  delete (cache invalidation)
  instead of set (cache update)
  - Why is this important?
  - What if another Webserver issues read() between 1 and 2?

Understanding caching strategy

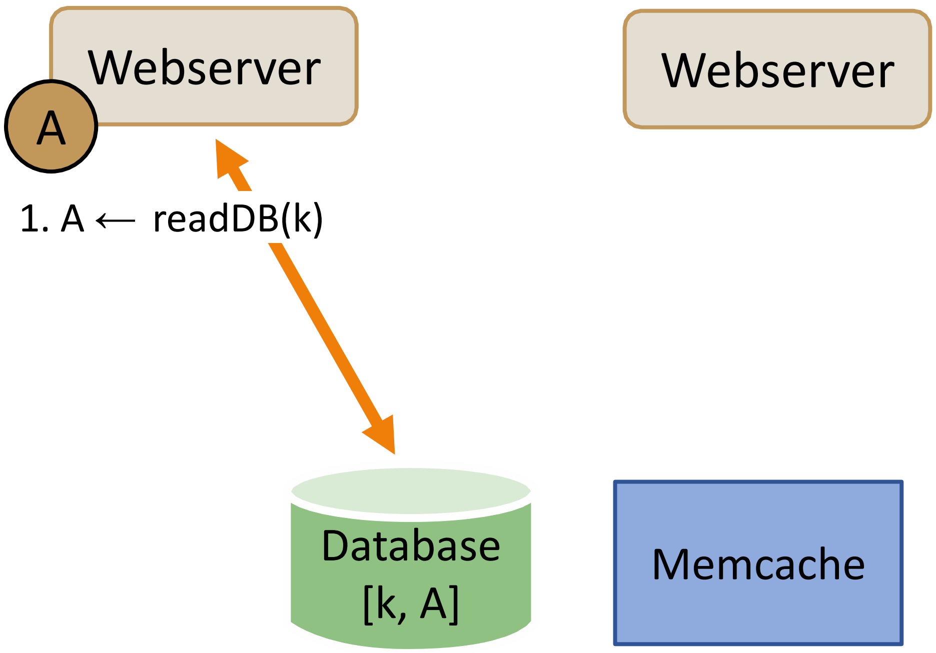
- Will caching increase write performance?
  - No, writes need to be sent to database (and deletes to Memcache)
- Why not use a write-back cache?
  - May cause inconsistency with multiple caches (discussed later)
- Will caching increase read performance?
  - Yes, and reads are dominant in the workload
- Any other benefits?
  - Helps reduce load on the database

Stale set consistency problem

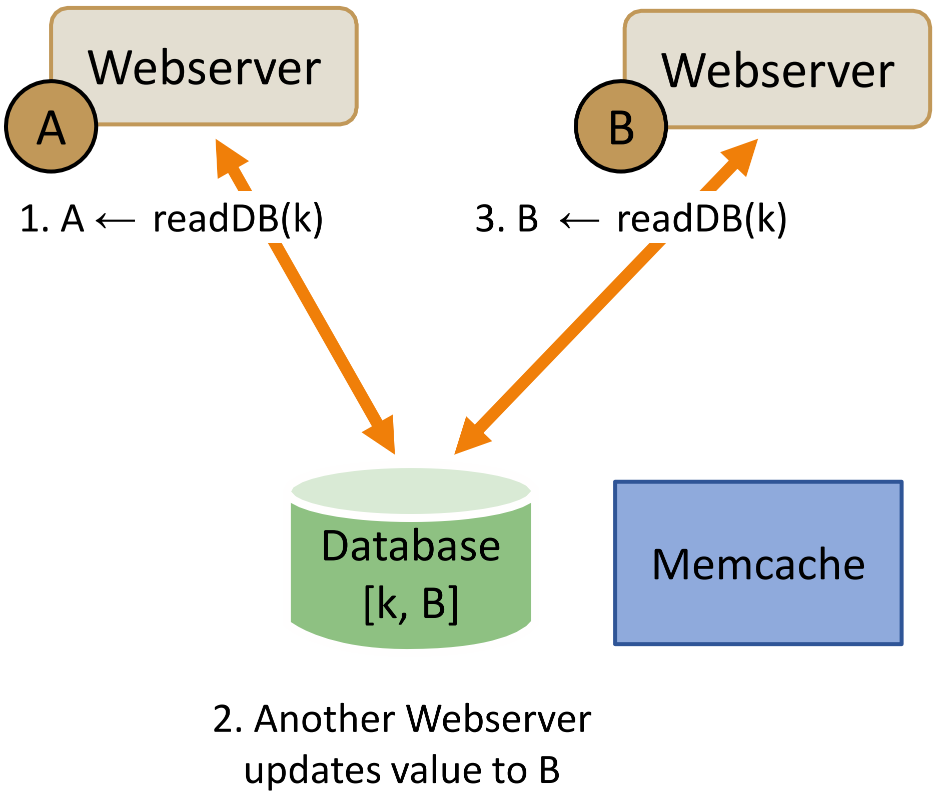
- Concurrent reads and updates to database can cause inconsistency between database and Memcache



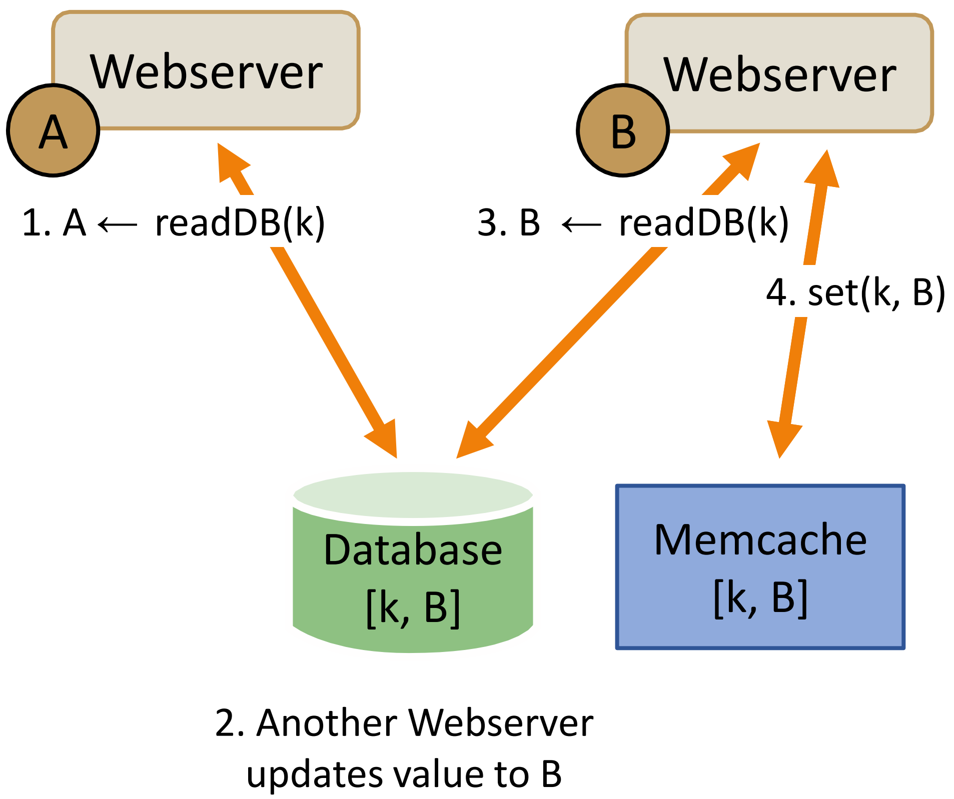
Stale set consistency problem



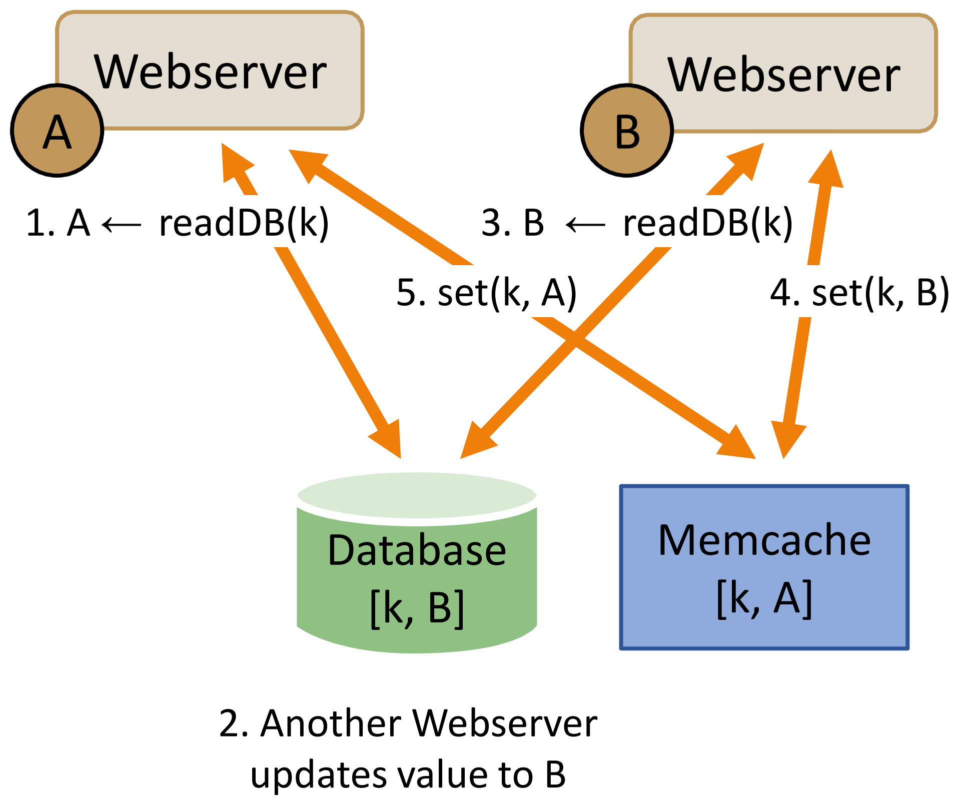
Stale set consistency problem



Stale set consistency problem



Stale set consistency problem

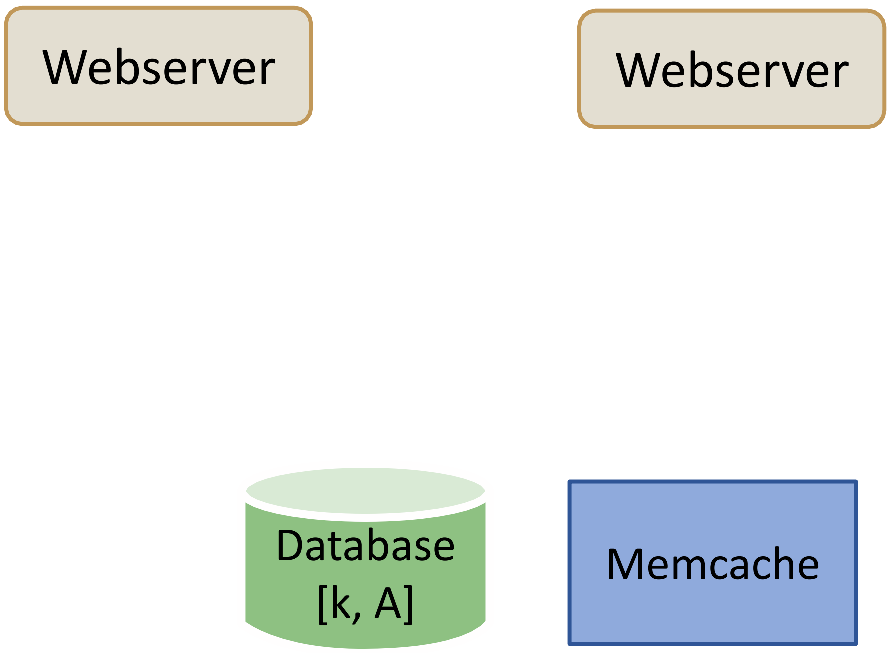


Memcache and database are inconsistent

Why does this problem occur?

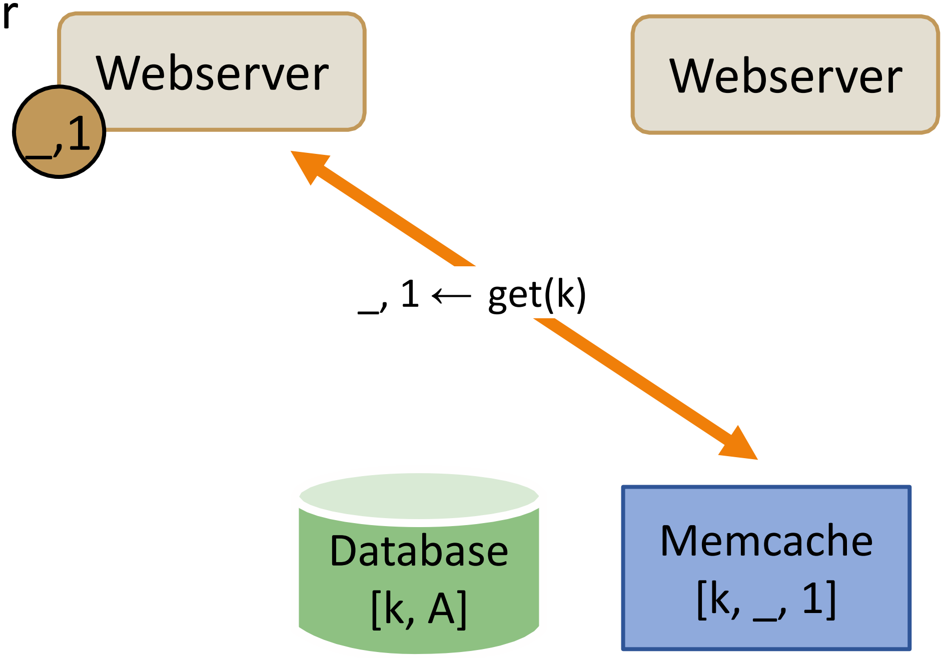
Avoiding stale set problem

- Extend Memcache protocol with “leases”



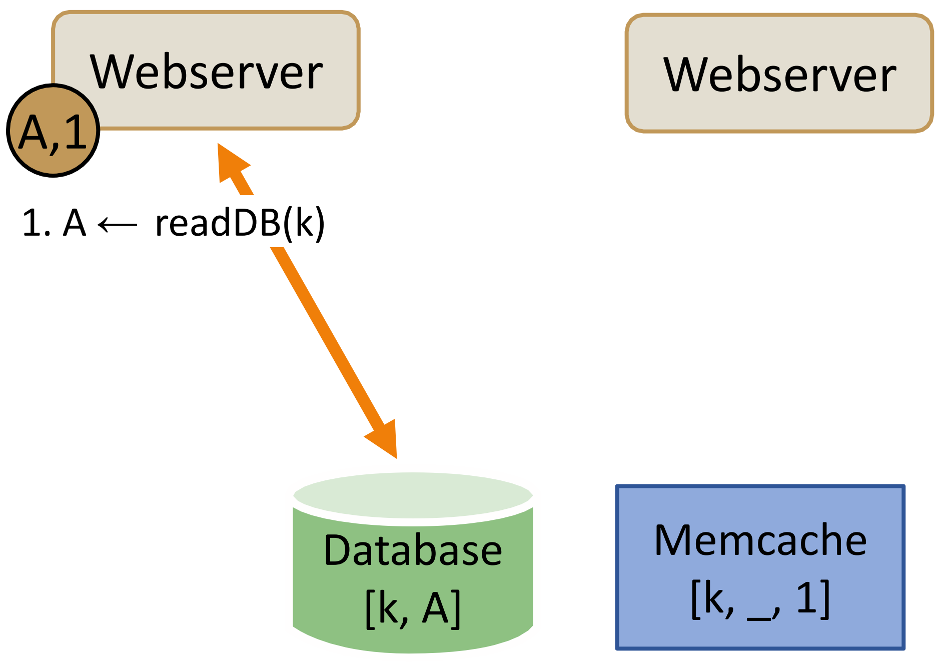
Avoiding stale set problem

- Extend Memcache protocol with “leases”
  - Step 1: On read-miss, Memcache returns new lease-id to Webserver



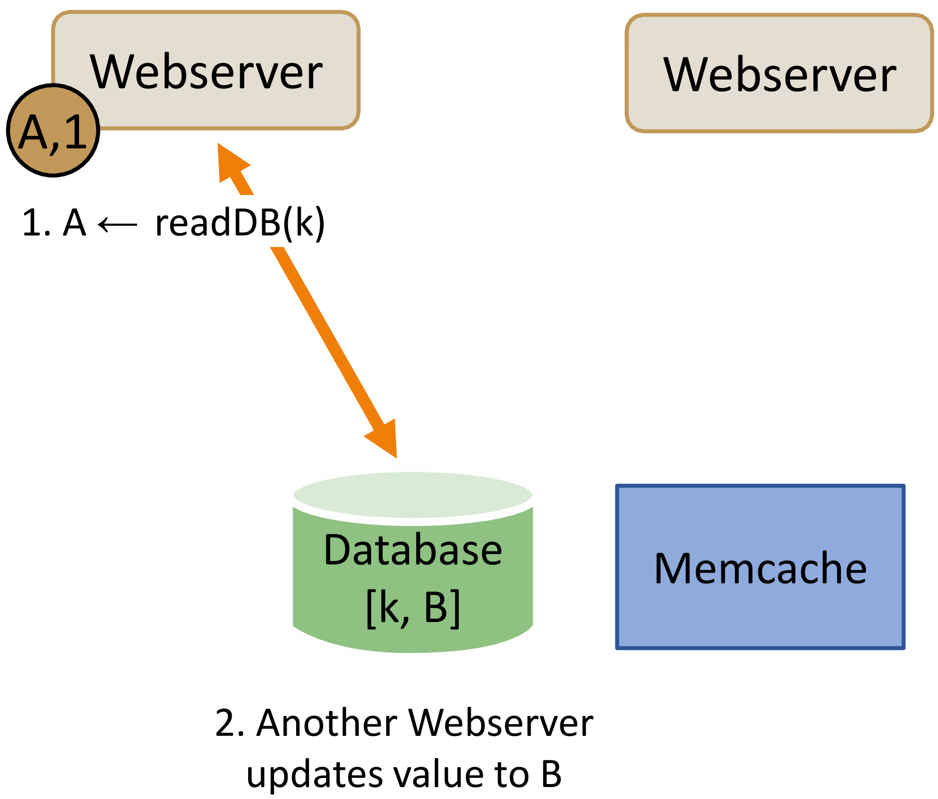
Avoiding stale set problem

- Extend Memcache protocol with "leases"
  - Step 1: On read-miss, Memcache returns new lease-id to Webserver, Webserver reads data



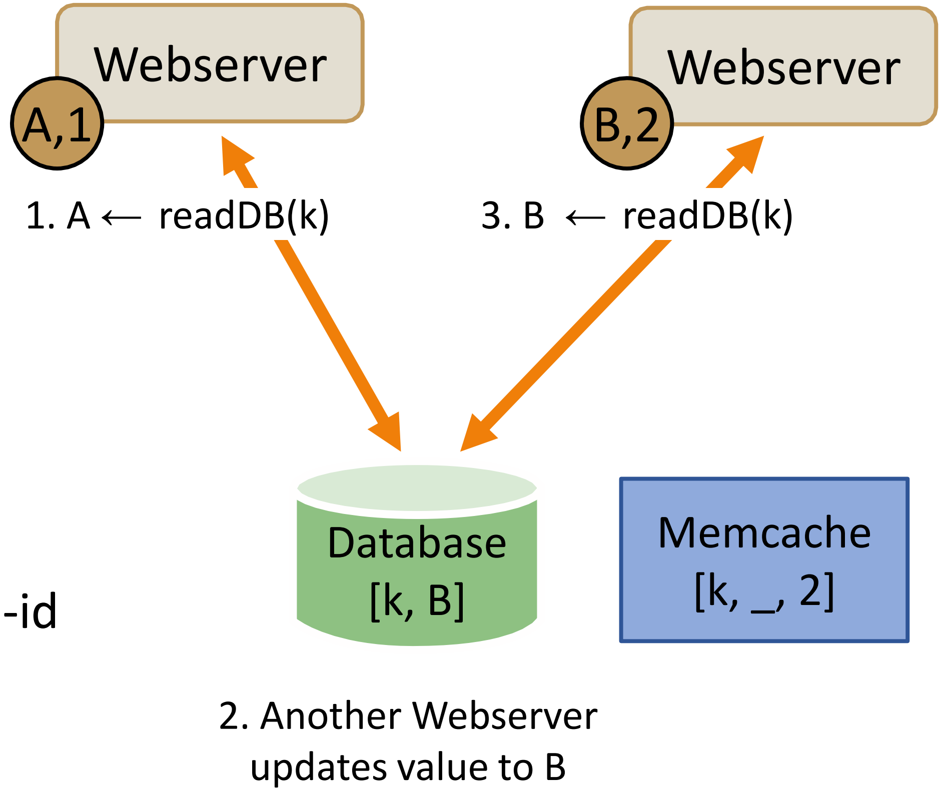
Avoiding stale set problem

- Extend Memcache protocol with “leases”
  - Step 1: On read-miss, Memcache returns new lease-id to Webserver, Webserver reads data
  - Step 2: On update, Memcache invalidates lease-id



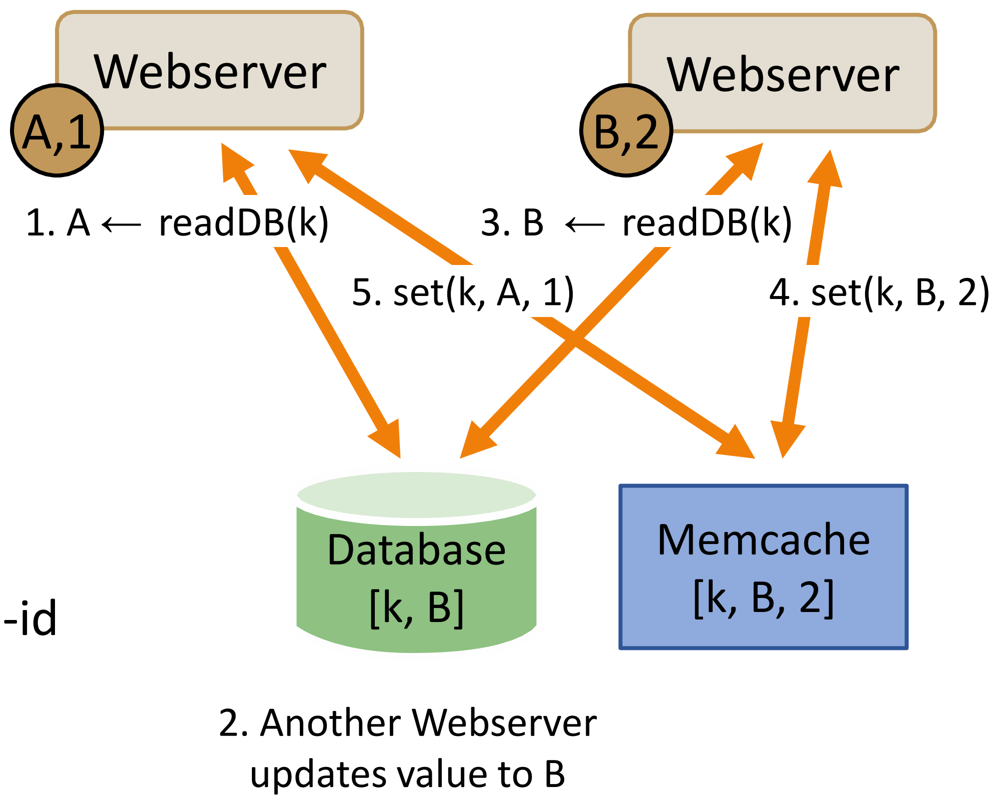
Avoiding stale set problem

- Extend Memcache protocol with “leases”
  - Step 1: On read-miss, Memcache returns new lease-id to Webserver, Webserver reads data
  - Step 2: On update, Memcache invalidates lease-id
  - Step 3: same as Step 1
  - Steps 4, 5: On set, Memcache checks Webserver provided lease-id



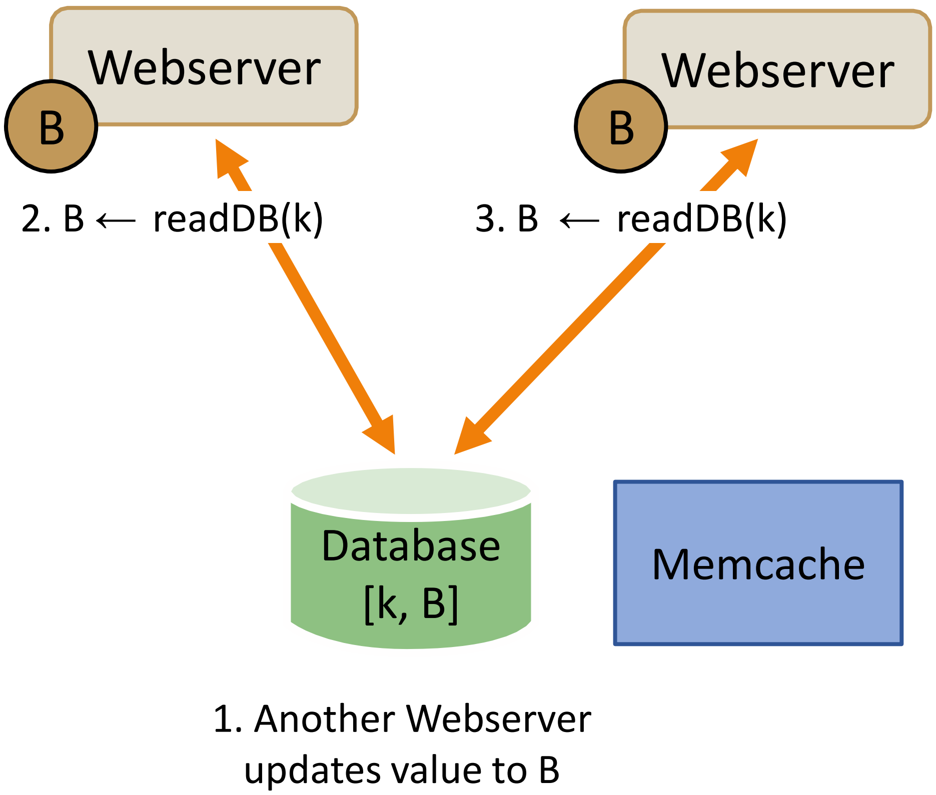
Avoiding stale set problem

- Extend Memcache protocol with "leases"
  - Step 1: On read-miss, Memcache returns new lease-id to Webserver, Webserver reads data
  - Step 2: On update, Memcache invalidates lease-id
  - Step 3: same as Step 1
  - Steps 4, 5: On set, Memcache checks Webserver provided lease-id
    - Step 4 allowed
    - Step 5 disallowed



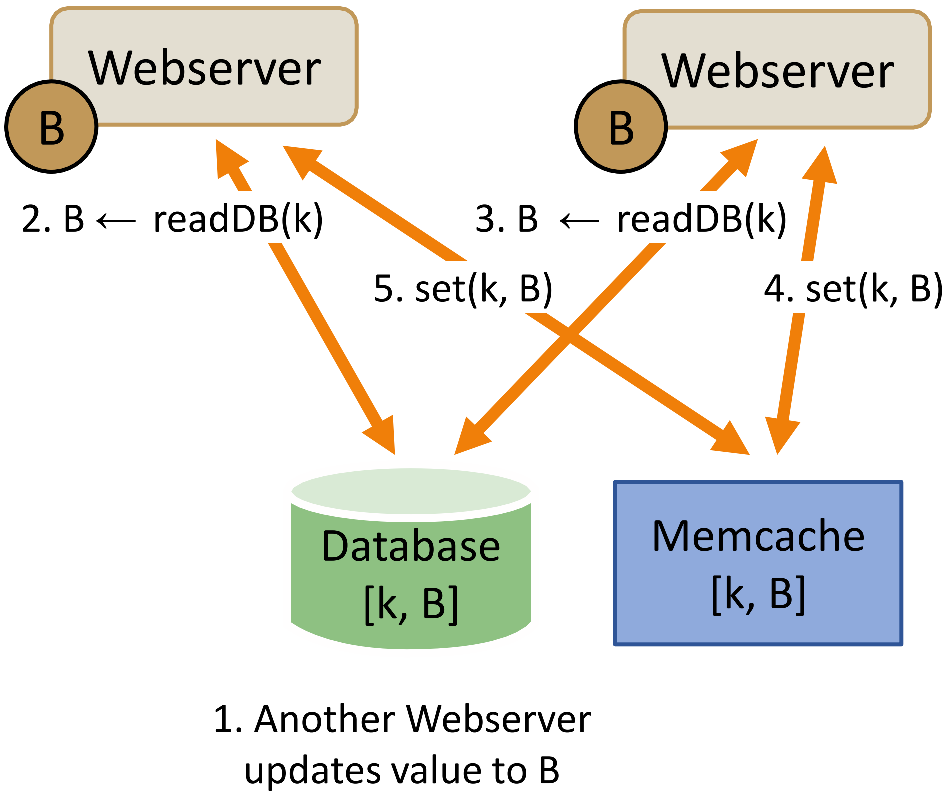
Thundering herd problem

- Say a key is read heavily
- Step 1: key is updated
- Steps 2, 3: all reads will cause read-misses, database accesses



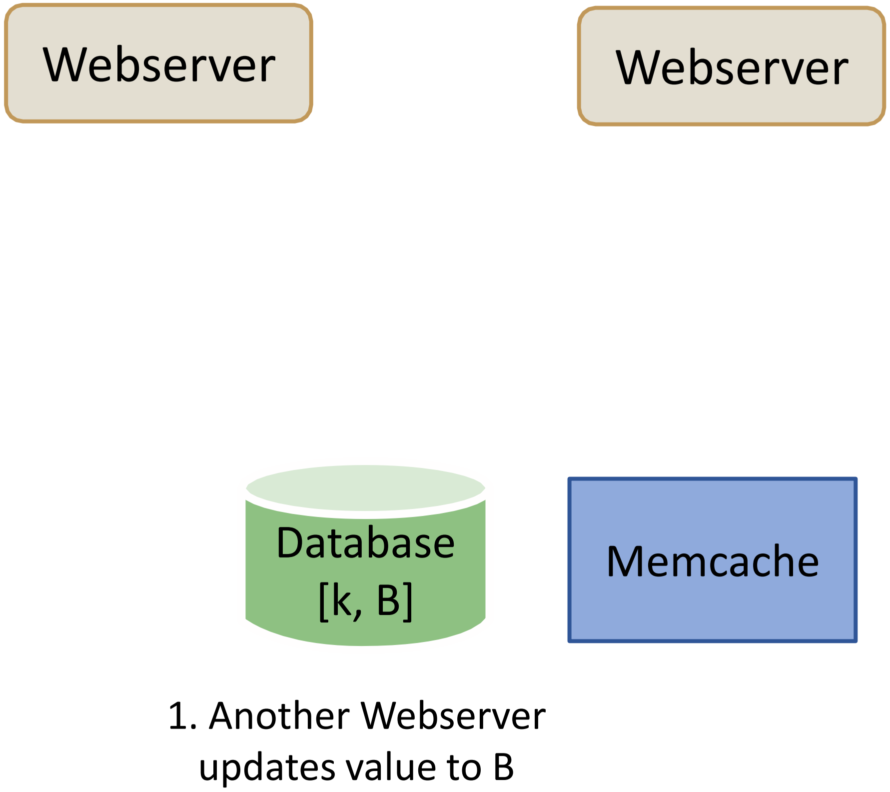
Thundering herd problem

- Say a key is read heavily
- Step 1: key is updated
- Steps 2, 3: all reads will cause read-misses, database accesses
- Steps 4, 5: until key is cached again



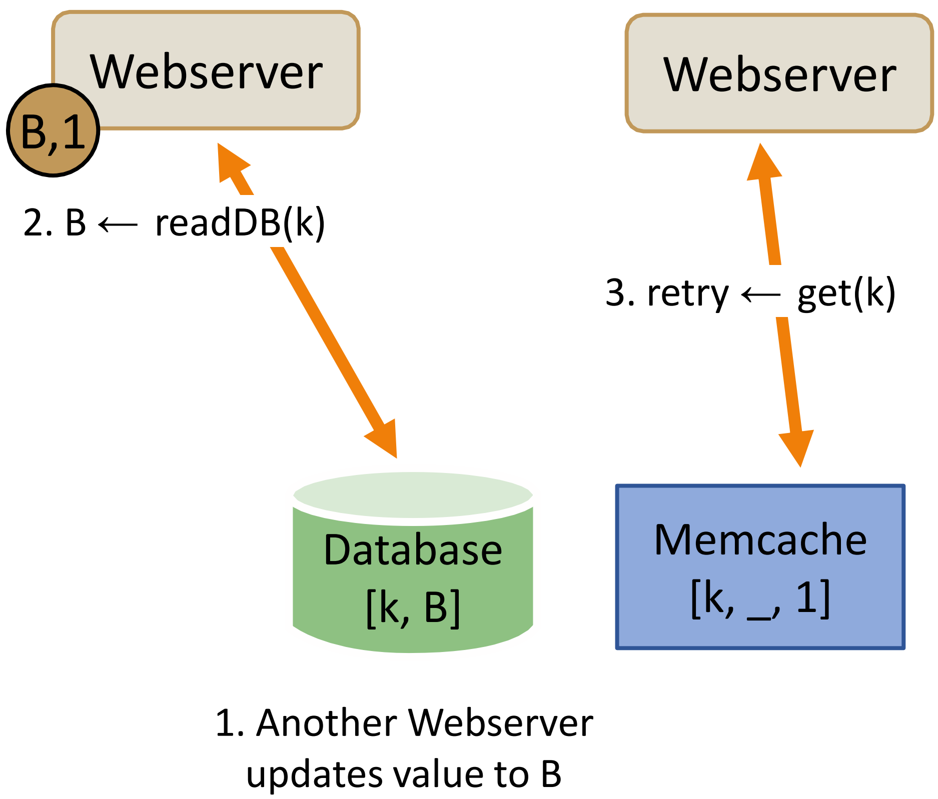
Avoiding thundering herd

- Limit rate at which leases are returned on read miss



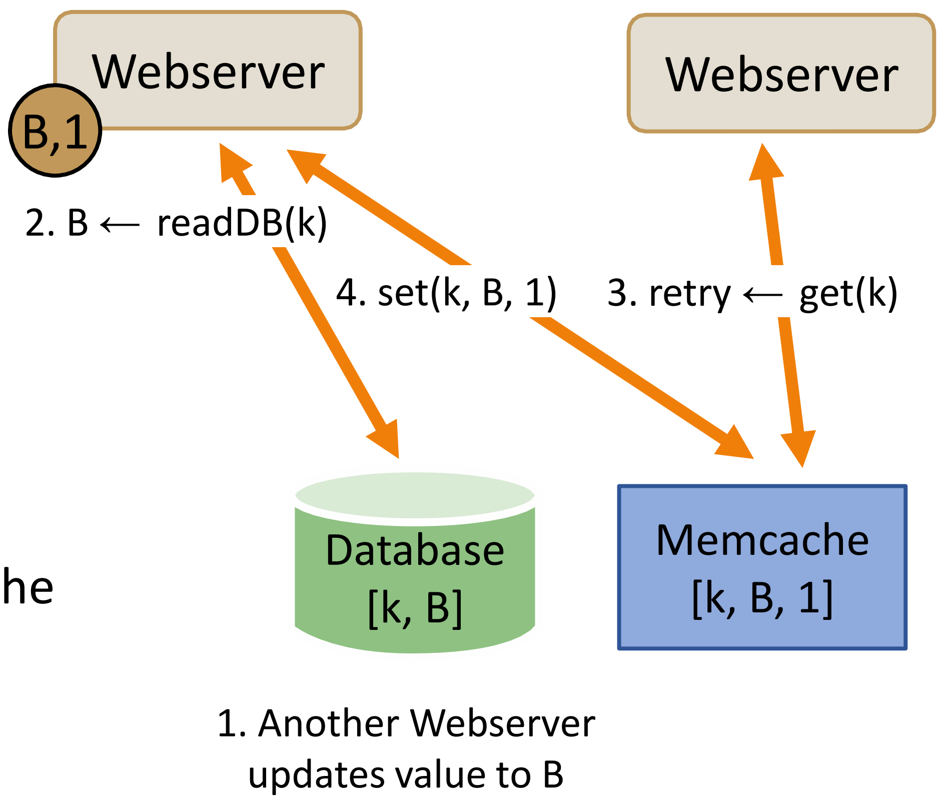
Avoiding thundering herd

- Limit rate at which leases are returned on read miss
  - Step 2: On read-miss, Memcache returns new lease-id to Webserver, Webserver reads data
  - Step 3: On next get, within 10 seconds, don't return lease, instead return notification to retry in a few milliseconds



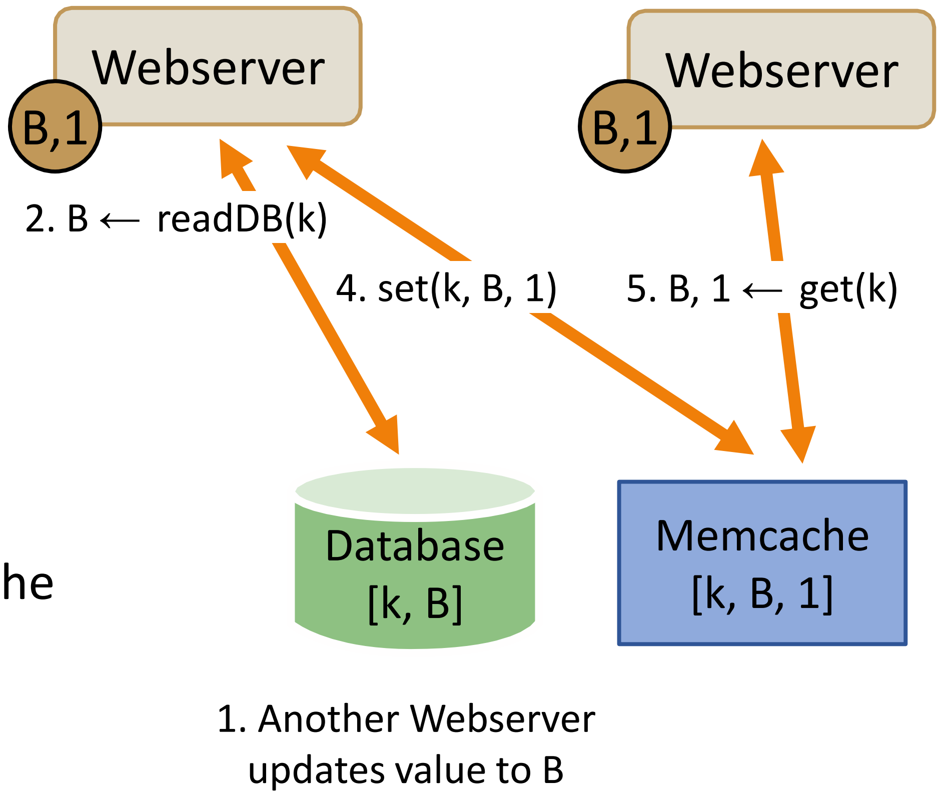
Avoiding thundering herd

- Limit rate at which leases are returned on read miss
  - Step 2: On read-miss, Memcache returns new lease-id to Webserver, Webserver reads data
  - Step 3: On next get, within 10 seconds, don't return lease, instead return notification to try again in a few milliseconds
  - Step 4: On set, update cache



Avoiding thundering herd

- Limit rate at which leases are returned on read miss
  - Step 2: On read-miss, Memcache returns new lease-id to Webserver, Webserver reads data
  - Step 3: On next get, within 10 seconds, don't return lease, instead return notification to try again in a few milliseconds
  - Step 4: On set, update cache
  - Step 5: On get again, return cached value

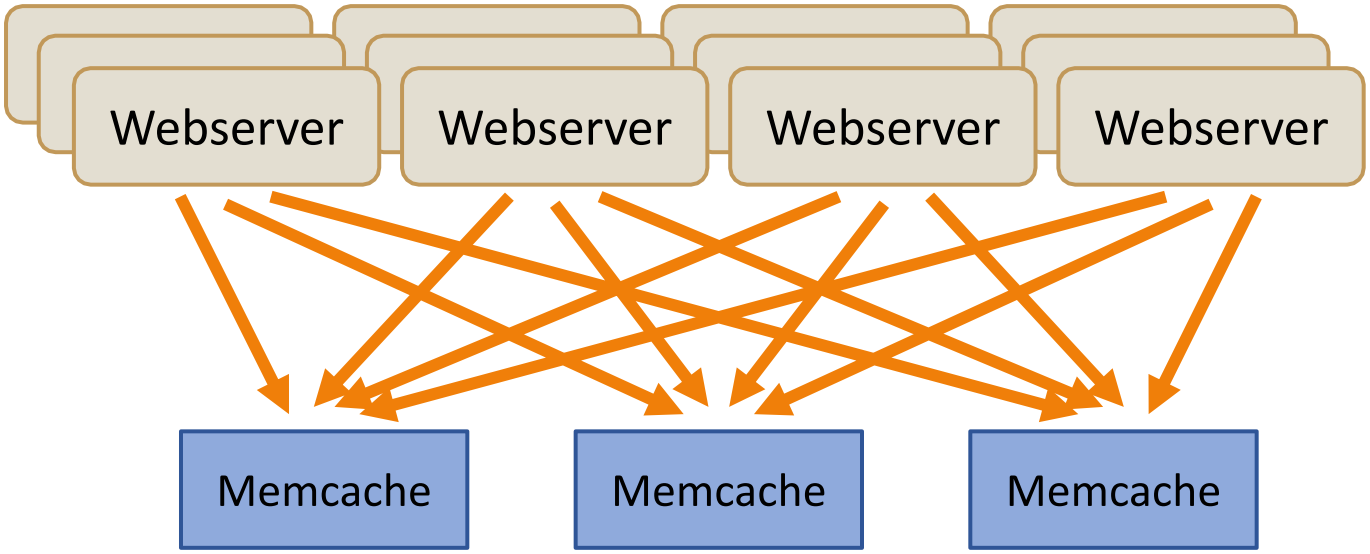


Scaling Memcache in 4 “easy” steps

0	No Memcache servers (pre-memcache)
1	One Memcache server
2	Memcache servers in a cluster
3	Memcache servers in multiple clusters within a region
4	Geographically distributed clusters in multiple regions

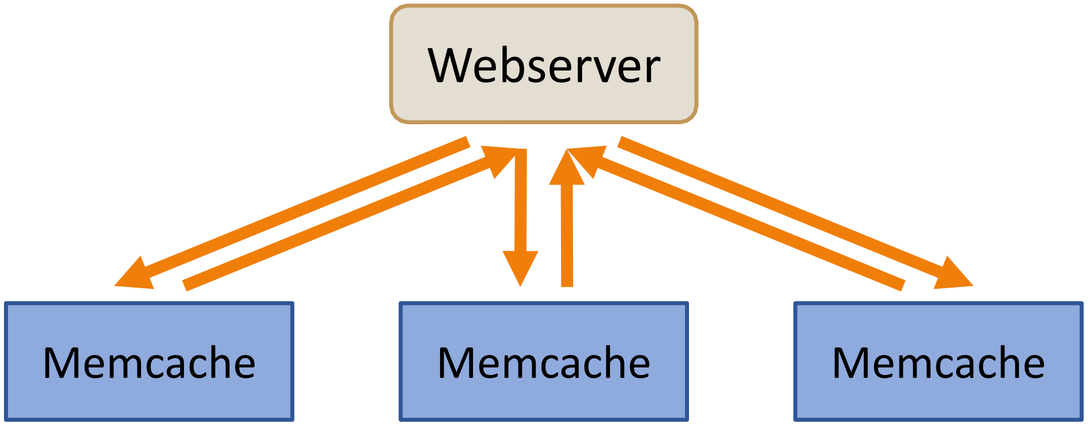
Need even more read capacity

- Use multiple Memcache servers
  - Items are sharded across Memcache servers using consistent hashing on the key, so any Webserver can find a cached key
- All Webservers talk to all Memcache servers



Problem: incast congestion

- For a user request, a Webserver may fetch 500+ keys from 100s of Memcache servers in parallel
  - Many simultaneous responses from Memcache servers may overwhelm networking resources, cause responses to be dropped
- Solution: Limit the number of outstanding requests by Webserver to Memcache with a sliding window (e.g., TCP)
  - Larger windows result in more congestion, smaller windows result in more network round trips

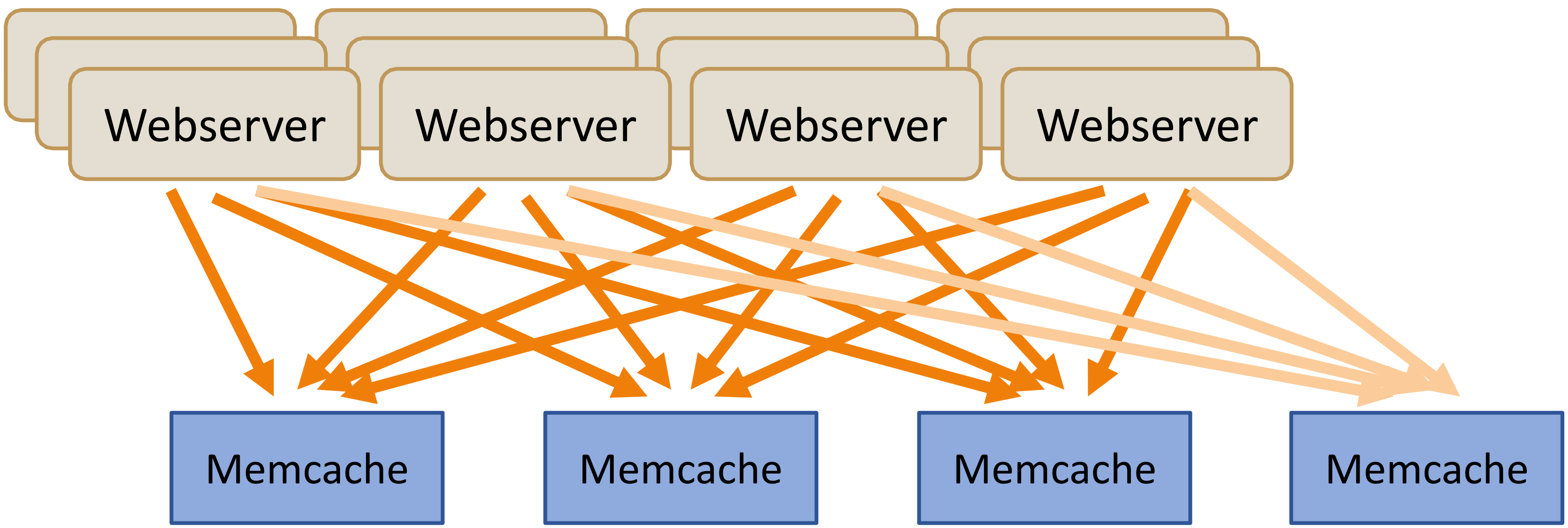


Scaling Memcache in 4 “easy” steps

0	No Memcache servers (pre-memcache)
1	One Memcache server
2	Memcache servers in a cluster
3	Memcache servers in multiple clusters within a region
4	Geographically distributed clusters in multiple regions

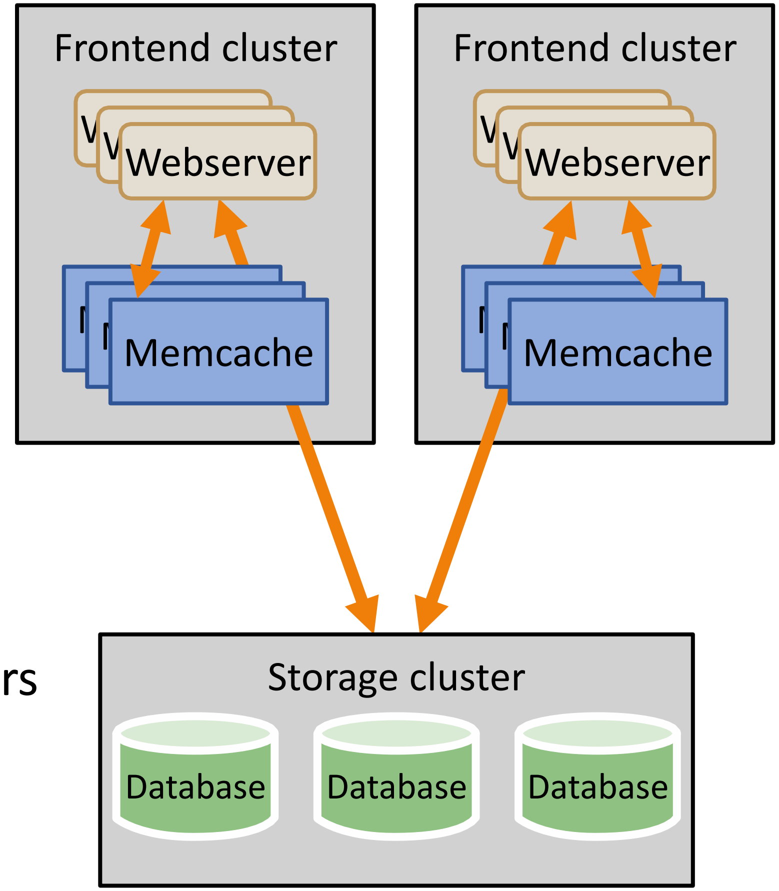
Scaling problem

- All-to-all communication between Webservers and Memcache servers limits horizontal scaling
- Communication ↑ with more Webservers, Memcaches
- Some Memcaches become hotspots, slowing all requests



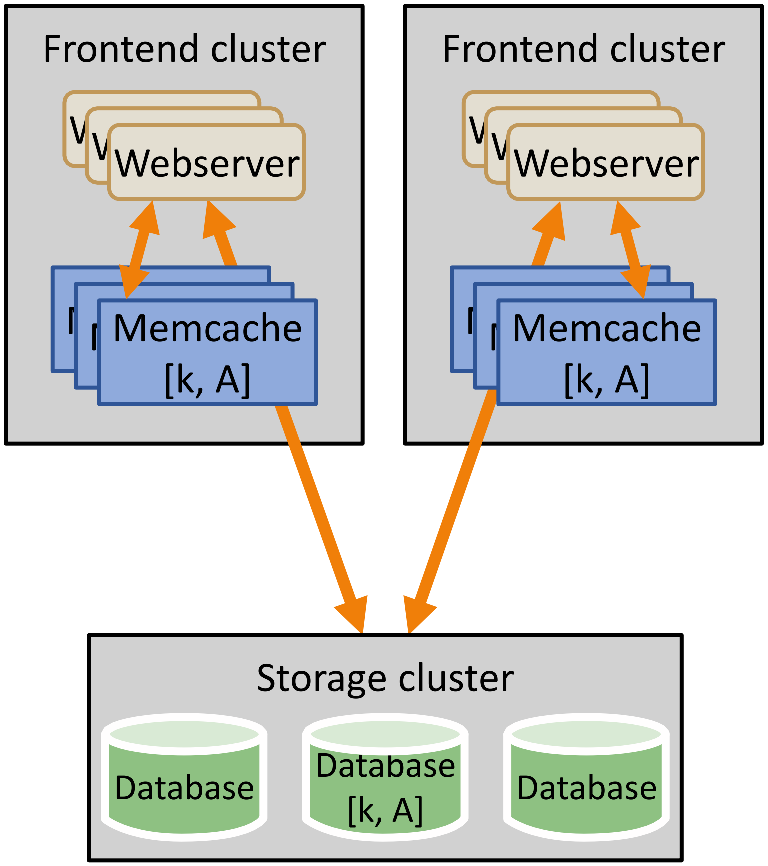
Solution: use multiple clusters

- Each cluster caches data in its own Memcache servers
  - A Webserver only accesses the Memcache servers in its cluster
  - All the clusters are backed by a single storage cluster
- Pros:
  - Helps limit # of servers per cluster
  - Hot keys get cached in multiple clusters
- Cons:
  - Fewer unique keys can be cached across all clusters



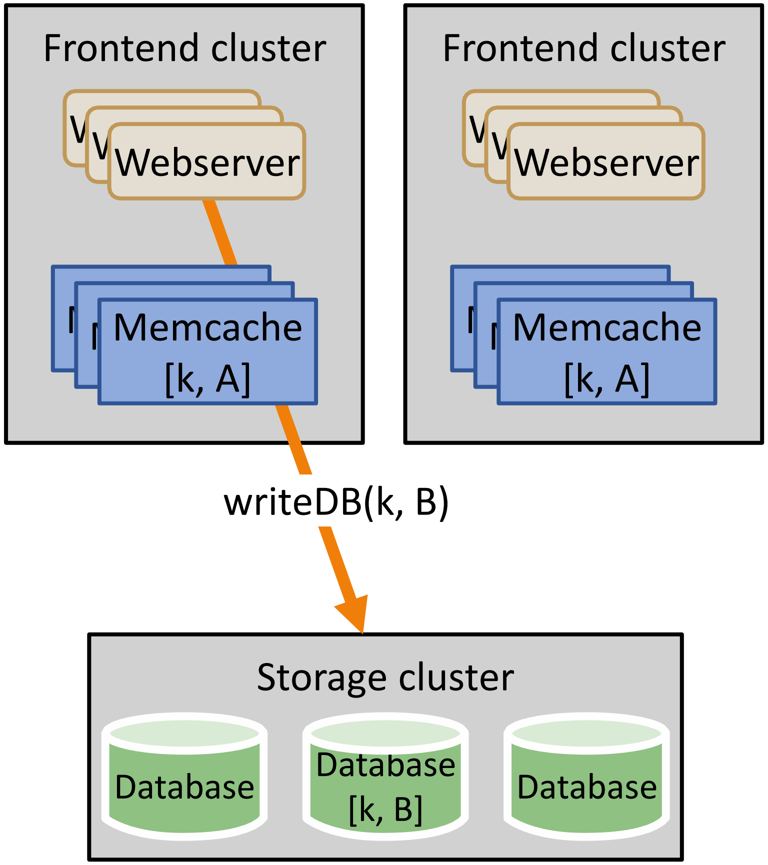
Cache consistency problem

- Same data may be cached in the Memcache servers in different clusters
- Need to keep caches consistent when data is updated



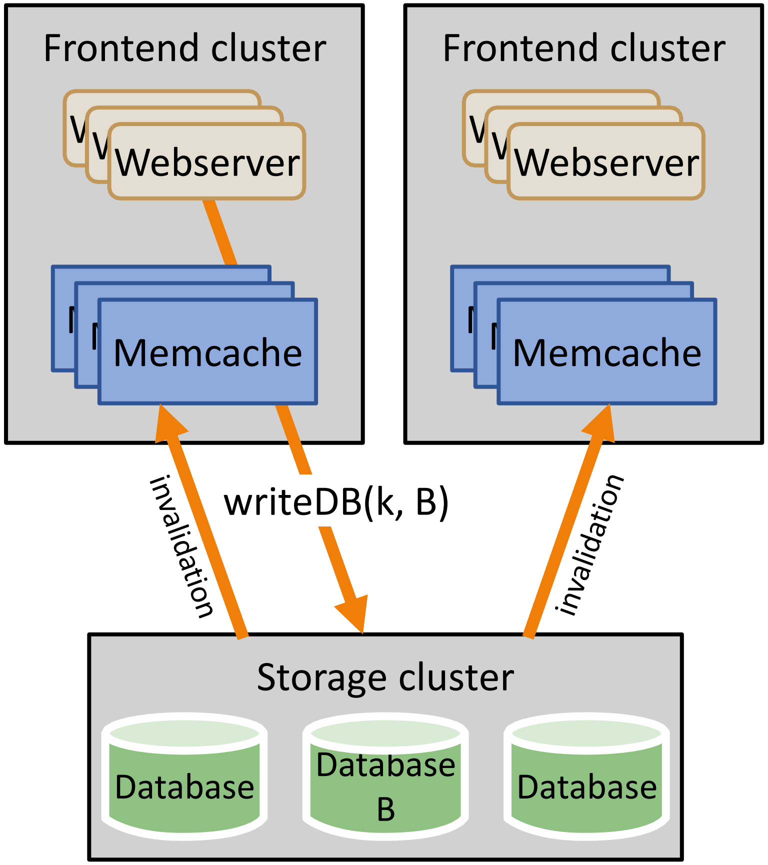
Solution: use invalidations

- When Webserver updates key in database



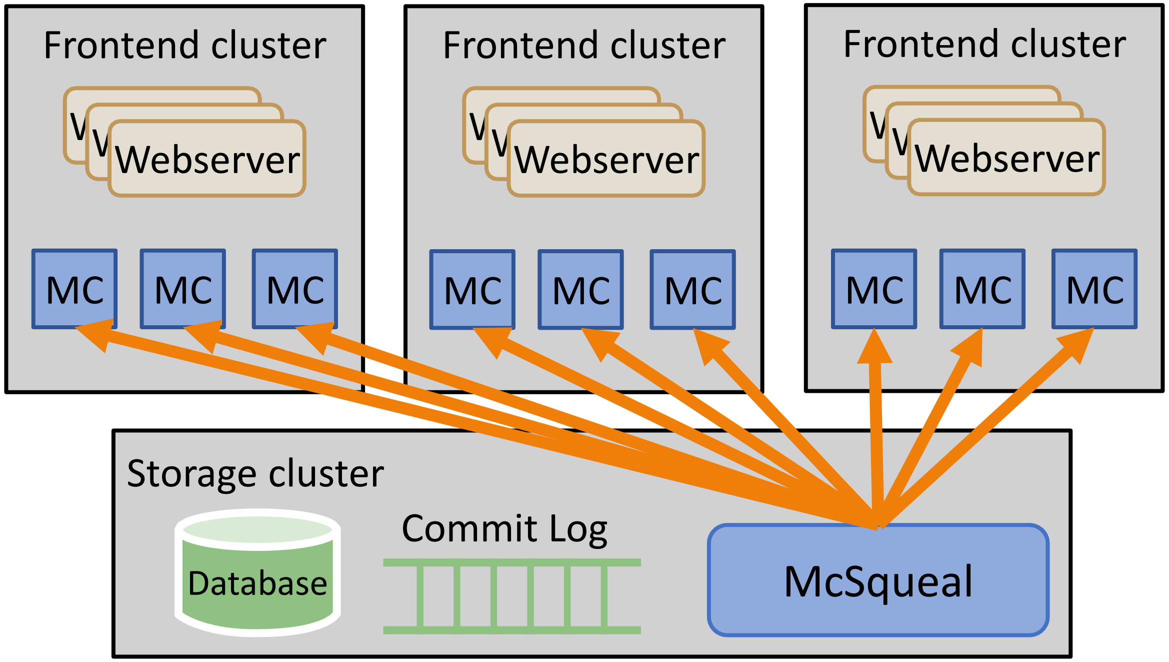
Solution: use invalidations

- When Webserver updates key in database
  - Storage cluster invalidates key in the Memcache servers in all clusters
- What if invalidations are lost?



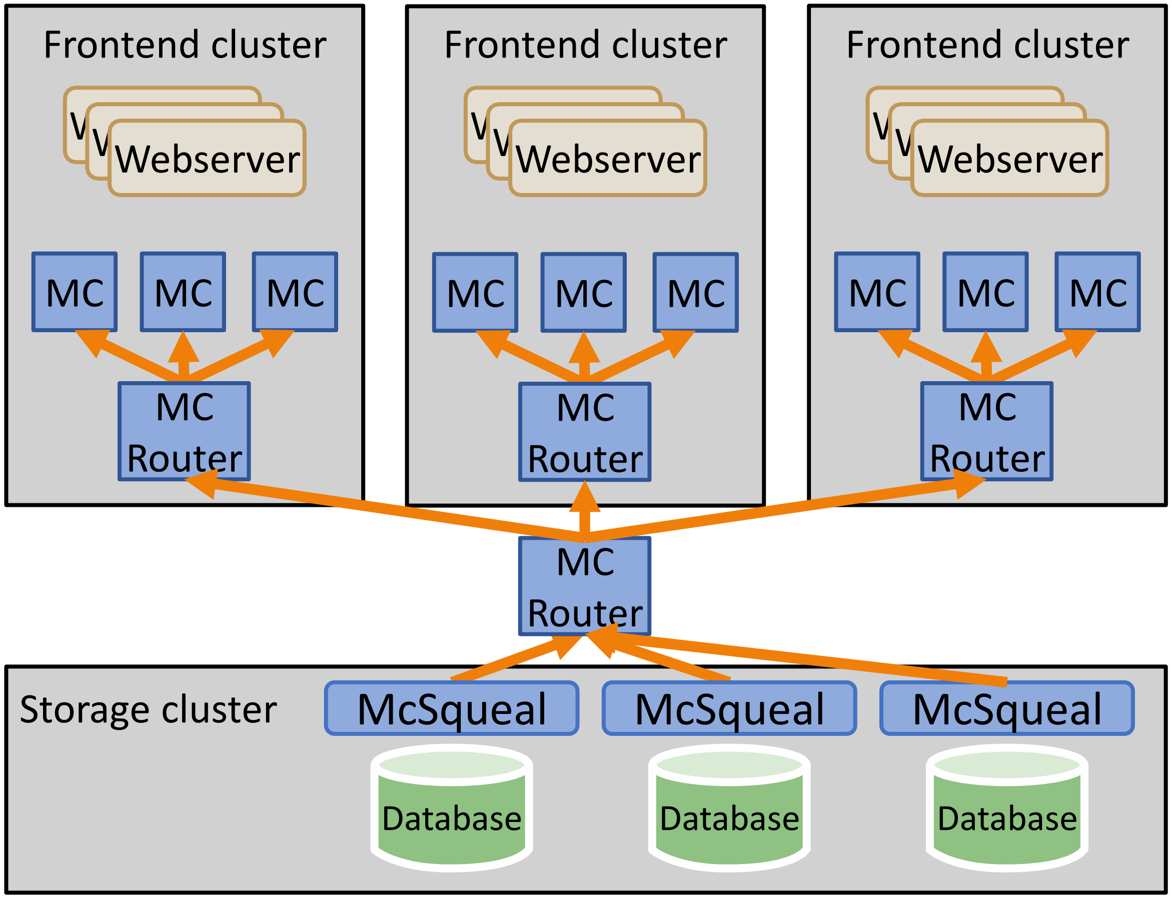
Reliable invalidations

- Storage cluster logs invalidations for updates, before sending them to all Memcache servers
- If frontend cluster fails, invalidation daemons (McSqueal) resend invalidations from log to resynchronize caches



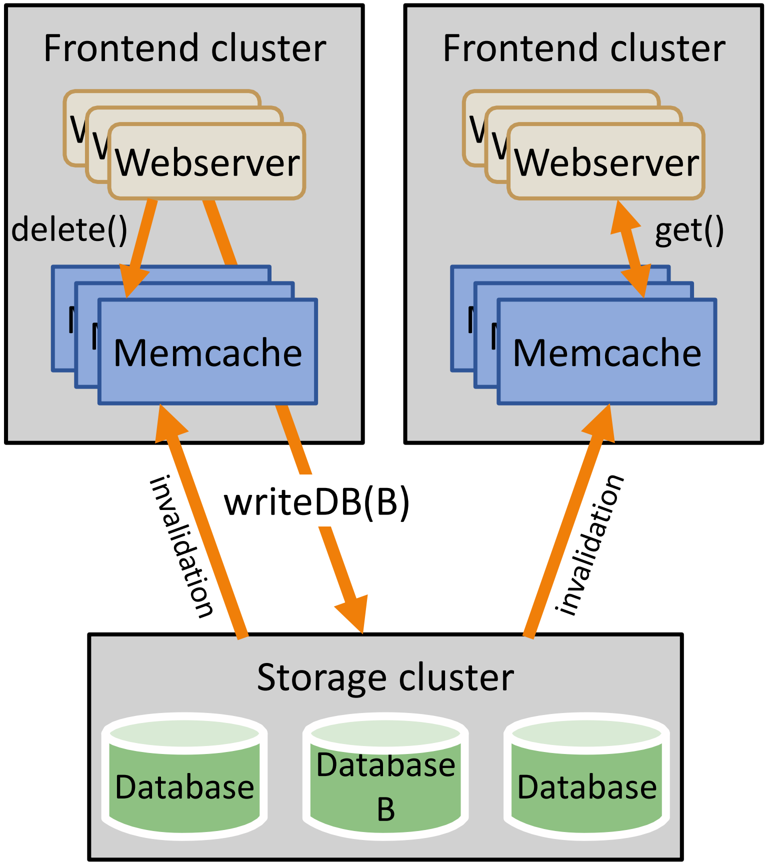
Scalable invalidations

- Invalidations are batched and routed hierarchically to reduce network bandwidth



Cache consistency and performance

- Webserver updates key in database directly
  - Database performs updates in a total order, so no conflicts, then sends invalidations
  - For read-your-write consistency, Webserver deletes key in the Memcache server of the local cluster as well
- For performance, updates do not wait for invalidations to complete
  - So get() at other clusters may return stale cached value for a short time



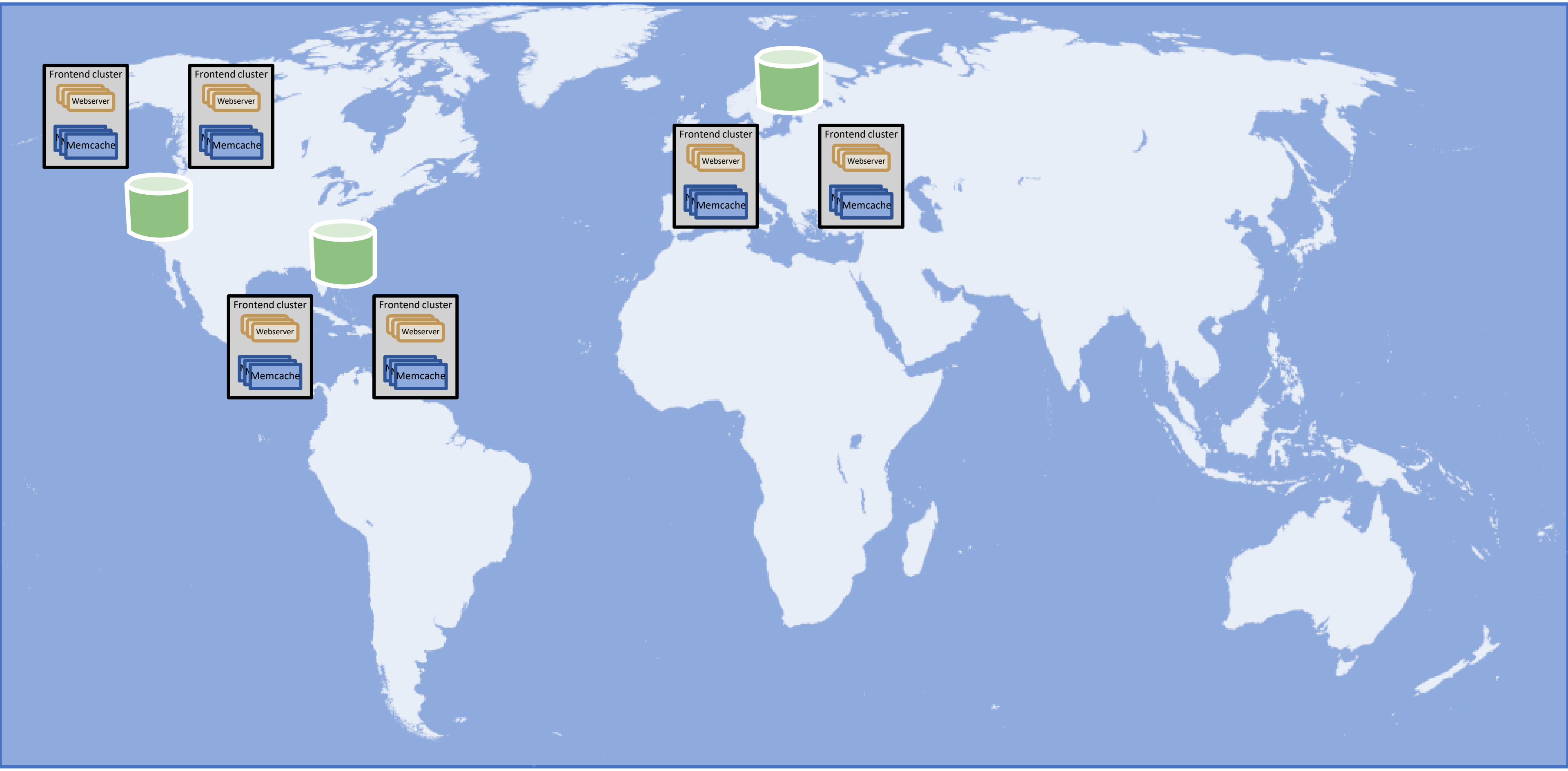
Why this consistency model?

- Writes are ordered and slow but not lost
  - E.g., “like” count is correct
- Caches are eventually consistent
  - Leases and reliable invalidates ensure that caches do not serve stale data forever
- Reads are fast but may return stale data
  - This is facebook!, data is news feed, likes, etc.
  - Most people will not notice or care about slightly stale data
  - Next refresh will fetch up-to-date data

Scaling Memcache in 4 “easy” steps

0	No Memcache servers (pre-memcache)
1	One Memcache server
2	Memcache servers in a cluster
3	Memcache servers in multiple clusters within a region
4	Geographically distributed clusters in multiple regions

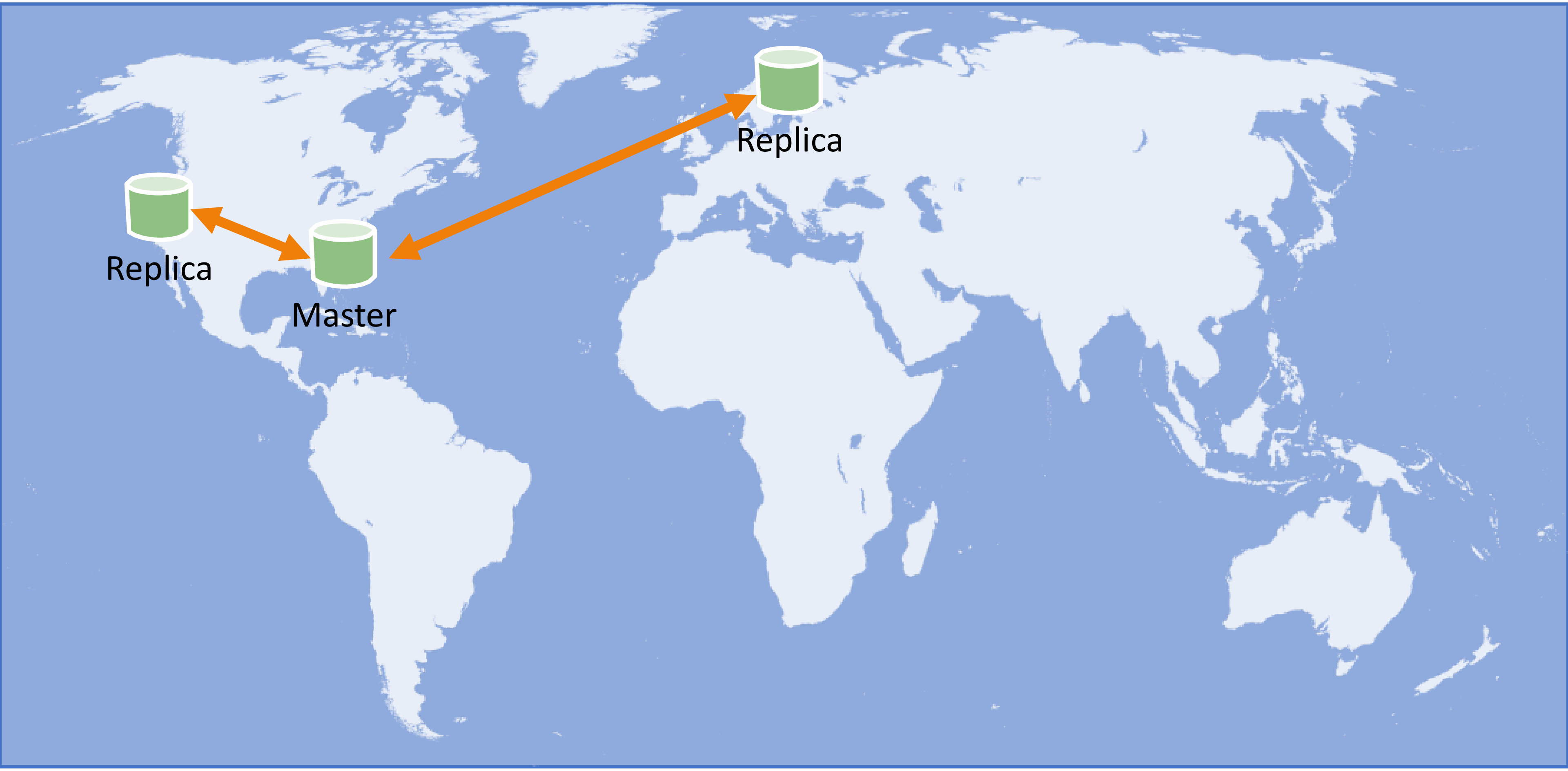
Geographically distributed clusters



Each region has a separate database replica

Why replicate databases, why not partition users?

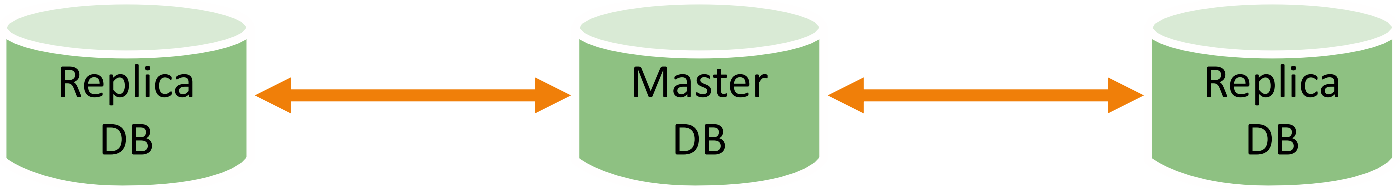
Geographically distributed clusters



One region holds master database,
rest are read-only replicas

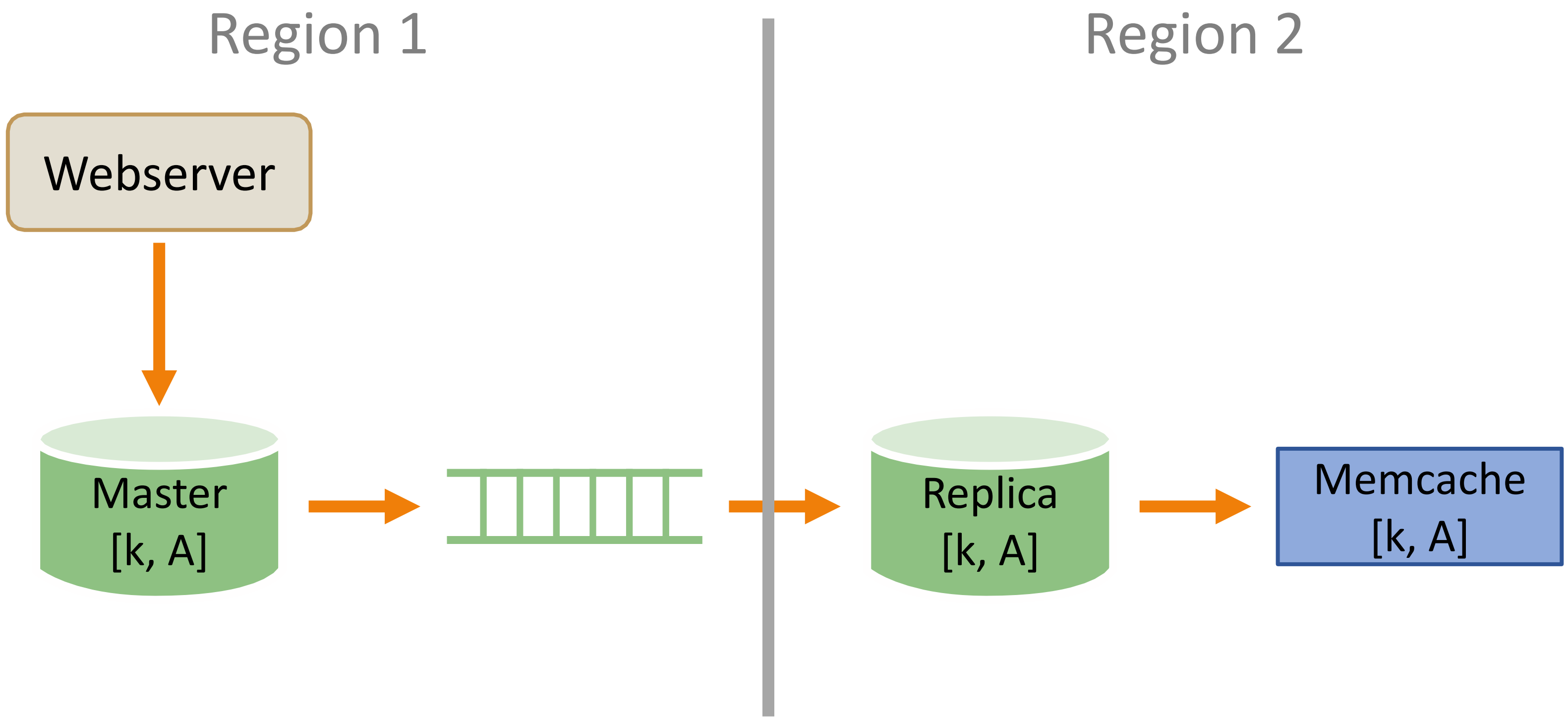
Geographically distributed clusters

- Fast local reads from local Memcache and database replica
- All writes from any region are sent to master
  - Avoids any conflicting writes
  - Why is performance acceptable?
- Master synchronizes replicas asynchronously using database's replication mechanism
  - Replicas lag master, potential for cache inconsistency
  - A replica take over in case master fails



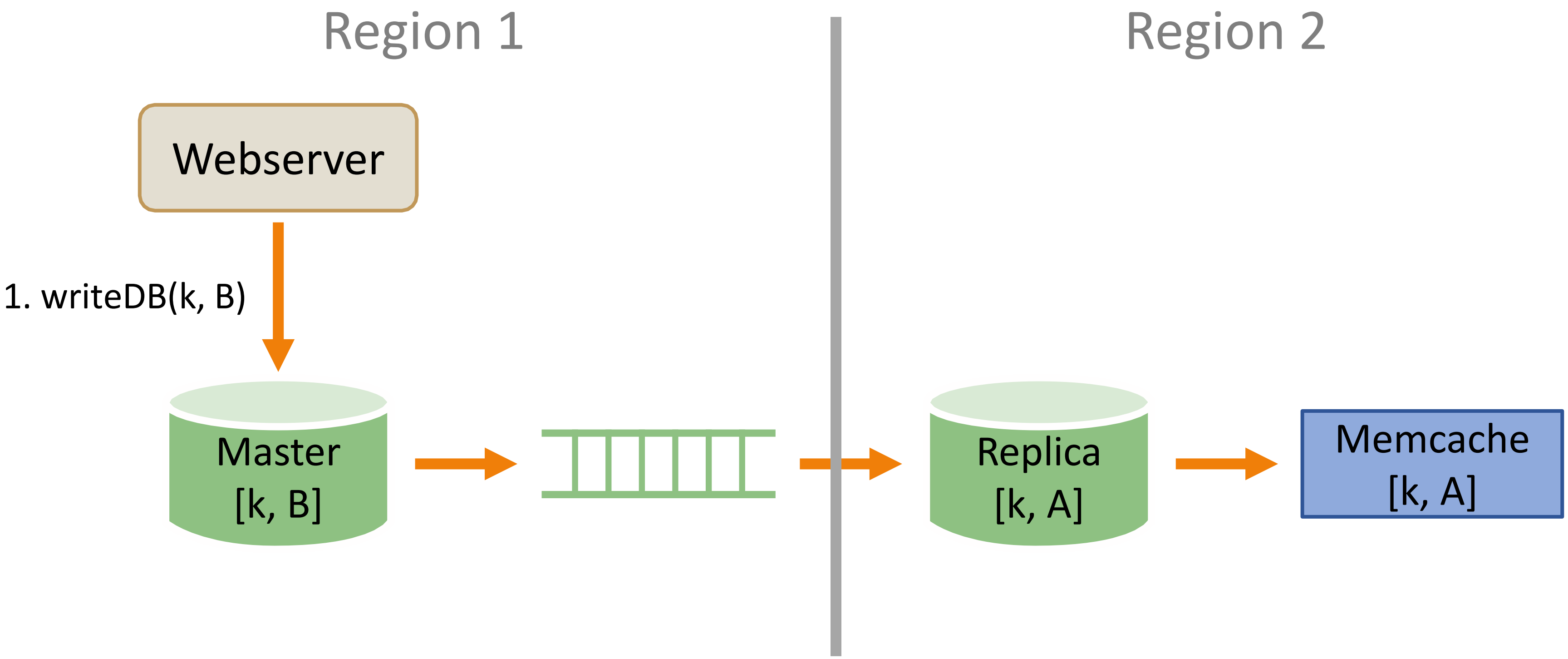
Write at master region

- Ensure consistency of replica and Memcache by reusing invalidation mechanism



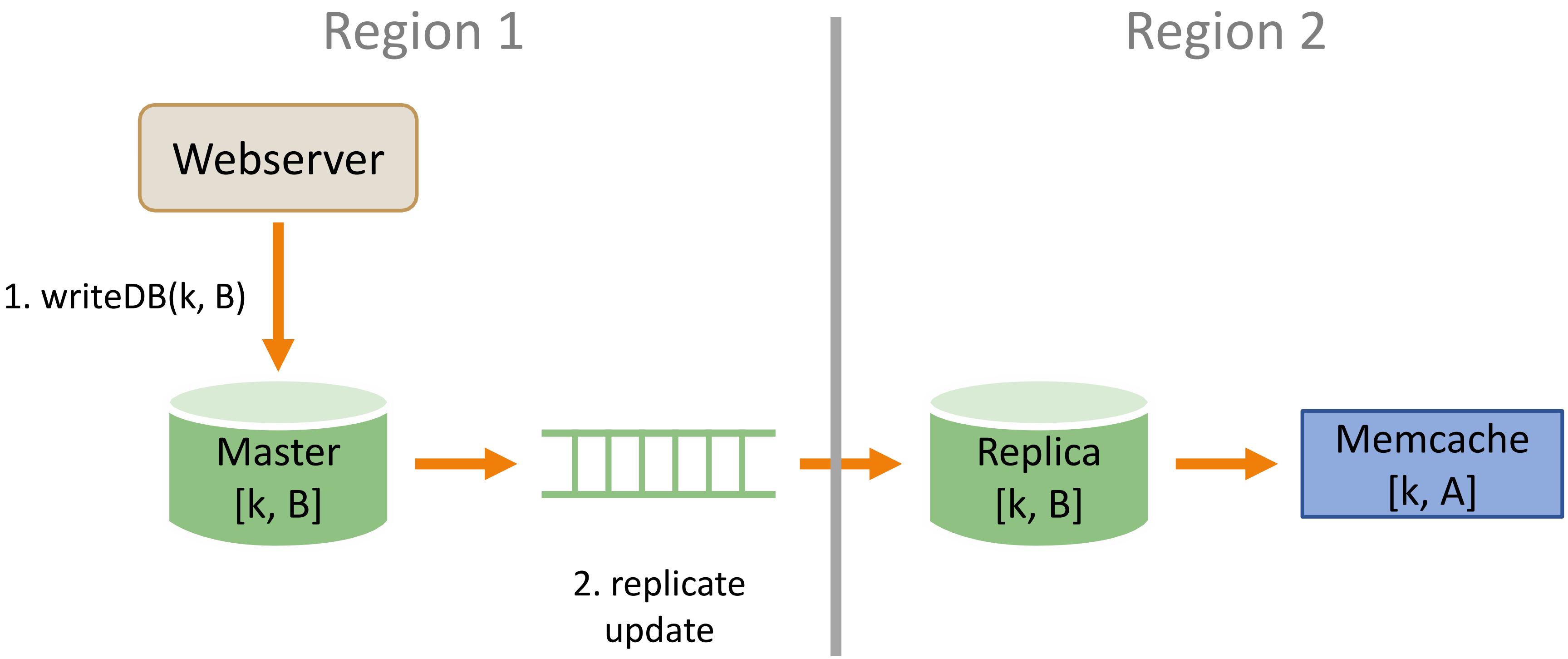
Write at master region

- Ensure consistency of replica and Memcache by reusing invalidation mechanism



Write at master region

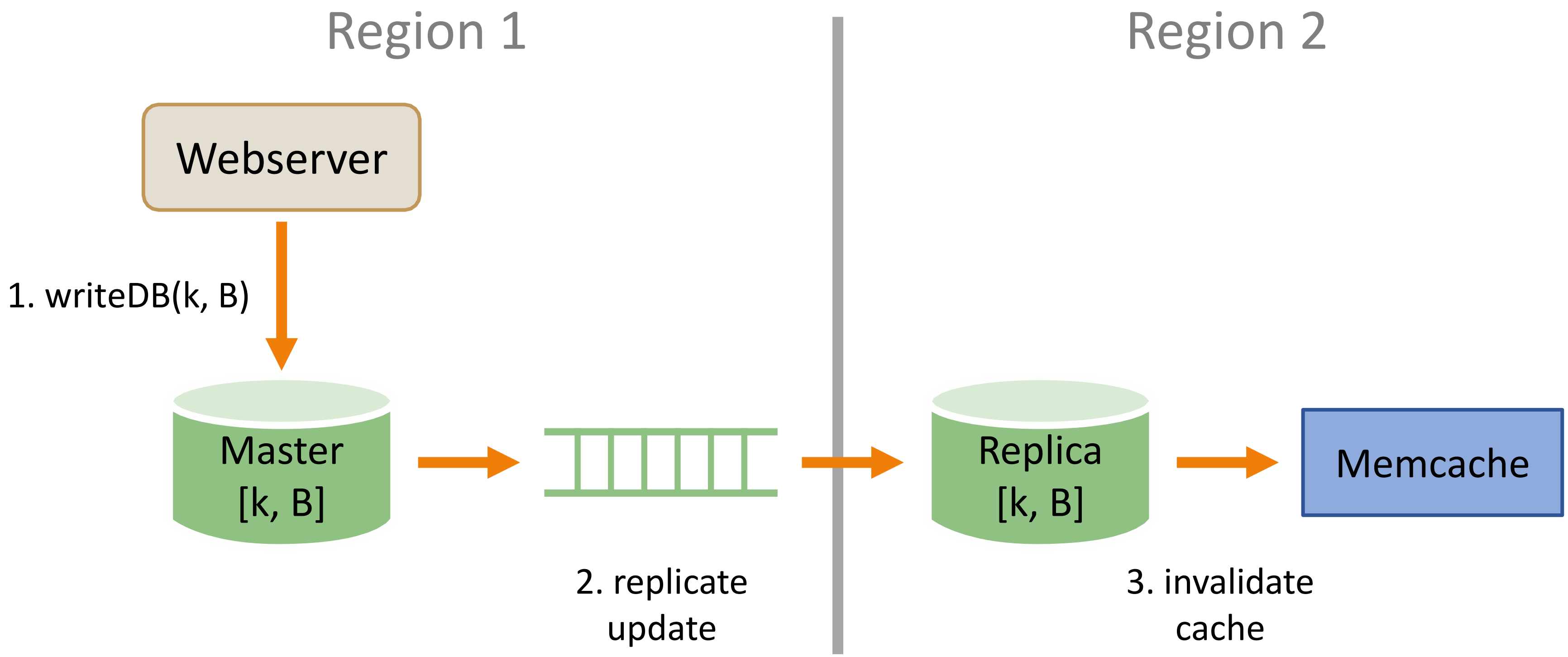
- Ensure consistency of replica and Memcache by reusing invalidation mechanism
  - Replicate update



Write at master region

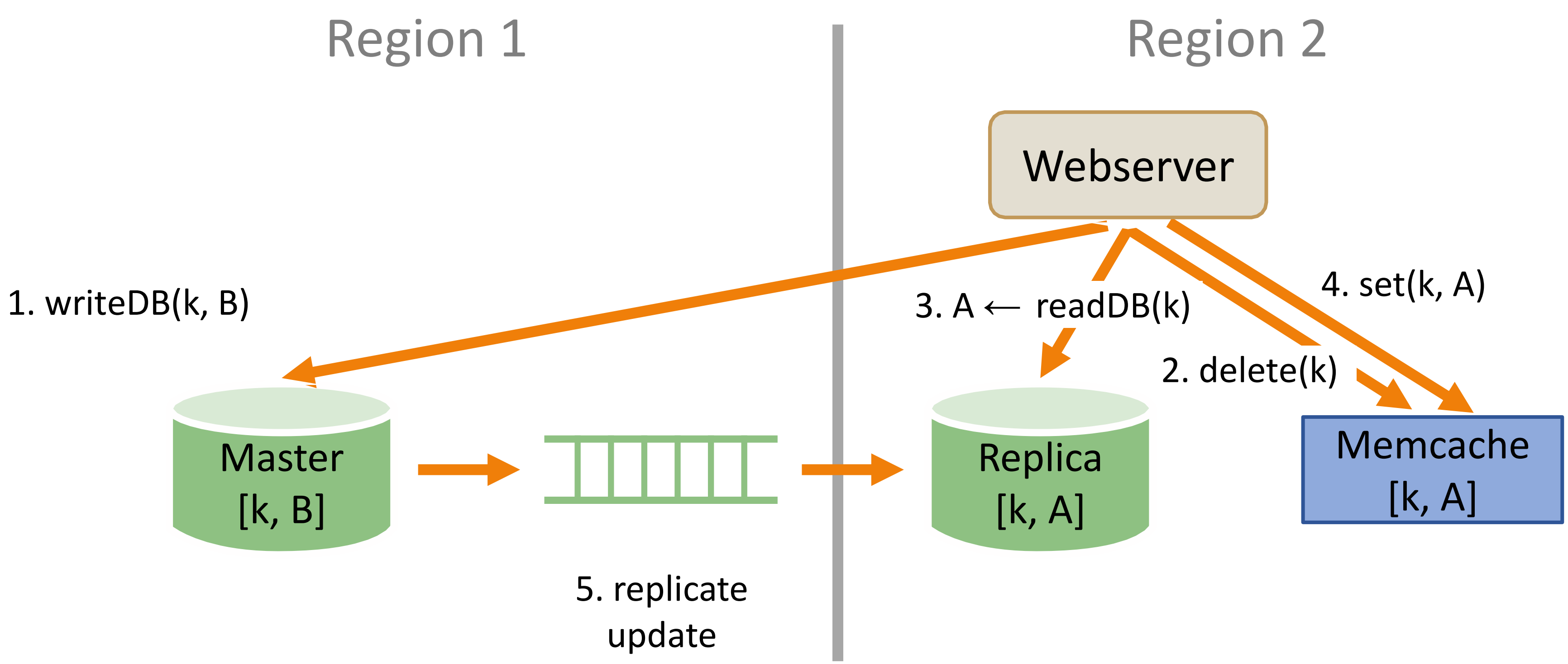
- Ensure consistency of replica and Memcache by reusing invalidation mechanism
  - Replicate update, then
  - Invalidate cache

why replicate and then invalidate?



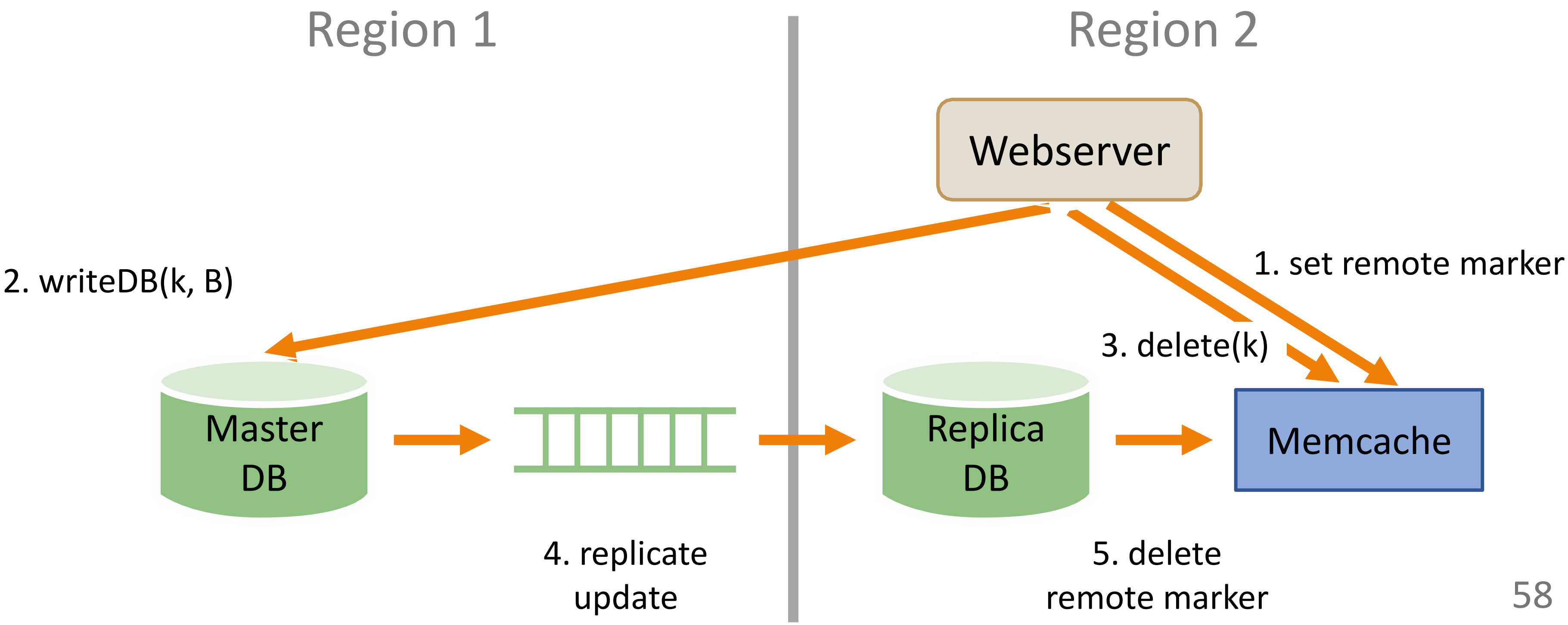
Write at replica region

- After Webserver issues writeDB(k, B) at master (Step 1), it can read and cache stale value from replica (Steps 3, 4) until update is replicated (Step 5)
    - Read-your-write consistency is violated



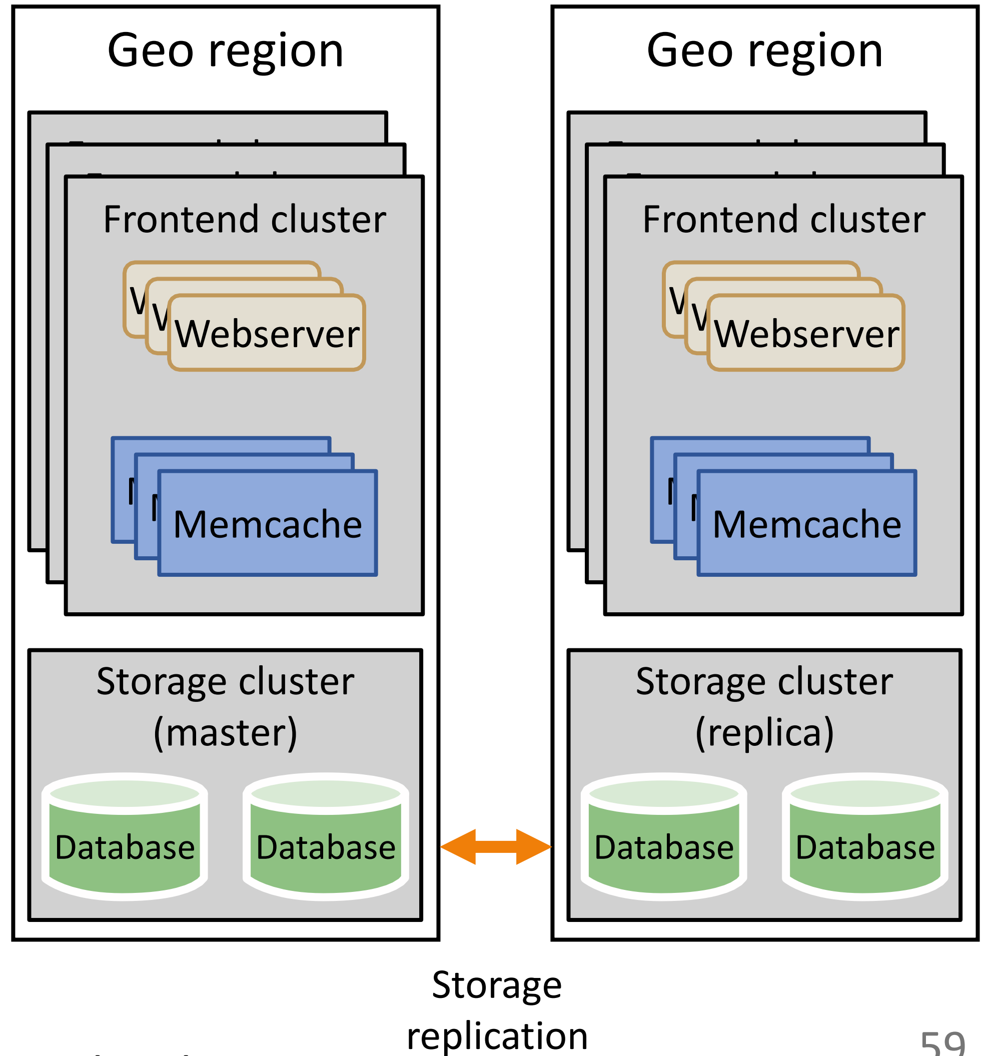
Use remote marker

- Set marker in Memcache indicating replica has stale value
  - If marker is set, read from master, else from replica
  - Ensures read-your-write consistency



Putting it all together

- Start with a single front-end cluster
  - Allows scaling by partitioning data set across caches
- Add multiple front-end clusters in region
  - Allows scaling by replicating caches, reduces communication, hotspots
- Add multiple regions
  - Allows scaling by replicating databases, improves locality



Lessons learned

- Caching reduces latency, vital for surviving high load
- Choose carefully when to shard versus replicate caches
- Provide consistency based on application needs
  - Linearizability will not scale, eventual consistency is okay
- Separate cache and persistent store
  - Allows them to be designed, scaled and operated independently
  - Reusing the standard MySQL database allows reusing standard asynchronous replication mechanisms, replica creation, bulk import, backup, monitoring tools, etc.
- Push complexity into the Webserver, when possible, to simplify design of caching and storage service

Conclusions

- Facebook needed scalable storage for its social graph
- Storage system uses
  - Sharded caches for scaling within a cluster
  - Replicated caches for locality and skew tolerance across clusters
  - Replicated databases for geographic locality across regions
- Design optimized for read-mostly workloads
  - Writes to master database, replicated using primary backup
    - Total order ensures no conflicts, but writes are slower
  - Reads from local database
    - Reads are fast, but may return stale data
    - Idempotent cache invalidations help ensure eventual consistency

Background reading

- *Scaling Memcache at Facebook,* NSDI 2013

Many other practical details

- Regional Memcache pools
- Warming up a new Memcache cluster
- Handling Memcache server failures