Memcache as a Service

Tom Anderson

Goals

Rapid application development ("velocity")

- Speed of adding new features is paramount

Scale

- Billions of users
- Every user on FB all the time

Performance

- Low latency for every user everywhere

Fault tolerance

- Scale implies failures

Consistency model:

- "Best effort eventual consistency"

Facebook's Scaling Problem

- Rapidly increasing user base
  - Small initial user base
  - 2x every 9 months
  - 2013: 1B users globally
- Users read/update many times per day
  - Increasingly intensive app logic per user
  - 2x I/O every 4-6 months
- Infrastructure has to keep pace

Scaling Strategy

Adapt off the shelf components where possible

Fix as you go

- no overarching plan

Rule of thumb: Every order of magnitude requires a rethink

Three-Tier Web Architecture

Cache Server

Front End Server

Cache Server

Front End Server

Cache Server

Front End Server

Client

Front End Server

Storage Server

Storage Server

Storage Server

Storage Server

Storage Server

Three-Tier Web Architecture

Cache Server

Front End Server

Cache Server

Front End Server

Cache Server

Front End Server

Client

Front End Server

Storage Server

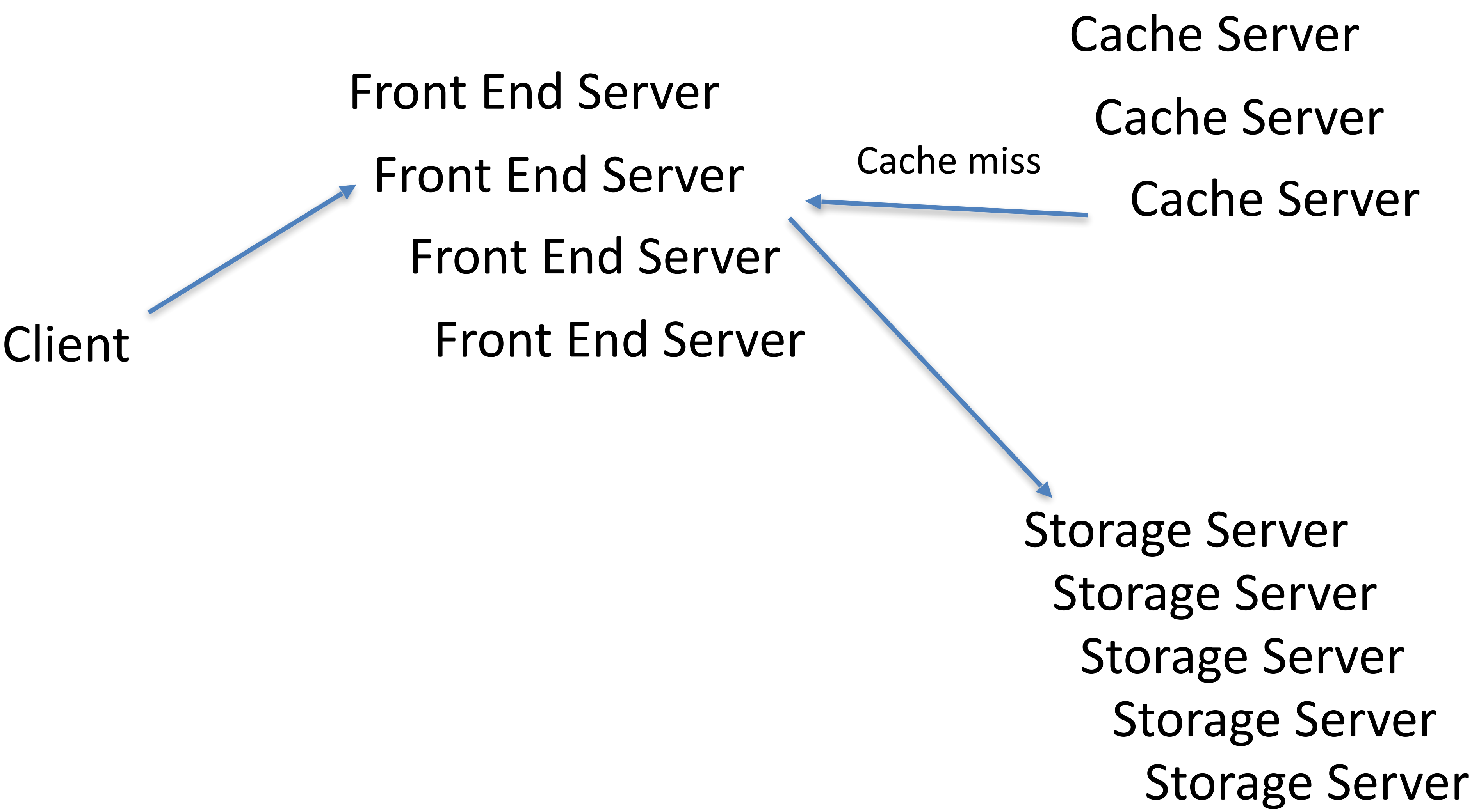
Storage Server

Storage Server

Storage Server

Storage Server

Three-Tier Web Architecture



Facebook Three Layer Architecture

- Application front end
  - Stateless, rapidly changing program logic
  - If app server fails, redirect client to new app server
- Memcache
  - Lookaside key-value cache
  - Keys defined by app logic (can be computed results)
- Fault tolerant storage backend
  - Stateful
  - Careful engineering to provide safety and performance
  - Both SQL and NoSQL

Facebook Workload

- Each user's page is unique
  - draws on events posted by other users
- Users not in cliques
  - For the most part
- User popularity is zipf
  - Some user posts affect very large #'s of other pages
  - Most affect a much smaller number

Workload

- Many small lookups
- Many dependencies
- App logic: many diffuse, chained reads
  - latency of each read is crucial
- Much smaller update rate
  - still large in absolute terms

Scaling

- A few servers
- Many servers
- An entire data center
- Many data centers

Each step 10-100x previous one

Facebook

- Scale by hashing to partitioned servers
- Scale by caching
- Scale by replicating popular keys
- Scale by replicating clusters
- Scale by replicating data centers

Scale By Consistent Hashing

Hash users to front end web servers

Hash keys to memcache servers

Hash files to SQL servers

Result of consistent hashing is all to all communication pattern

– Each web server pulls data from all memcache servers and all storage servers

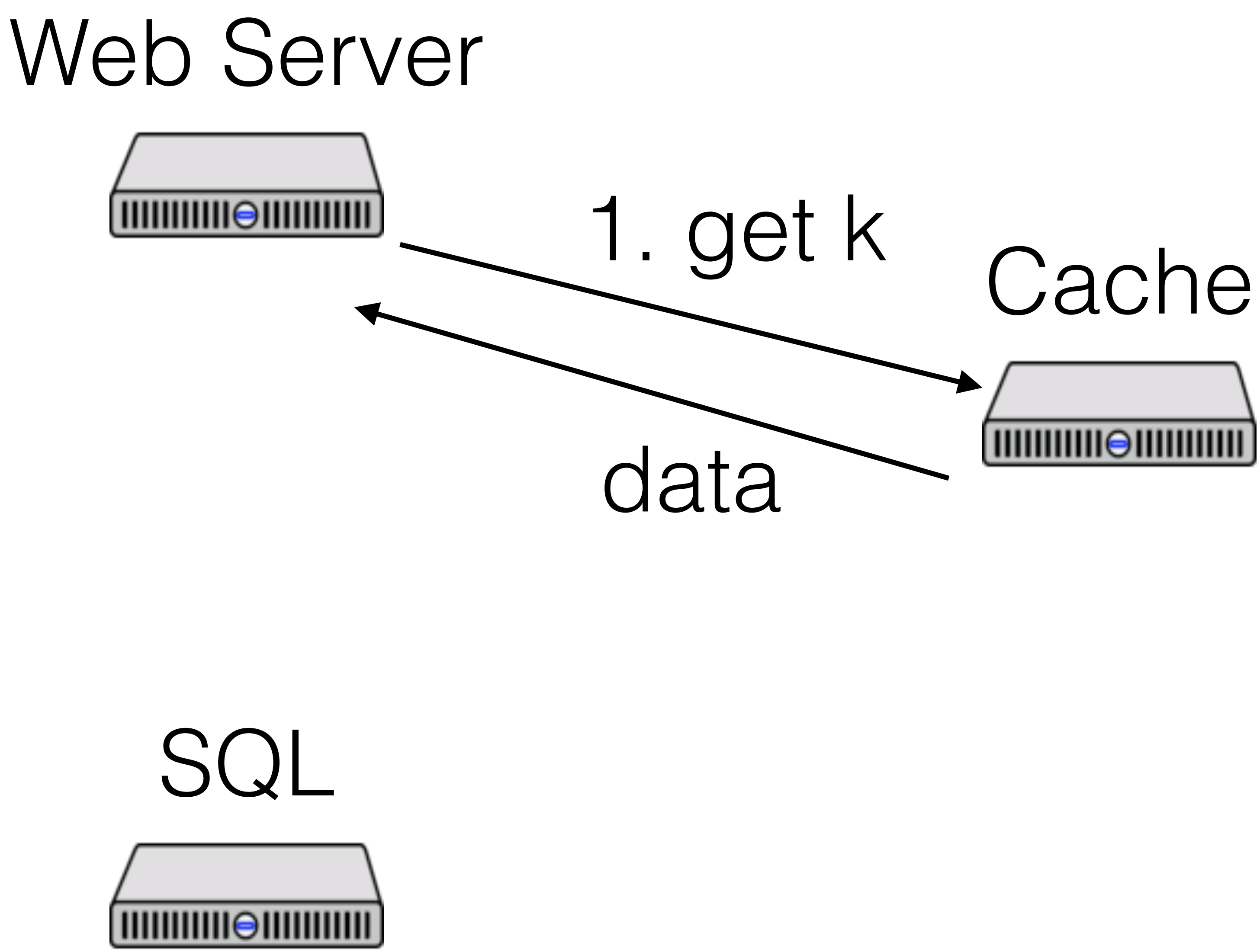
Scale By Caching: Memcache

Sharded in-memory key-value cache

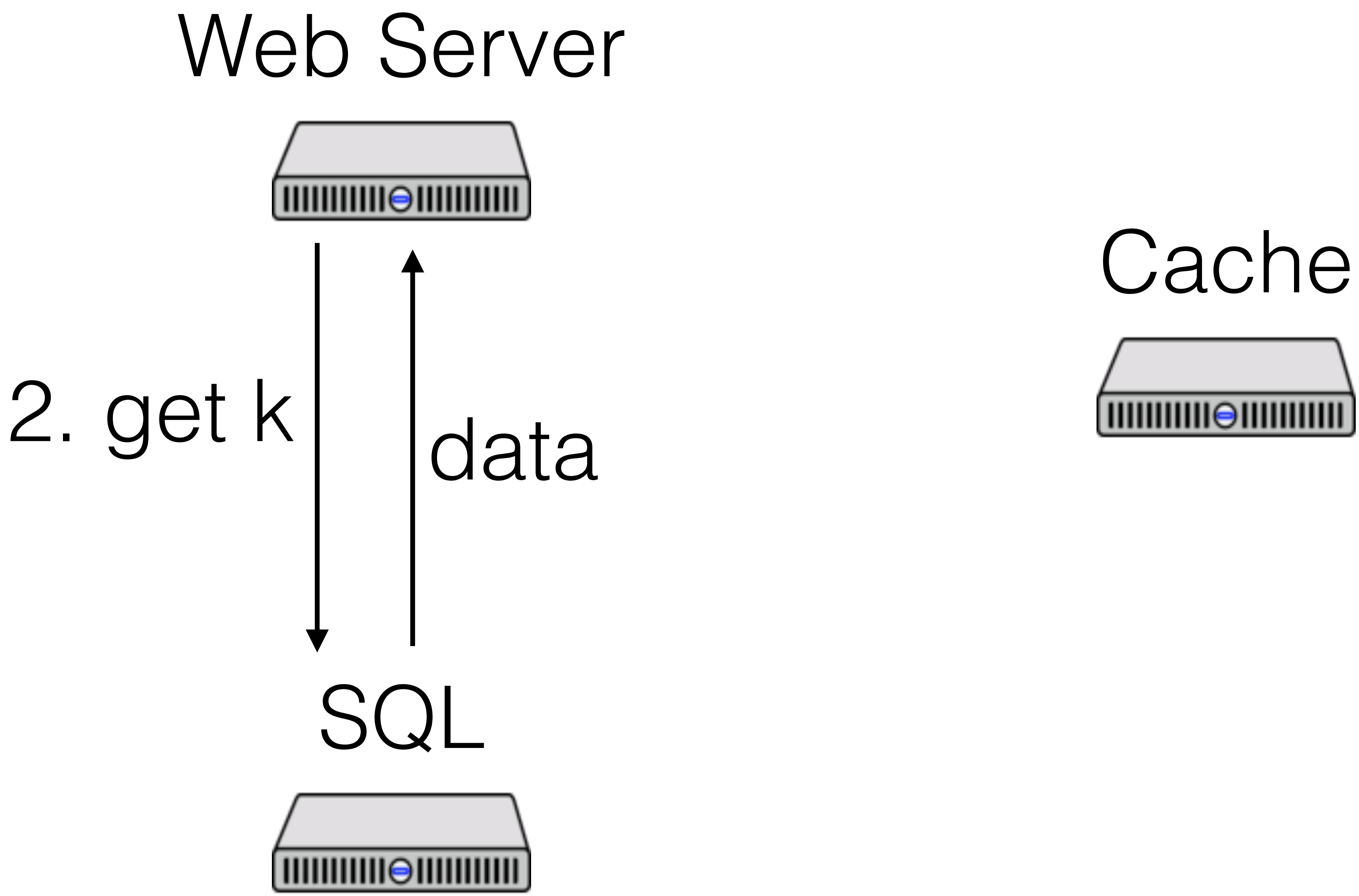
- Key, values assigned by application code
- Values can be data, result of computation
- Independent of backend storage architecture (SQL, noSQL) or format
- Design for high volume, low latency

Lookaside architecture

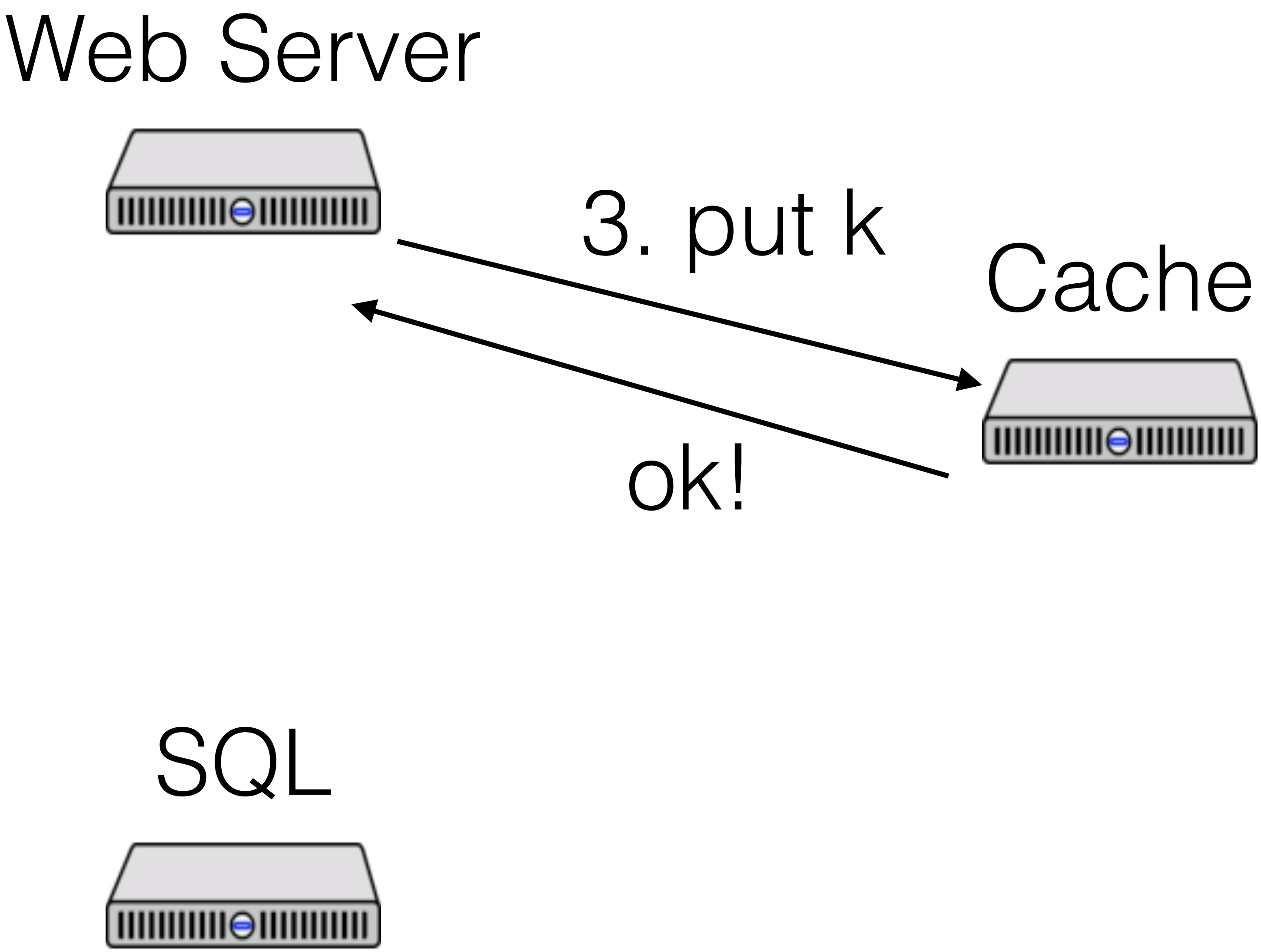
Lookaside Read



Lookaside Read



Lookaside Read



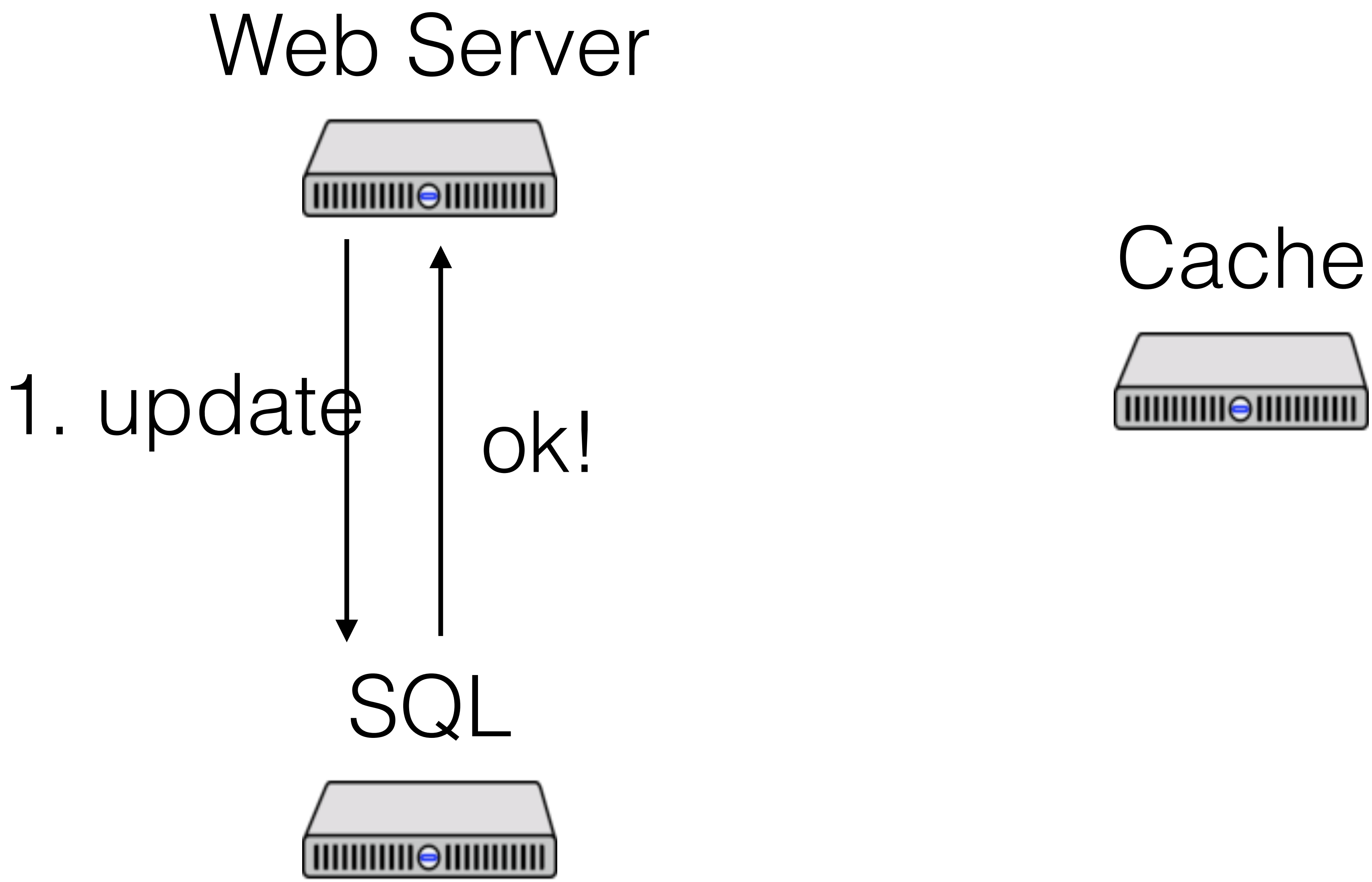
Lookaside Operation (Read)

- Webserver needs key value
- Webserver requests from memcache
- Memcache: If in cache, return it
- If not in cache:
  - Return error
  - Webserver gets data from storage server
  - Possibly an SQL query or complex computation
  - Webserver stores result back into memcache

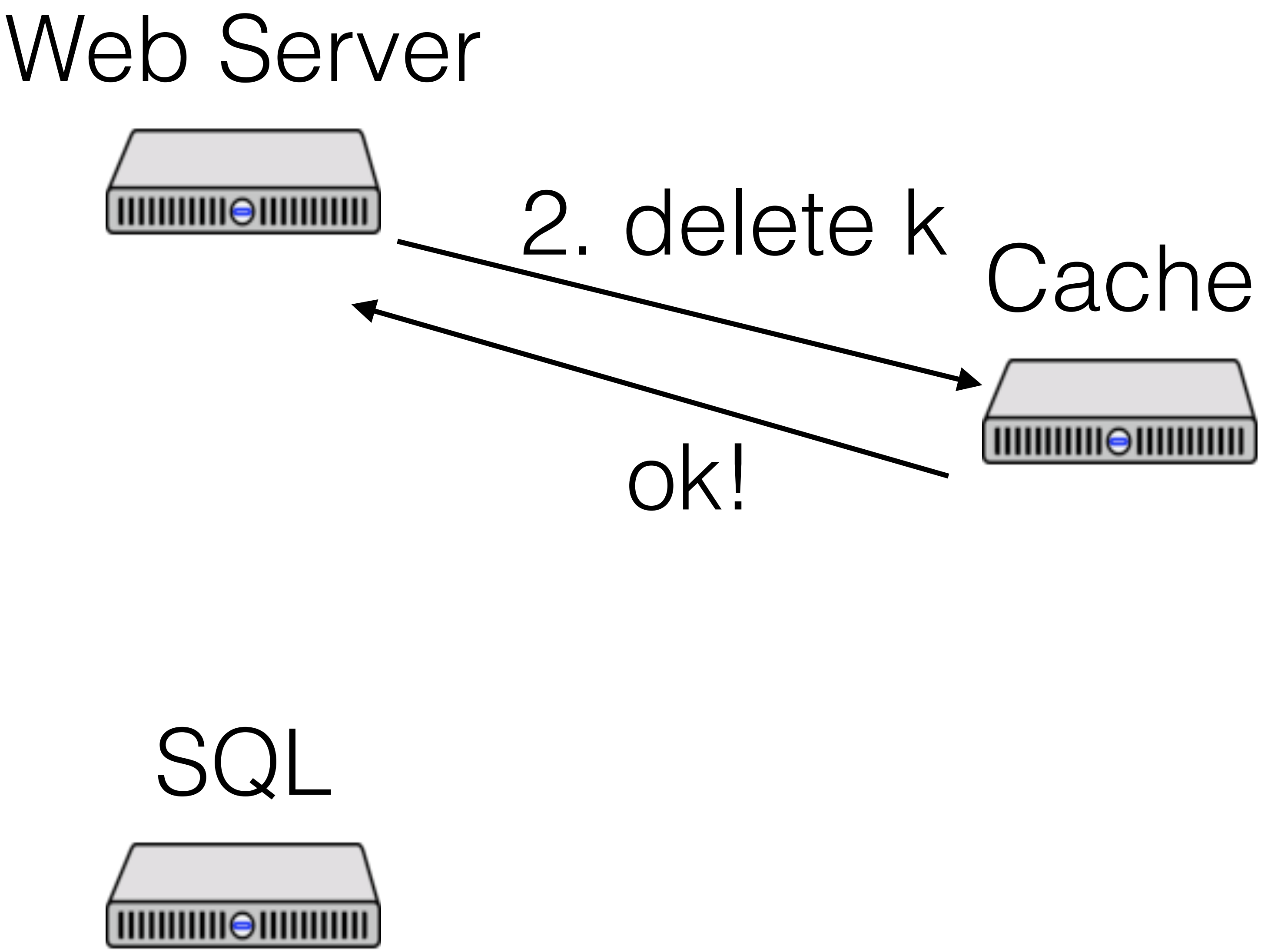
Question

What if swarm of users read same key at the same time?

Lookaside Write



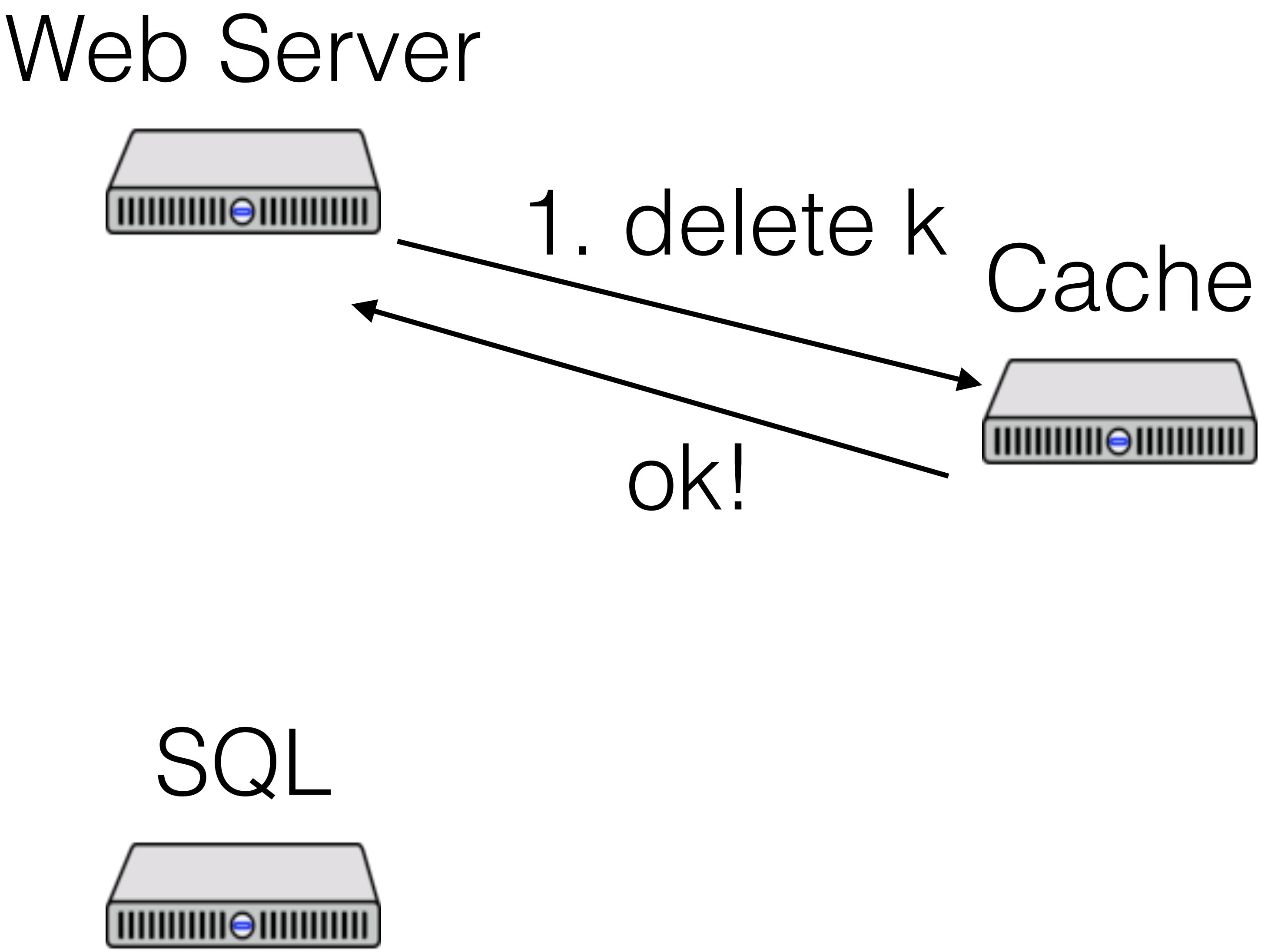
Lookaside Write



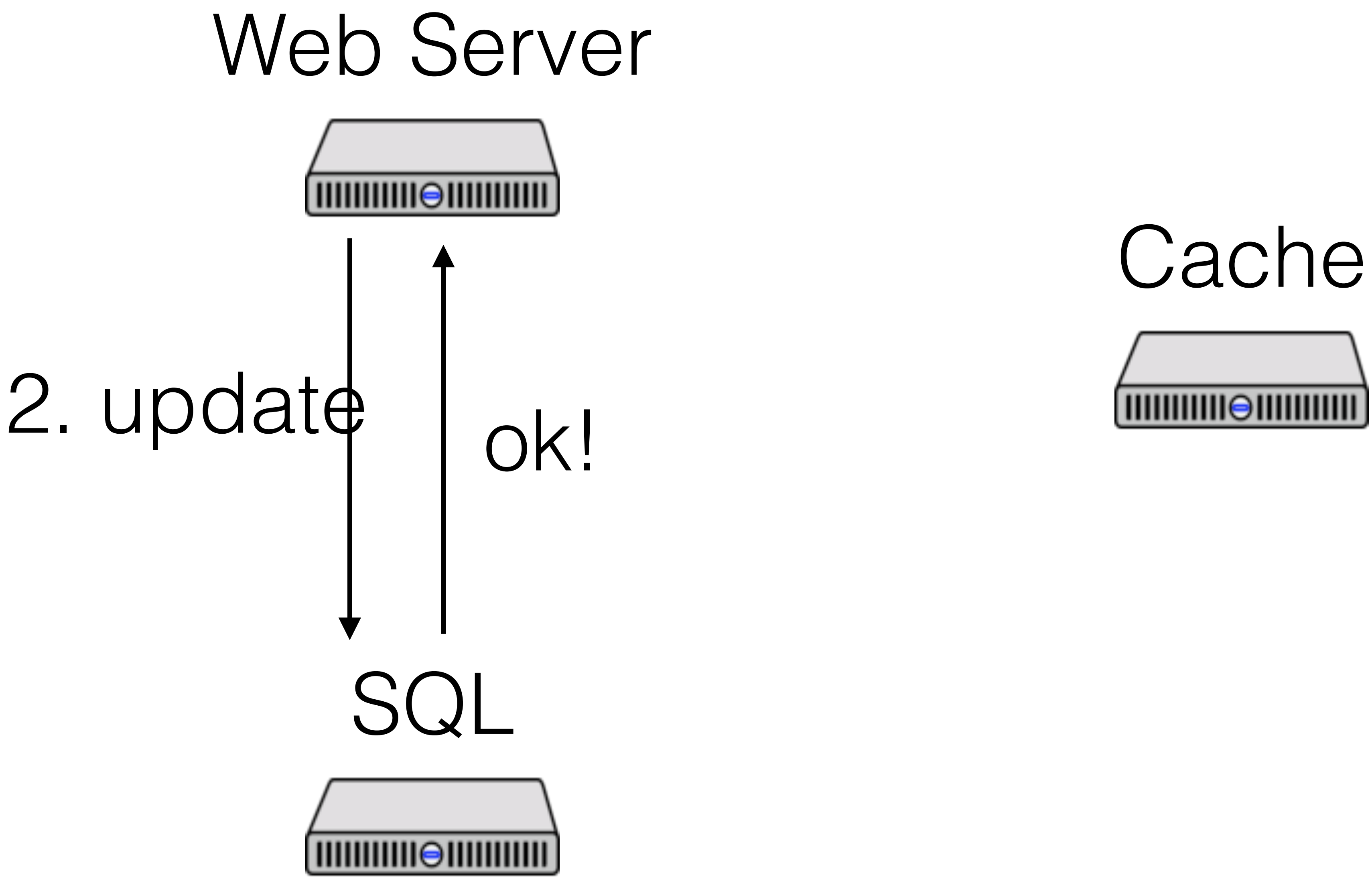
Lookaside Operation (Write)

- Webserver changes a value that would invalidate a memcache entry
  - Could be an update to a key
  - Could be an update to a table
  - Could be an update to a value used to derive some key value
- Client puts new data on storage server
- Client invalidates entry in memcache

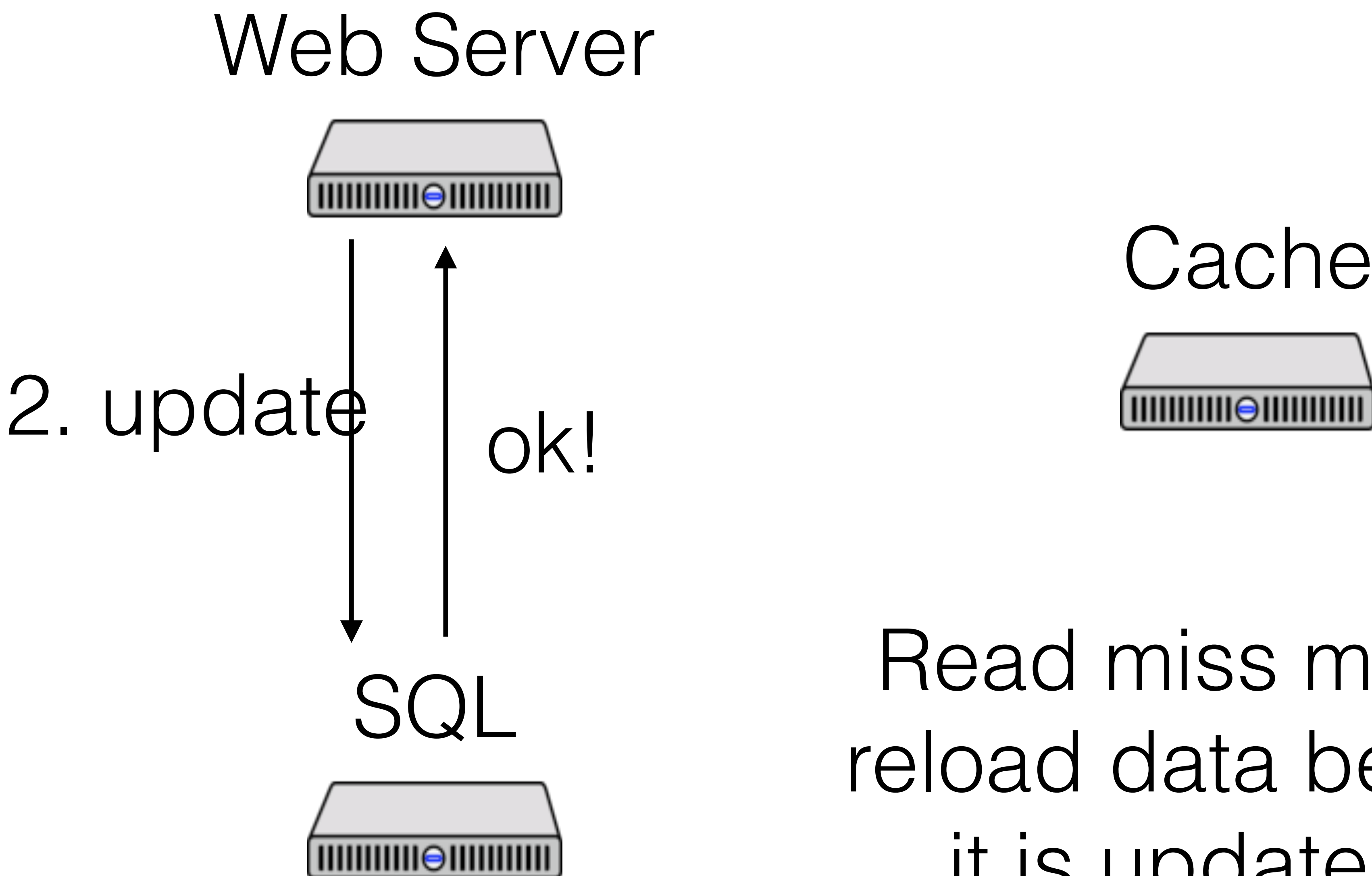
Why Not Delete then Update?



Why Not Delete then Update?



Why Not Delete then Update?



Read miss might reload data before it is updated.

Memcache Consistency

Is memcache linearizable?

Example

Reader

Read cache

If missing,

  Fetch from database

  Store back to cache

Writer

Change database

Delete cache entry

Interleave any # of readers/writers

Example

- Read cache
- Read database

- change database
- Delete entry

- Store back to cache

Memcache Consistency

Is the lookaside protocol eventually consistent?

Lookaside With Leases

Goals:
- Reduce (not eliminate) per-key inconsistencies
- Reduce cache miss swarms

On a read miss:
- leave a marker in the cache (fetch in progress)
- return timestamp
- check timestamp when filling the cache
- if changed in meantime: don't overwrite

If another thread read misses:
- find marker and wait for update (retry later)

Question

What if front end crashes while holding read lease?

Would any other front end be able to read the data?

Question

Is FB’s version of lookaside with leases linearizable?

Example: Cache data with 1 replica

Reader1

(data cached)

Writer

Change database

Read replica1

(old value)

CRASH!

(before Delete cache)

Read replica1

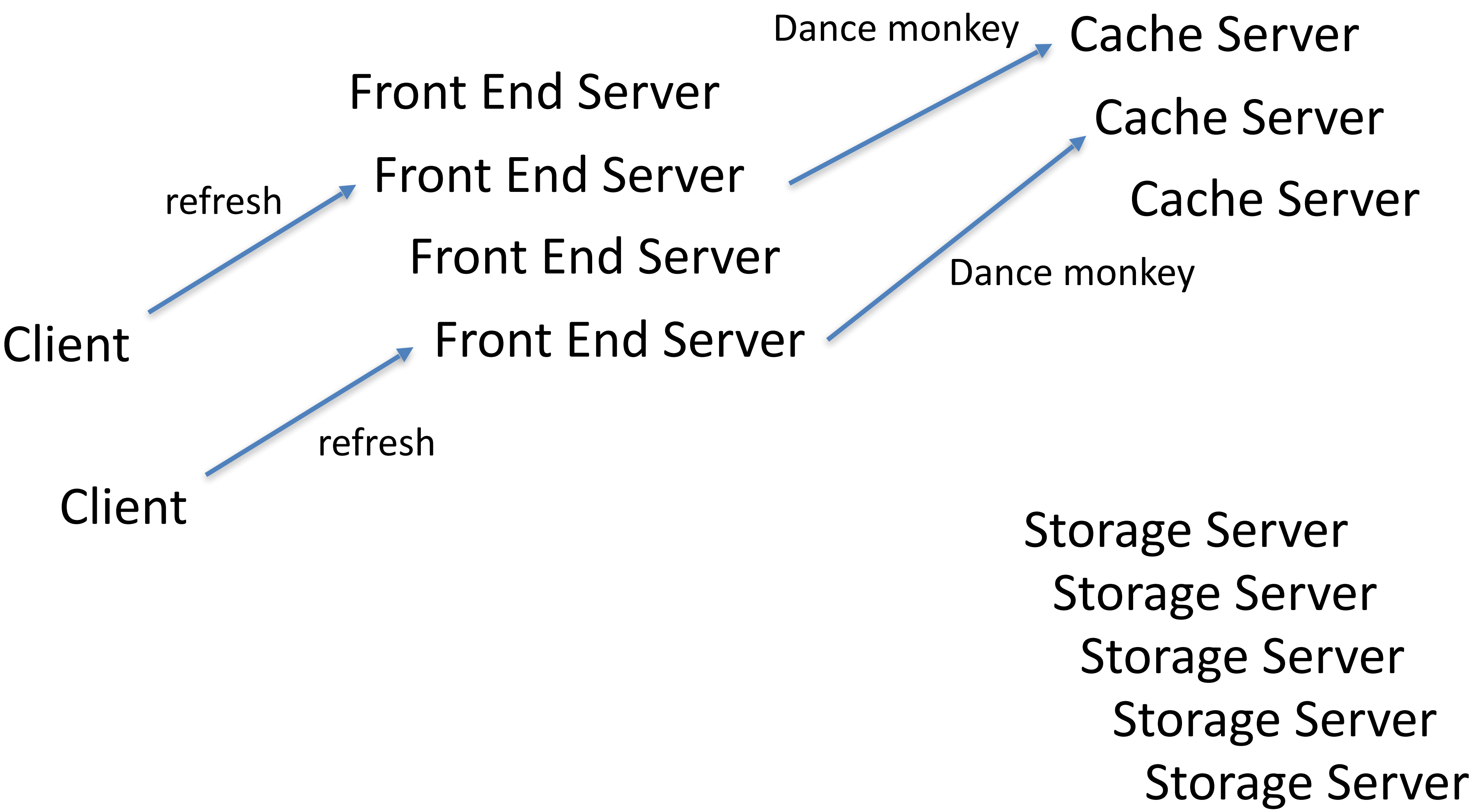
(old value)

Question

Is FB’s version of lookaside with leases linearizable?

Note FB allows popular data to be found in multiple cache servers

FB Replicates Popular Data Across Caches



Example: Cache data with 2 replicas

Reader1	Reader2	Writer
(data cached)	(not cached)	
		Change database
Read replica1		
(old value)	Read replica2	
	Miss	CRASH!
	Fetch from db	(before Delete cache)
Read replica1	Write back to replica 2	
(old value)	(new value)	

Latency Optimizations

Concurrent lookups

- Issue many lookups concurrently
- Prioritize those that have chained dependencies

Batching

- Batch multiple requests (e.g., for different end users) to the same memcache server

Incast control:

- Limit concurrency to avoid collisions among RPC responses

More Optimizations

Return stale data to web server if lease is held

- No guarantee that concurrent requests returning stale data will be consistent with each other

Partitioned memory pools

- Infrequently accessed, expensive to recompute
- Frequently accessed, cheap to recompute
- If mixed, frequent accesses will evict all others

Replicate keys if access rate is too high

Gutter Cache

When a memcache server fails, flood of requests to fetch data from storage layer

- Slower for users needing any key on failed server
- Slower for users due to storage server contention

Solution: backup (gutter) cache

- Time-to-live invalidation (ok if clients disagree as to whether memcache server is still alive)
- TTL is eventually consistent

Scaling Within a Cluster

What happens as we increase the number of memcache servers to handle more load?

- Recall: All to all communication pattern
- Less data between any pair of nodes: less batching
- Need even more replication of popular keys
- More failures: need bigger gutter cache
- …

Multi-Cluster Scaling

Multiple independent clusters within data center

- Each with front-ends, memcache servers
- Data replicated in the caches in each partition
- Shared storage backend

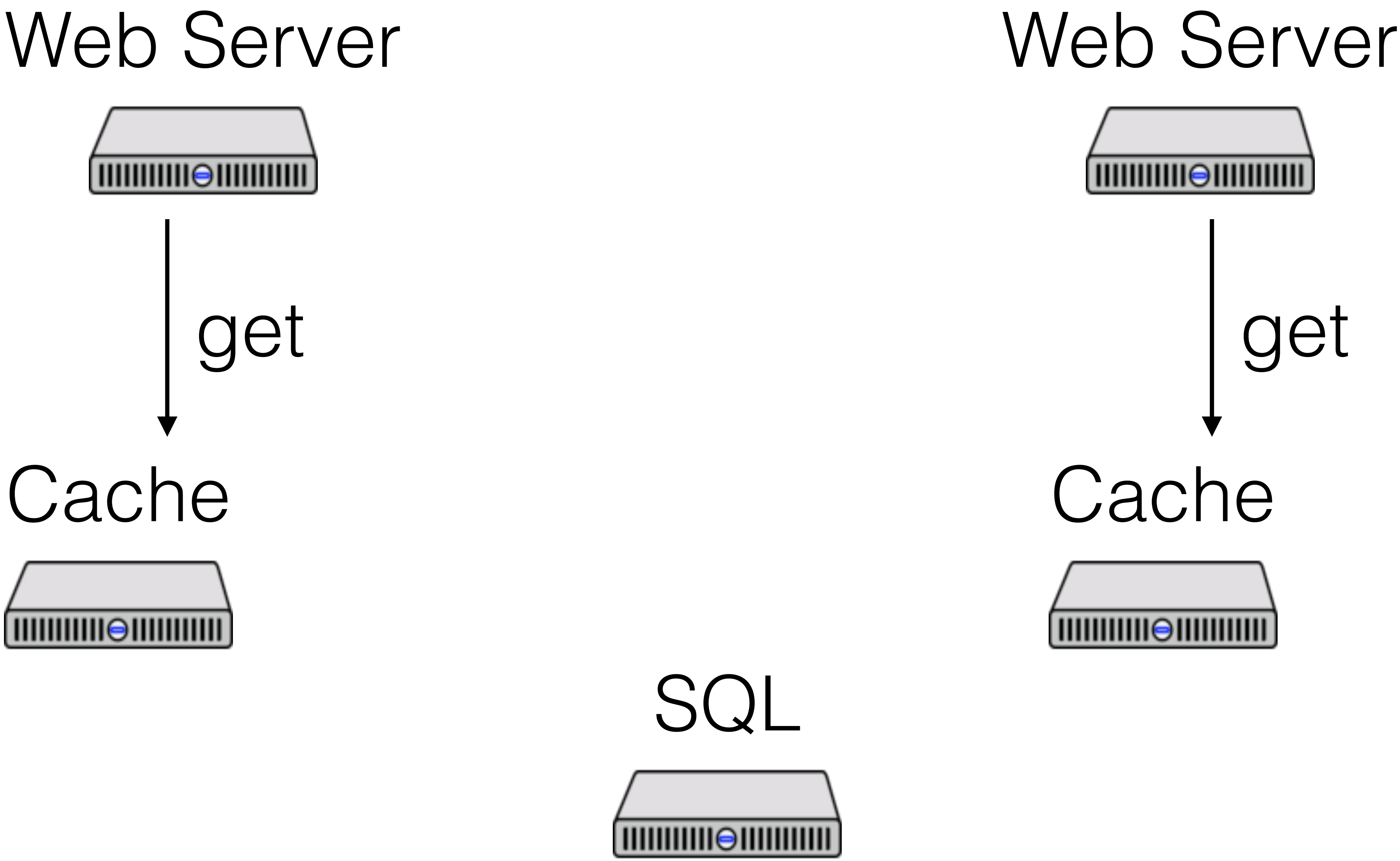
Data is replicated in each cluster (inefficient?)

- need to invalidate every cluster on every update

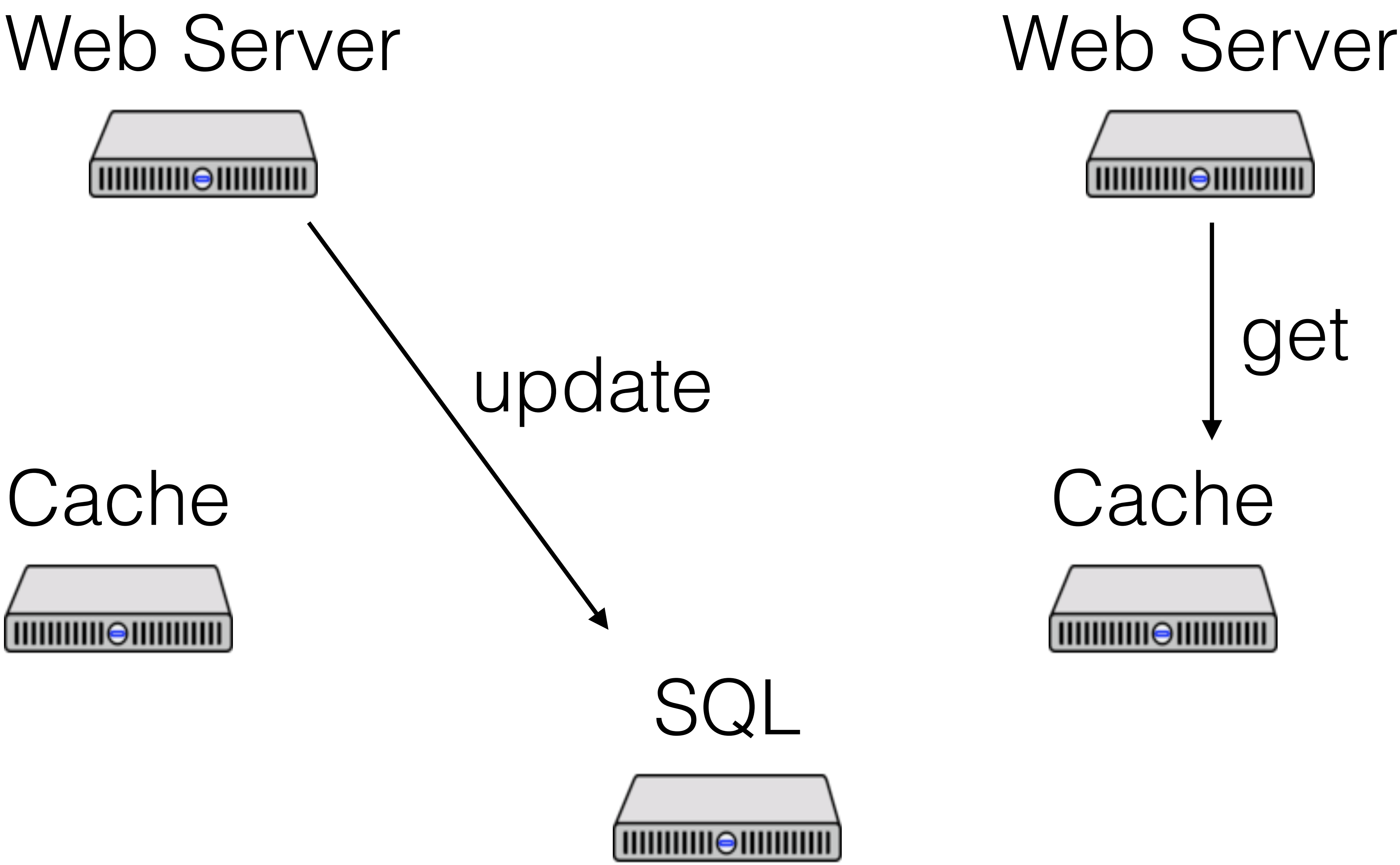
Instead:

- invalidate local cluster on update (read my writes)
- background invalidate driven off of database update log
- temporary inconsistency!

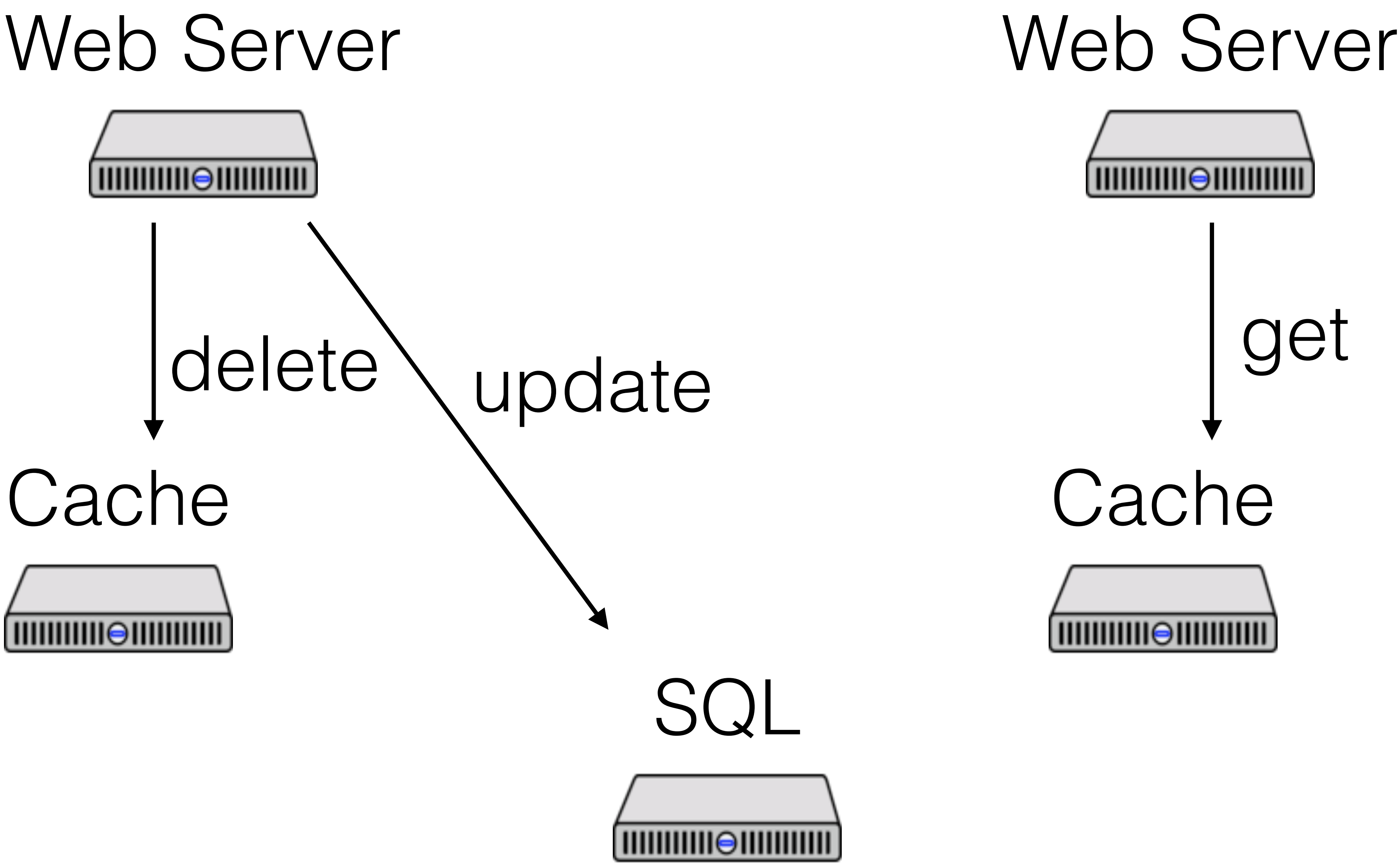
Multi-Cluster Scaling



Multi-Cluster Scaling



Multi-Cluster Scaling



mcsqueal

- Web servers talk to local memcache. On update:
  - Acquire local lease
  - Tell storage layer which keys to invalidate
  - Invalidate local memcache
- Storage layer sends invalidations to other clusters
  - Scan database log for updates/invalidations
  - Batch invalidations to each cluster (mcrouter)
  - Forward/batch invalidations to remote memcache servers

Per-Cluster vs. Multi-Cluster

Per-cluster memcache servers

- Frequently accessed data
- Inexpensive to compute data
- Lower latency, less efficient use of memory

Shared multi-cluster memcache servers

- infrequently accessed
- hard to compute data
- higher latency, more memory efficient

Cold Start Consistency

During new cluster startup:

- Many cache misses!
- Lots of extra load on SQL servers

Instead of going to SQL server on cache miss:

- Webserver gets data from warm memcache cluster
- Puts data into local cluster
- Subsequent requests hit in local cluster

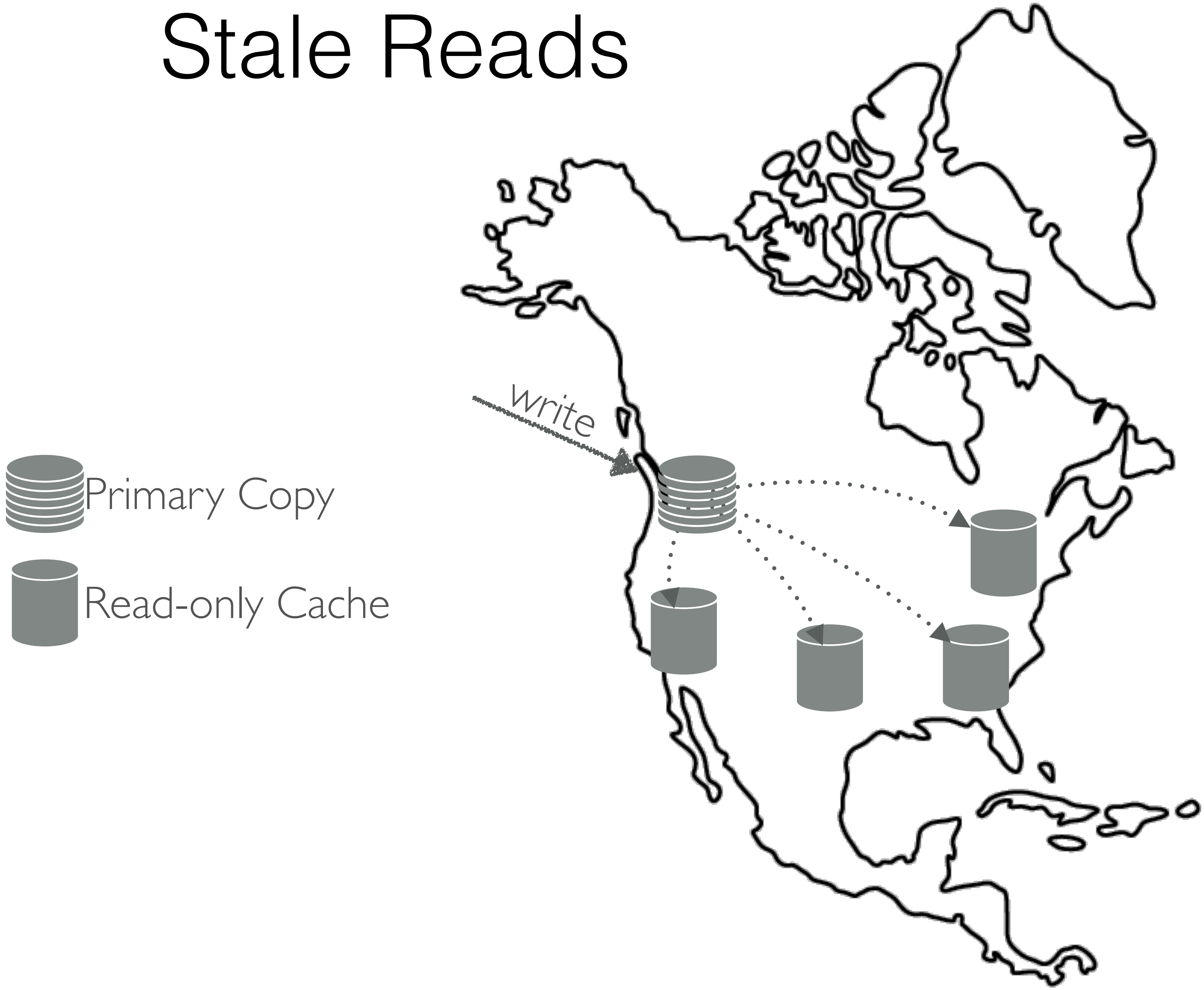
Multi-Region Scaling

Storage layer consistency

- Storage at one data center designated as primary
- All updates applied at primary
- Updates propagated in background to other data centers

What could go wrong?

Stale Reads



Multi-Region Consistency

To perform an update to key:

- put marker into local region
- Send write to primary region
- Delete local copy

On a cache miss:

- Check if local marker
- If so, fetch data from primary region
- Fill local copy

FB: Data Centers without Data

Tradeoff in increasing number of data centers

- Lower latency when data near clients
- More consistency overhead
- More opportunity for inconsistency

Mini-data centers

- Front end web servers
- Memcache servers
- No backend storage: remote access for cache misses

Linearizability?

Is linearizability possible with a memcache layer?

- Needs help from storage layer
- Every cached copy removed before write

What about snapshot reads?

- Needs help from storage layer
- Every copy has version timestamp range
- Multikey query valid if ranges overlap