facebook

Scaling Memcache at Facebook

Presenter: Rajesh Nishtala (rajesh.nishtala@fb.com)

Co-authors: 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

facebook

Infrastructure Requirements for Facebook

1. Near real-time communication
2. Aggregate content on-the-fly from multiple sources
3. Be able to access and update very popular shared content (skewed)
4. Scale to process millions of user requests per second

Design Requirements

100k QPS

Support a very heavy read load

- Over 1 billion reads / second
- Insulate backend services from high read rates

Geographically Distributed

Support a constantly evolving product

- System must be flexible enough to support a variety of use cases
- Support rapid deployment of new features

Persistence handled outside the system

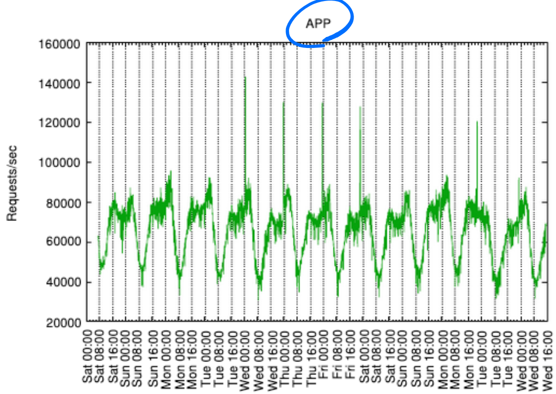
- Support mechanisms to refill after updates

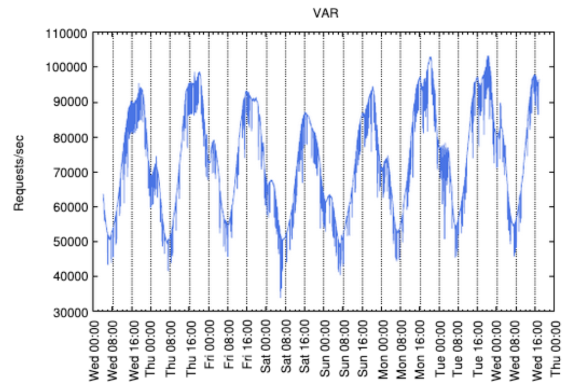
50GB × 100 = 5TB

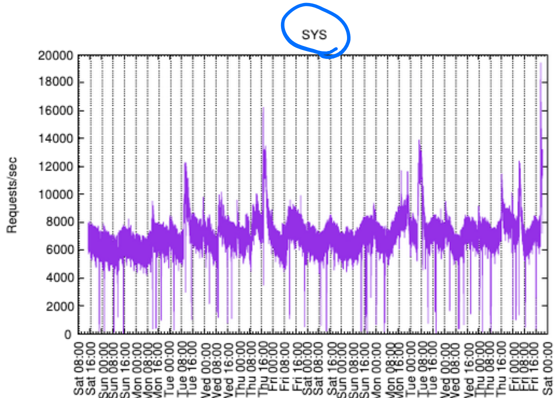
Table 1: Memcached pools sampled (in one cluster). These pools do not match their UNIX namesakes, but are used for illustrative purposes here instead of their internal names.

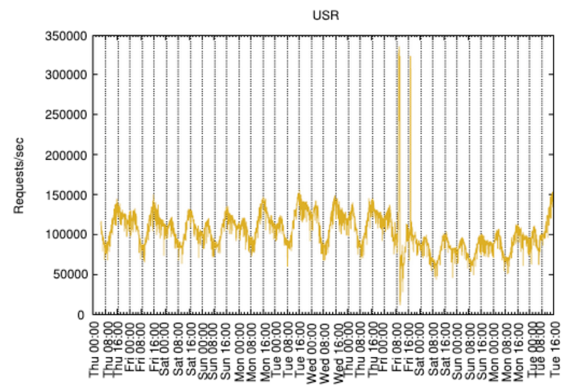
Pool	Size	Description
USR	few	user-account status information
APP	dozens	object metadata of one application
ETC	hundreds	nonspecific, general-purpose
VAR	dozens	server-side browser information
SYS	few	system data on service location

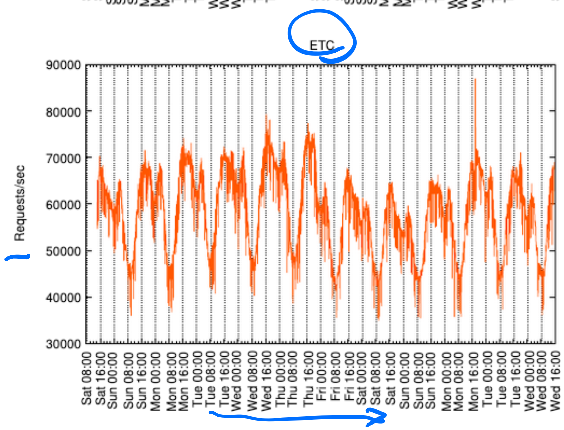
*: Workload analysis of a large-scale key-value store, Sigmetrics 2012

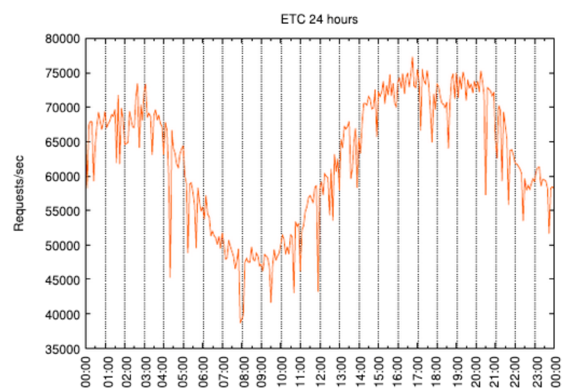

APP
Requests/sec


VAR
Requests/sec


SYS
Requests/sec

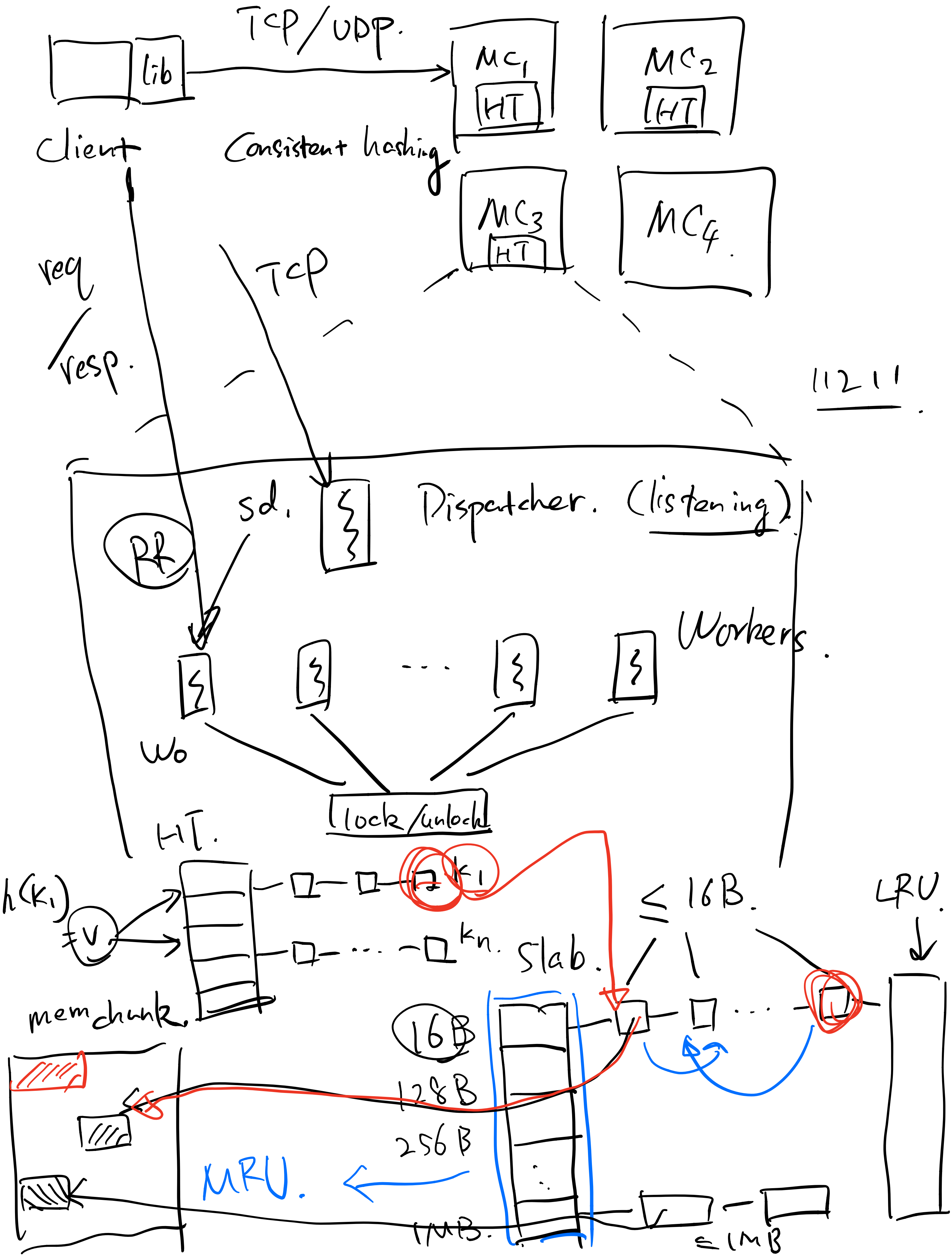

USR
Requests/sec


ETC
Requests/sec


ETC 24 hours
Requests/sec

memcached

- Basic building block for a distributed key-value store for Facebook
  - Trillions of items
  - Billions of requests / second
- Network attached in-memory hash table
  - Supports LRU based eviction

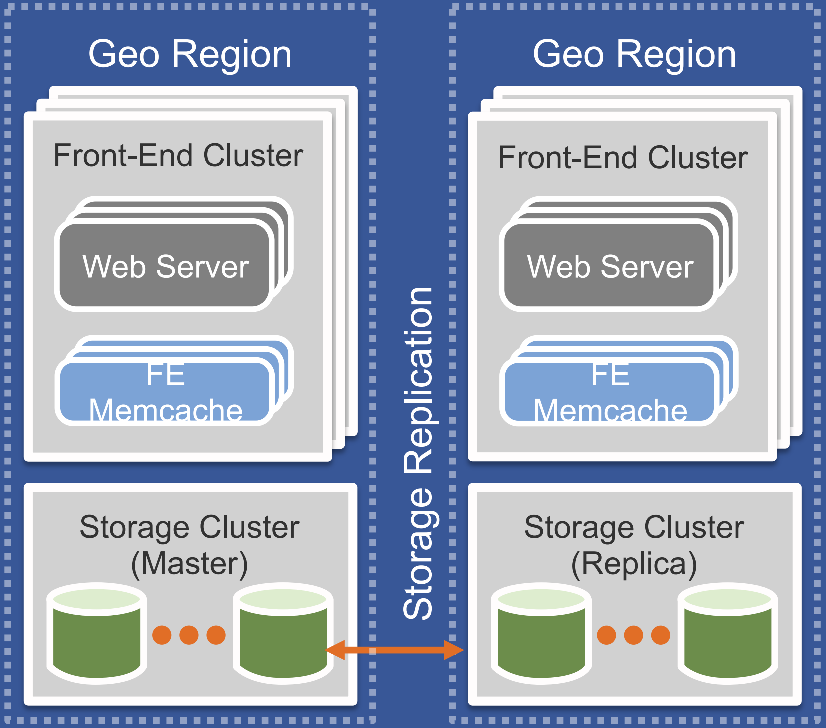


TCP/UDP.
lib
client
Consistent hashing
MC1
HT
MC2
HT
MC3
HT
MC4.
req / resp.
TCP
11211.
sd.
RR
Dispatcher. (listening)
Workers.
W0
lock/unlock
HT.
h(k1)
%V
k1
kn.
Slab.
≤ 16B.
LRU.
memchunk.
16B
128B
256B
MRU.
1MB.
≤ 1MB

Roadmap

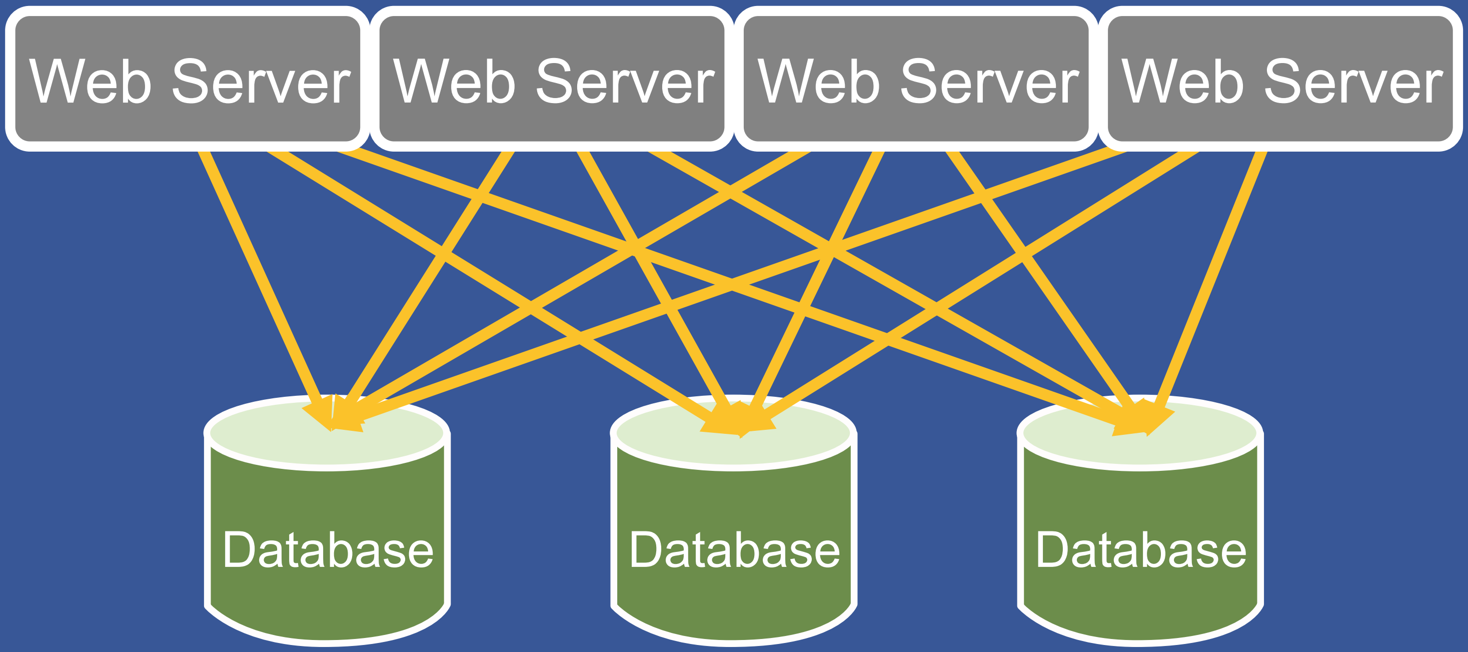
0. Pre-memcached Era

1. Single front-end cluster
   - Read heavy workload
   - Wide fanout
   - Handling failures
2. Multiple front-end clusters
   - Controlling data replication
   - Data consistency
3. Multiple Regions
   - Data consistency



Pre-memcache

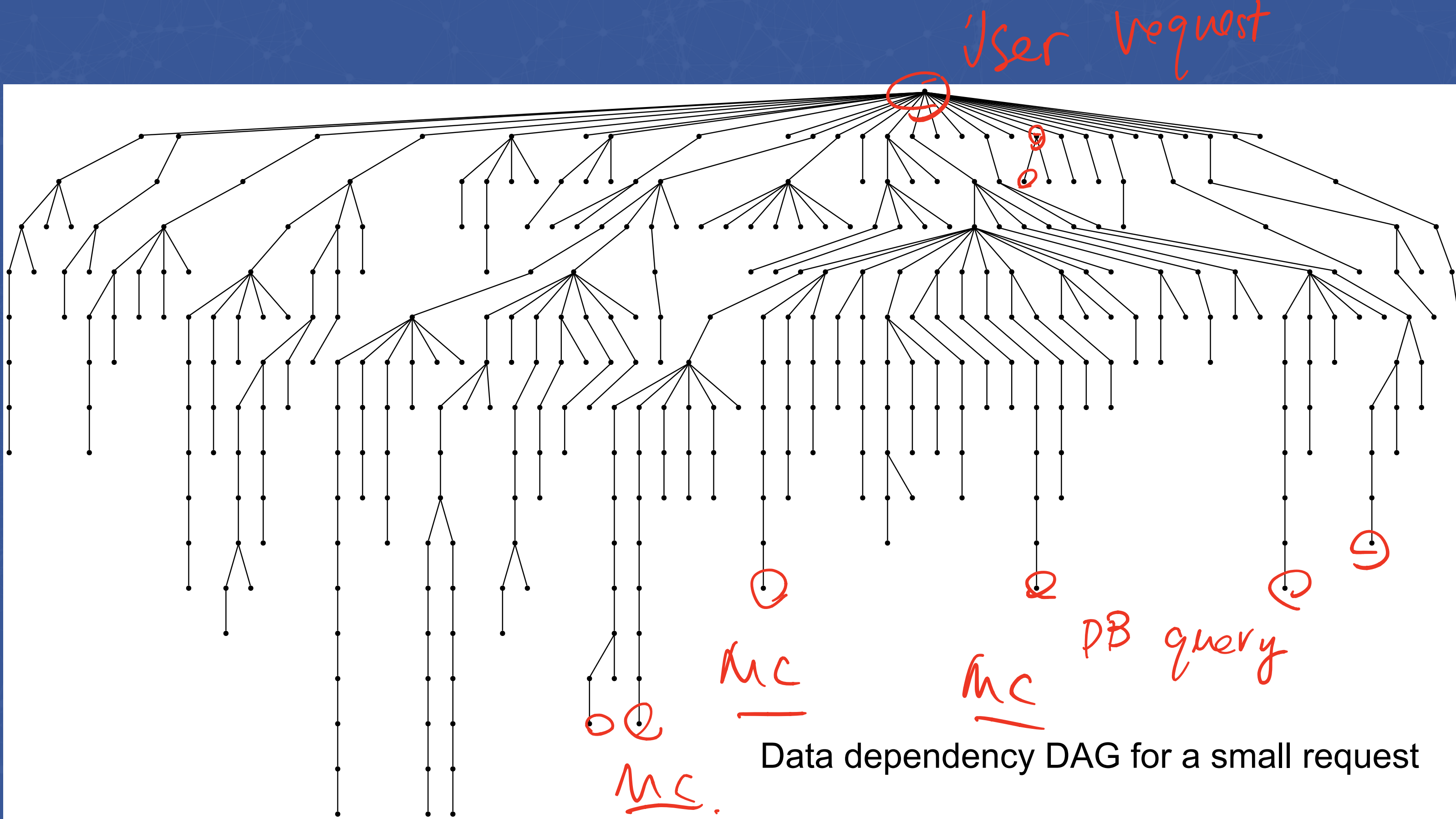
Just a few databases are enough to support the load



Data sharded across the databases

Why Separate Cache?

High fanout and multiple rounds of data fetching



Data dependency DAG for a small request

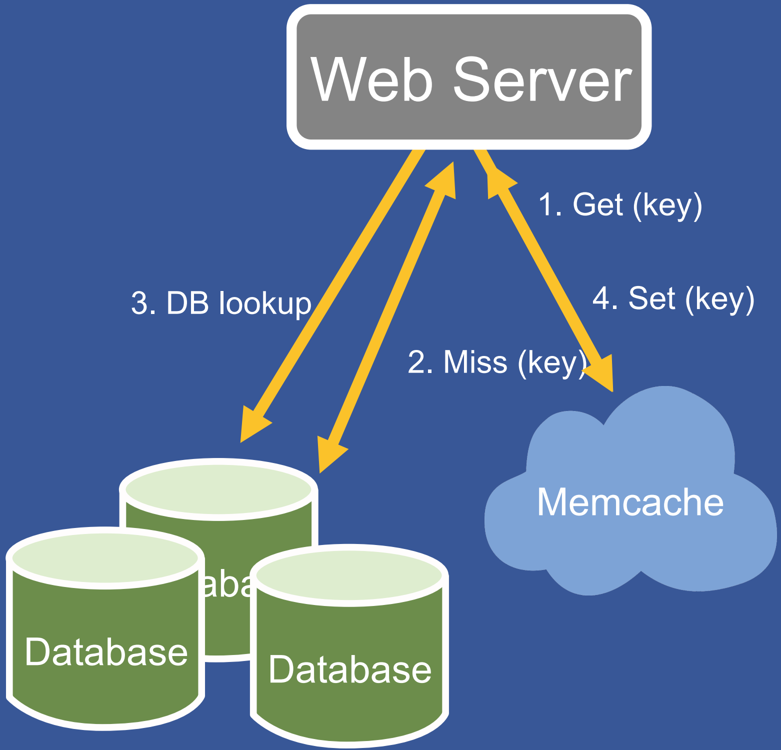
Scaling memcache in 4 “easy” steps

10s of servers & millions of operations per second

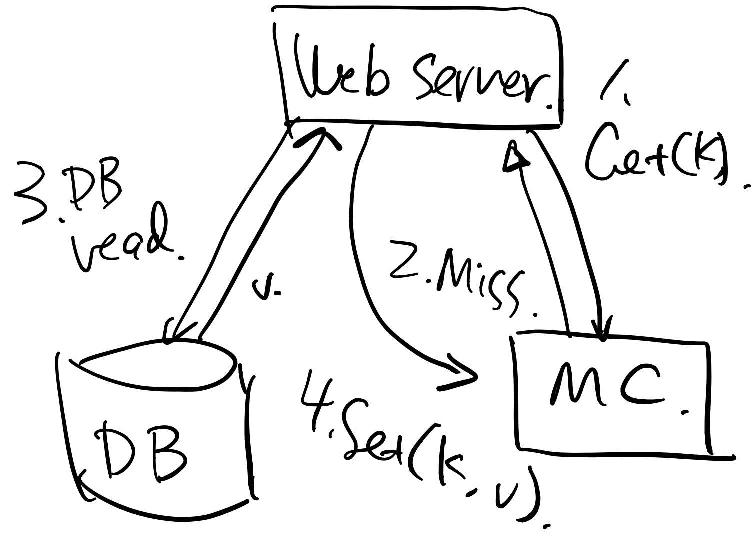
0	No memcache servers
1	A few memcache servers
2	Many memcache servers in one cluster
3	Many memcache servers in multiple clusters
4	Geographically distributed clusters

Need more read capacity

- Two orders of magnitude more reads than writes
- Solution: Deploy a few memcache hosts to handle the read capacity
- How do we store data?
  - Demand-filled look-aside cache
  - Common case is data is available in the cache



95+ % Read hit ratio



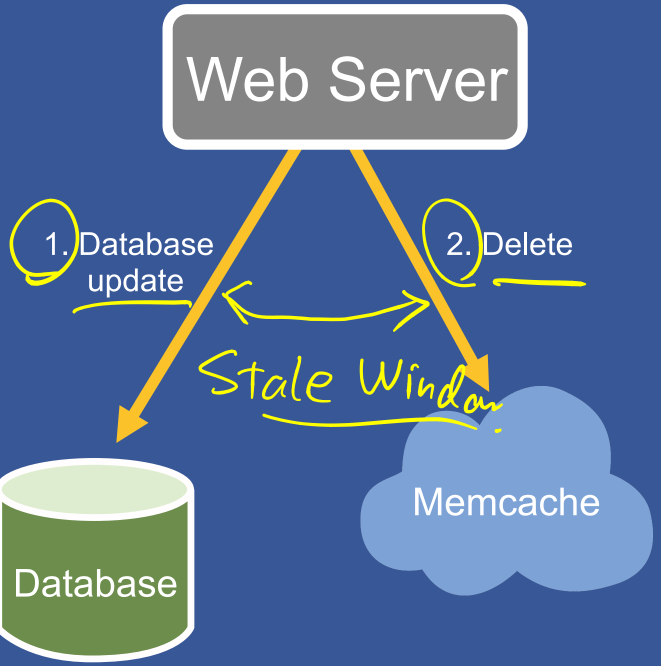
lookaside cache.

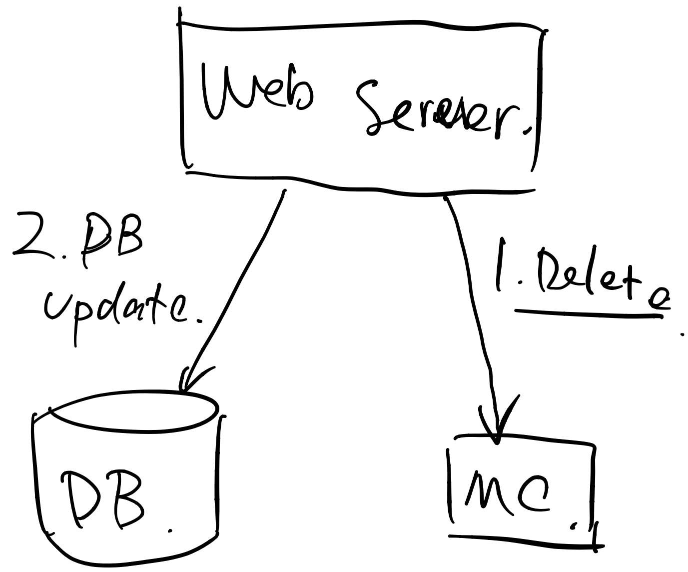
Handling updates

- Memcache needs to be invalidated after DB write
- Prefer deletes to sets
  - Idempotent
  - Demand filled

Deletes.

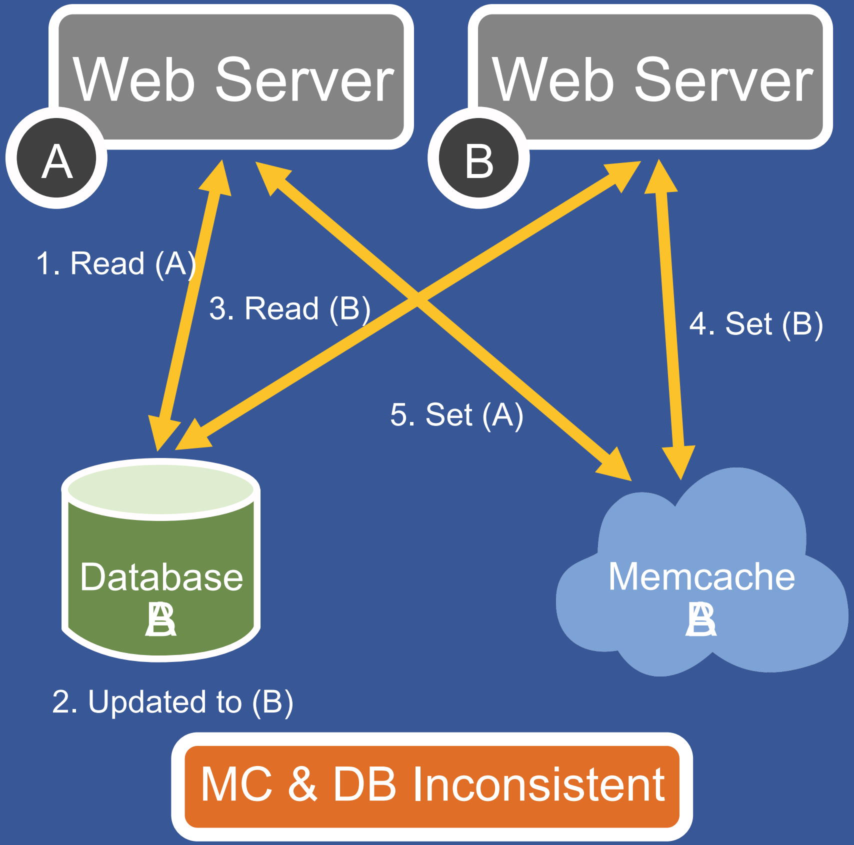
- Up to web application to specify which keys to invalidate after database update



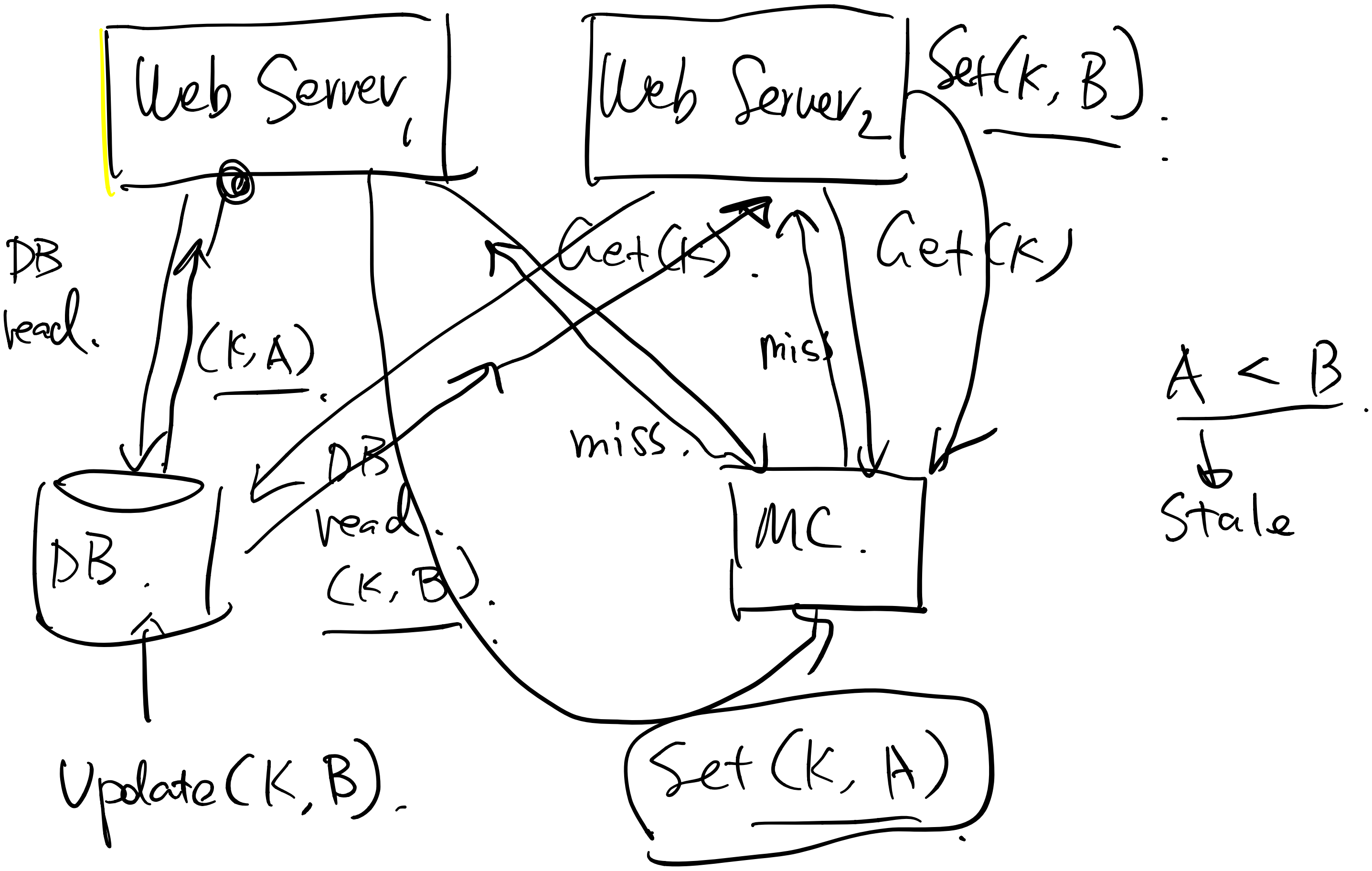

Web Server.
2. DB update.
1. Delete
DB.
MC.

Problems with look-aside caching

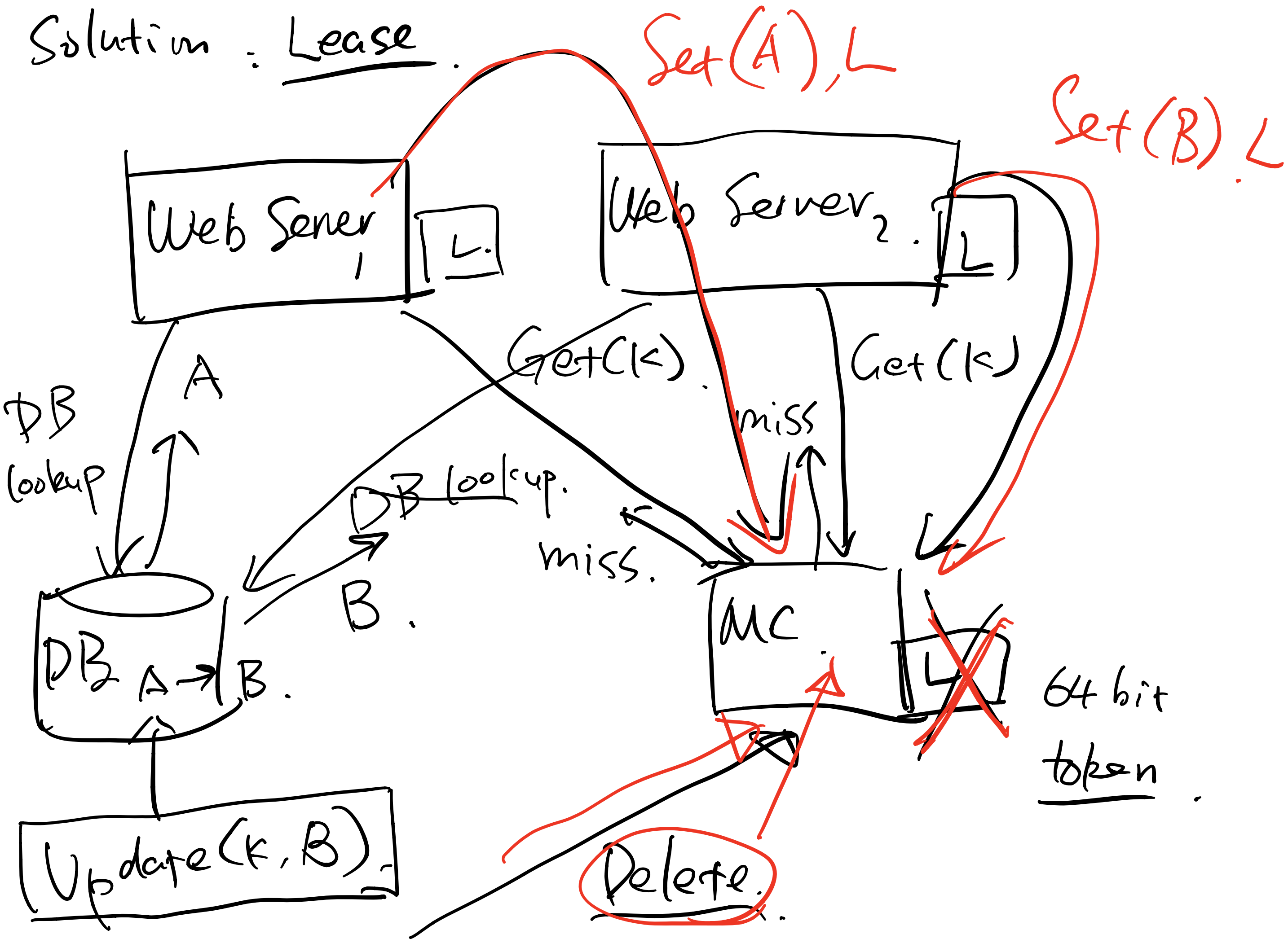
Stale Sets

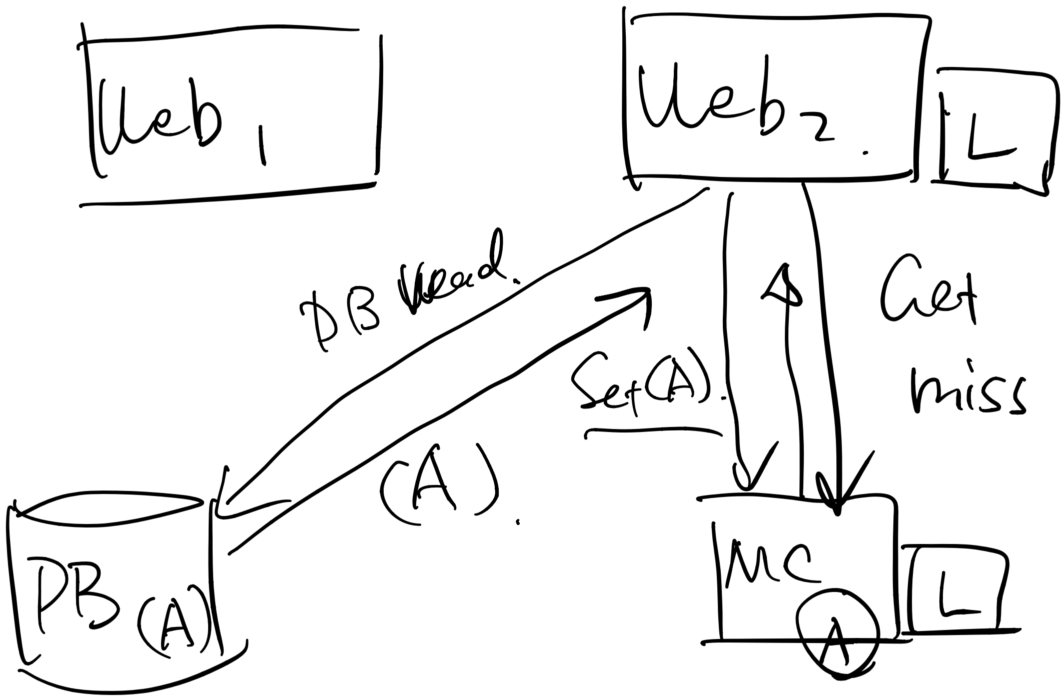


- Extend memcache protocol with “leases”
  - Return and attach a lease-id with every miss
  - Lease-id is invalidated inside server on a delete
  - Disallow set if the lease-id is invalid at the server



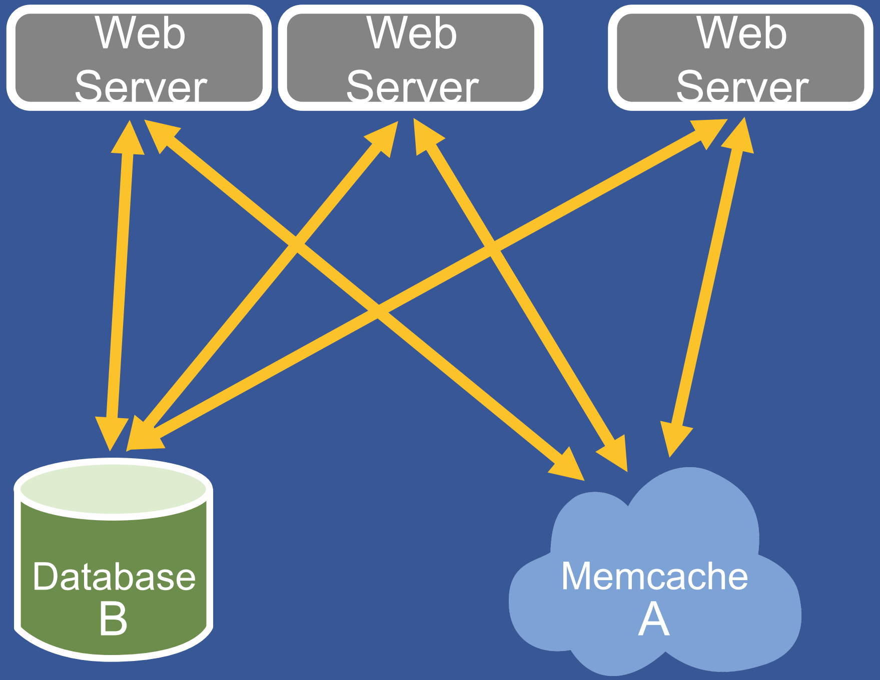
Solution: Lease.



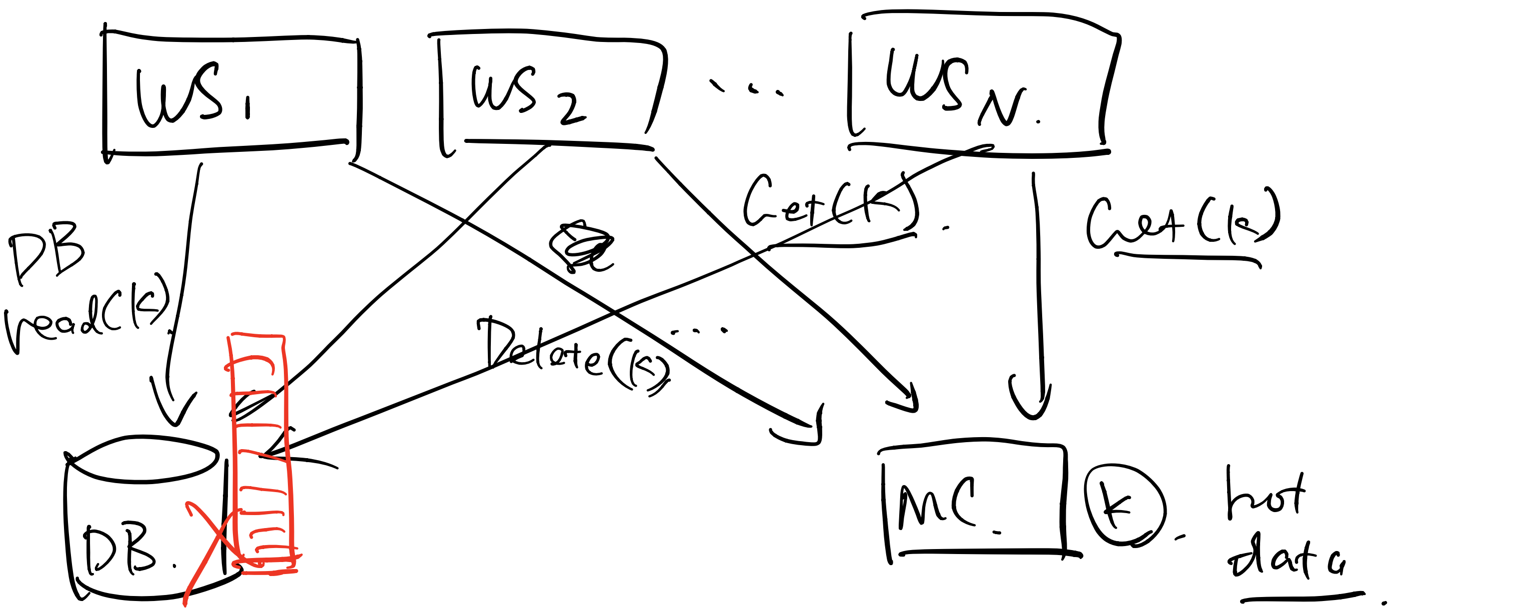

Web1
Web2.
L
DB read.
Get
miss
Set(A).
(A).
DB(A)
MC
A
L

Problems with look-aside caching

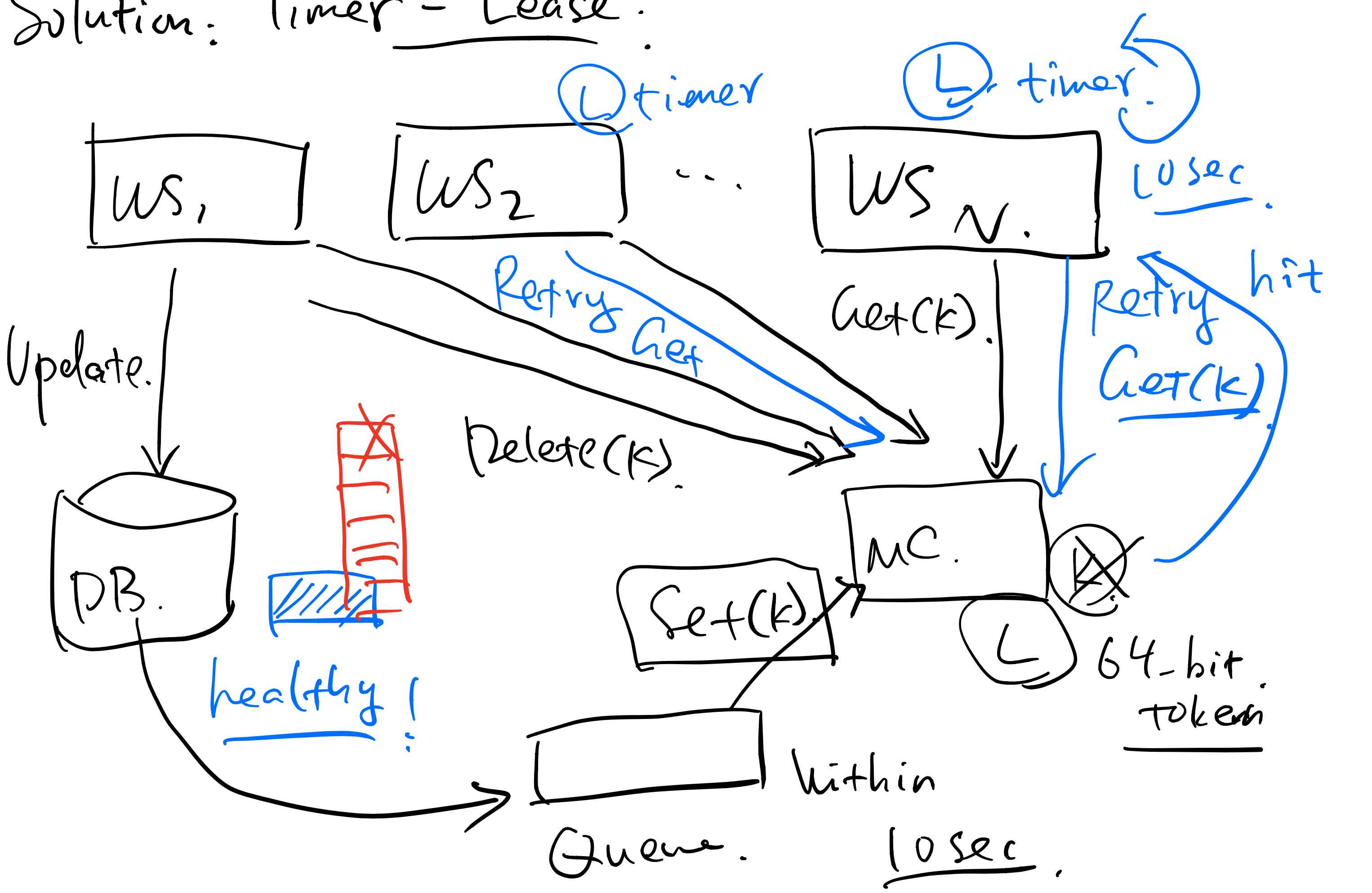
Thundering Herds



- Memcache server arbitrates access to database
  - Small extension to leases
- Clients given a choice of using a slightly stale value or waiting



Solution: Timer-Lease.



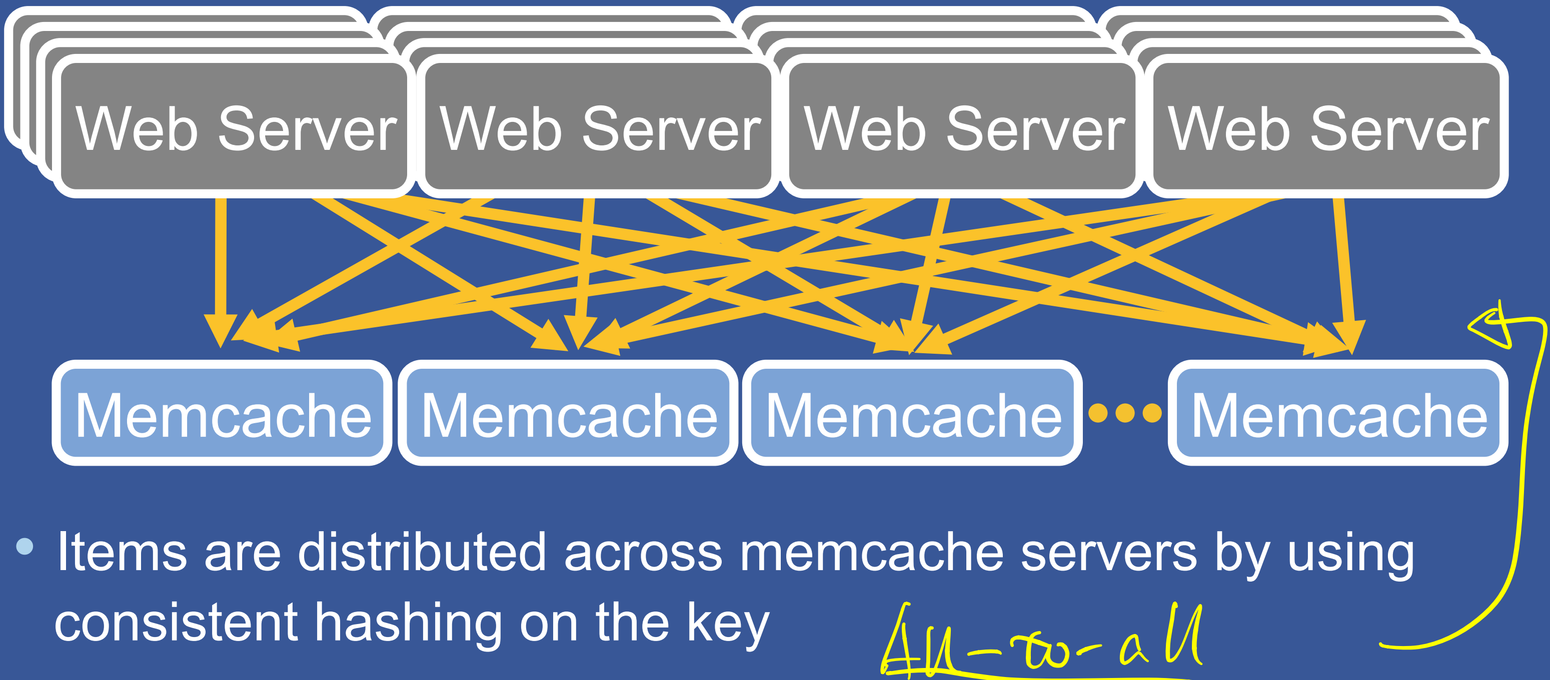
Bloom filter →

Scaling memcache in 4 “easy” steps

100s of servers & 10s of millions of operations per second

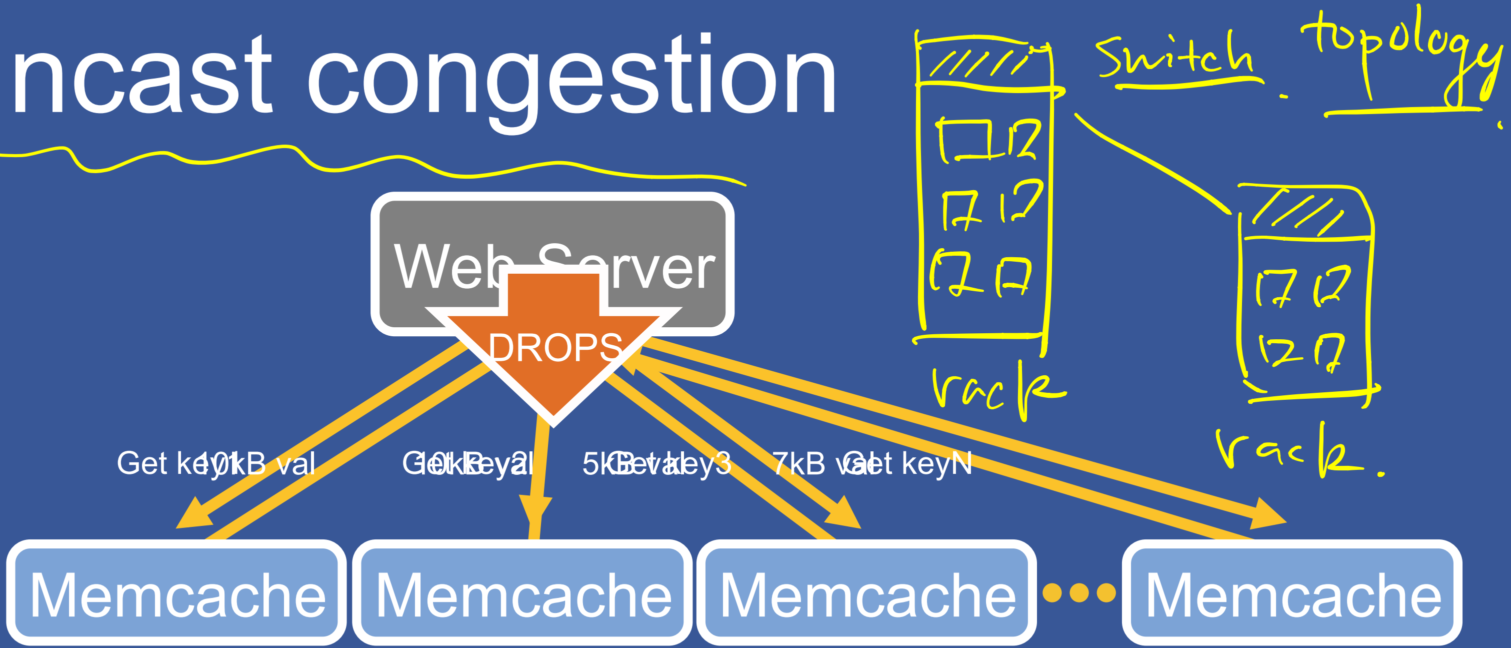
0	No memcache servers
1	A few memcache servers
2	Many memcache servers in one cluster
3	Many memcache servers in multiple clusters
4	Geographically distributed clusters

Need even more read capacity



- Items are distributed across memcache servers by using consistent hashing on the key
  - Individual items are rarely accessed very frequently so over replication doesn't make sense
- All web servers talk to all memcache servers
  - Accessing 100s of memcache servers to process a user request is common

Incast congestion



- Many simultaneous responses overwhelm shared networking resources
- Solution: Limit the number of outstanding requests with a sliding window
  - Larger windows cause result in more congestion
  - Smaller windows result in more round trips to the network

read-heavy

Mcrouter:

LB or proxy

300 Window.

Throttling

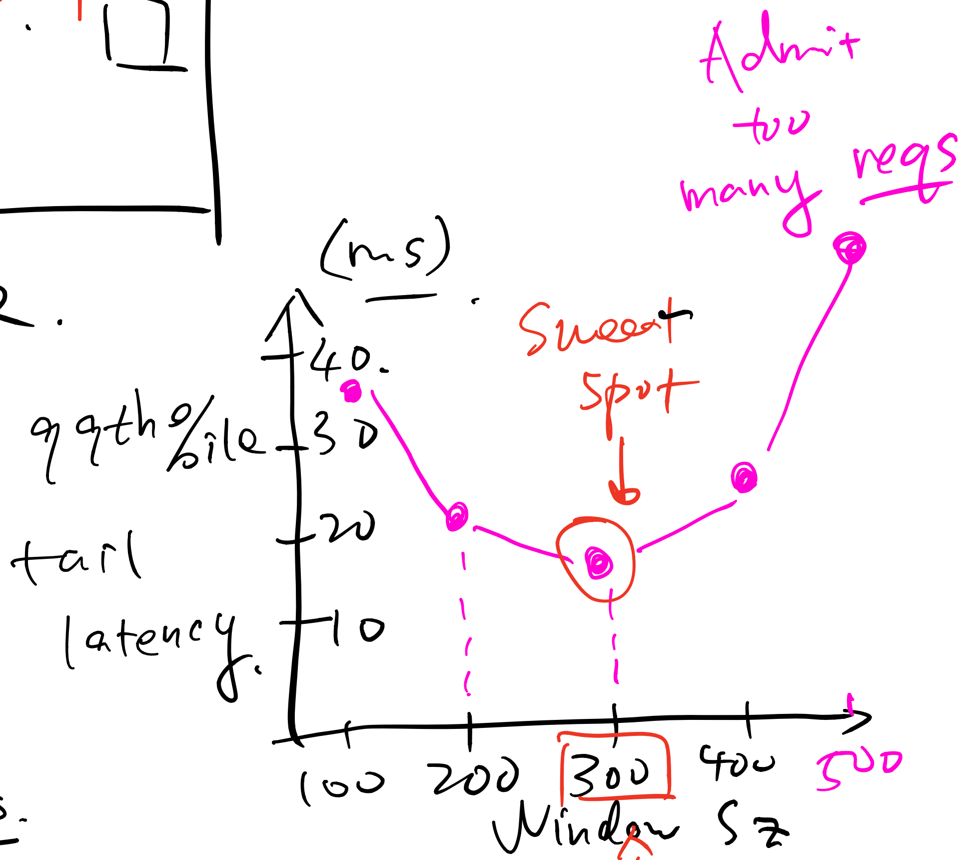
100ms

N tunable.

payload of 10s B

100s B.

< 1 ms.



Perf tuning

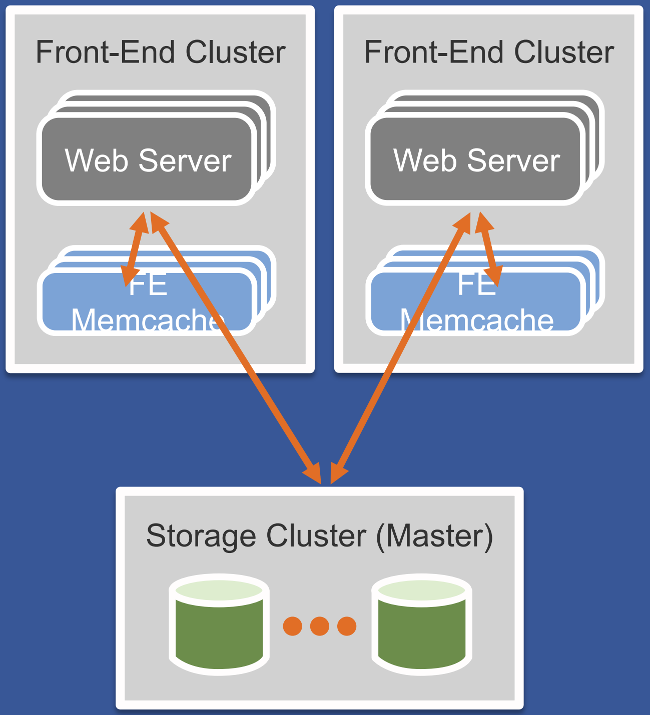
Scaling memcache in 4 "easy" steps

1000s of servers & 100s of millions of operations per second

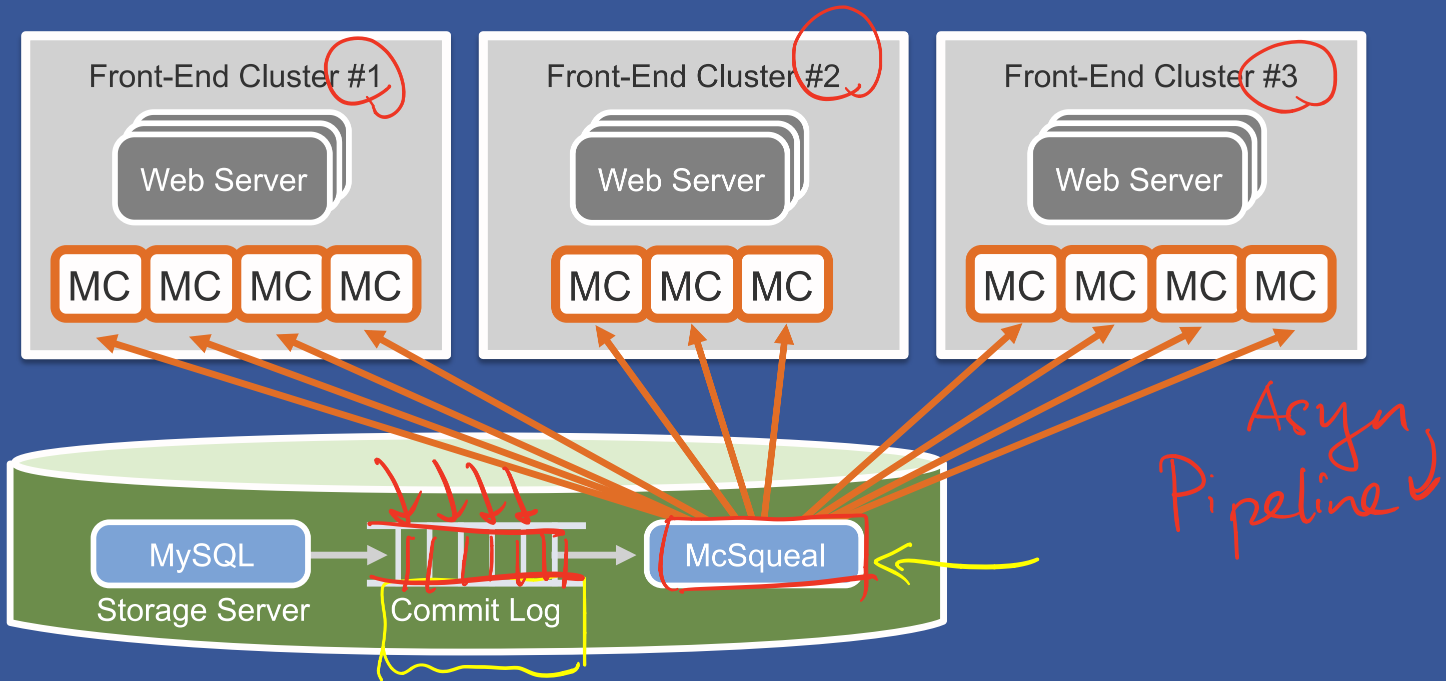
0	No memcache servers
1	A few memcache servers
2	Many memcache servers in one cluster
3	Many memcache servers in multiple clusters
4	Geographically distributed clusters

Multiple clusters

- All-to-all limits horizontal scaling
- Multiple memcache clusters front one DB installation
  - Have to keep the caches consistent
  - Have to manage over-replication of data



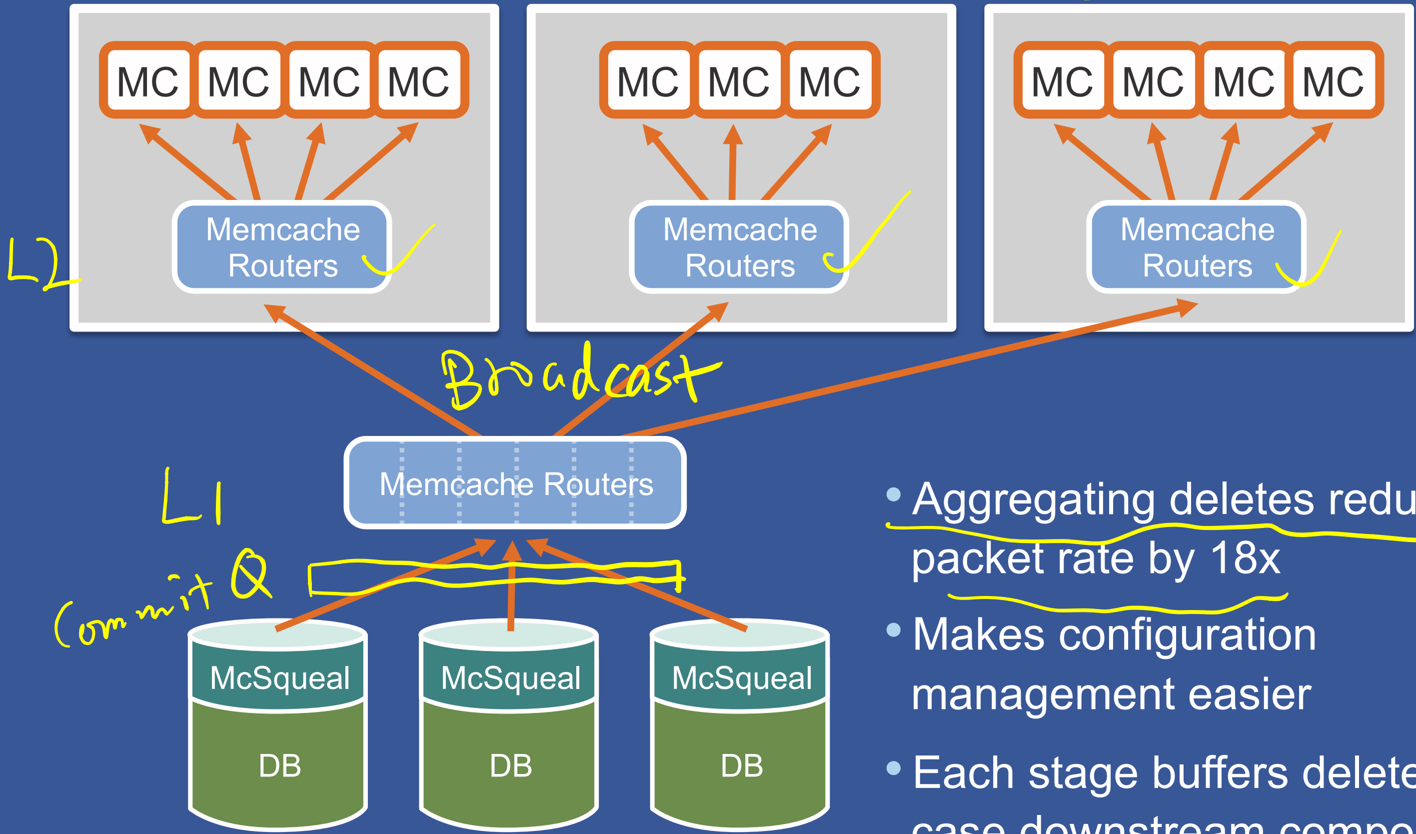
Databases invalidate caches



- Cached data must be invalidated after database updates
- Solution: Tail the mysql commit log and issue deletes based on transactions that have been committed
  - Allows caches to be resynchronized in the event of a problem

Invalidation pipeline

Too many packets



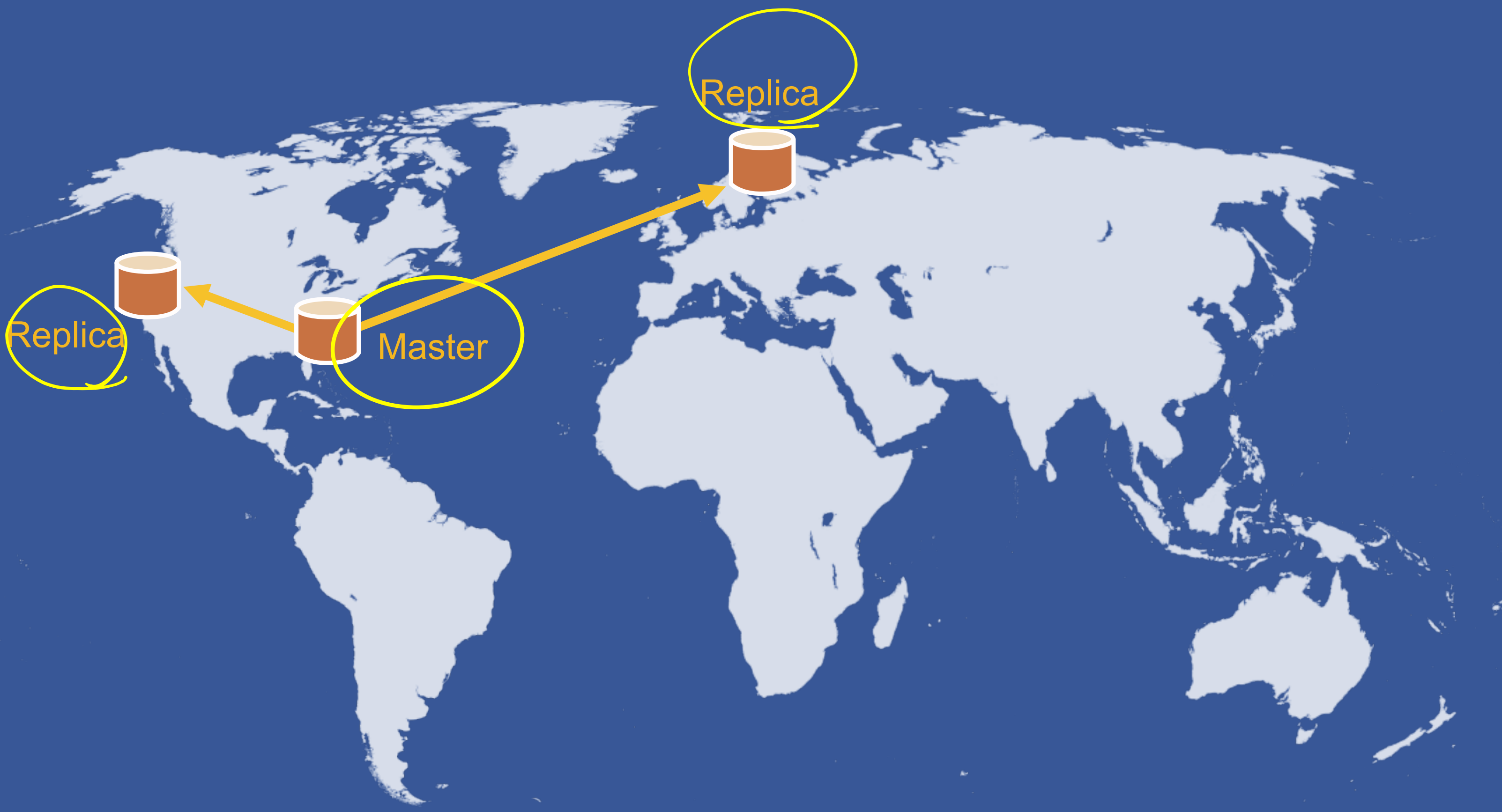
- Aggregating deletes reduces packet rate by 18x
- Makes configuration management easier
- Each stage buffers deletes in case downstream component is down

Scaling memcache in 4 “easy” steps

1000s of servers & > 1 billion operations per second

0	No memcache servers
1	A few memcache servers
2	Many memcache servers in one cluster
3	Many memcache servers in multiple clusters
4	Geographically distributed clusters

Geographically distributed clusters

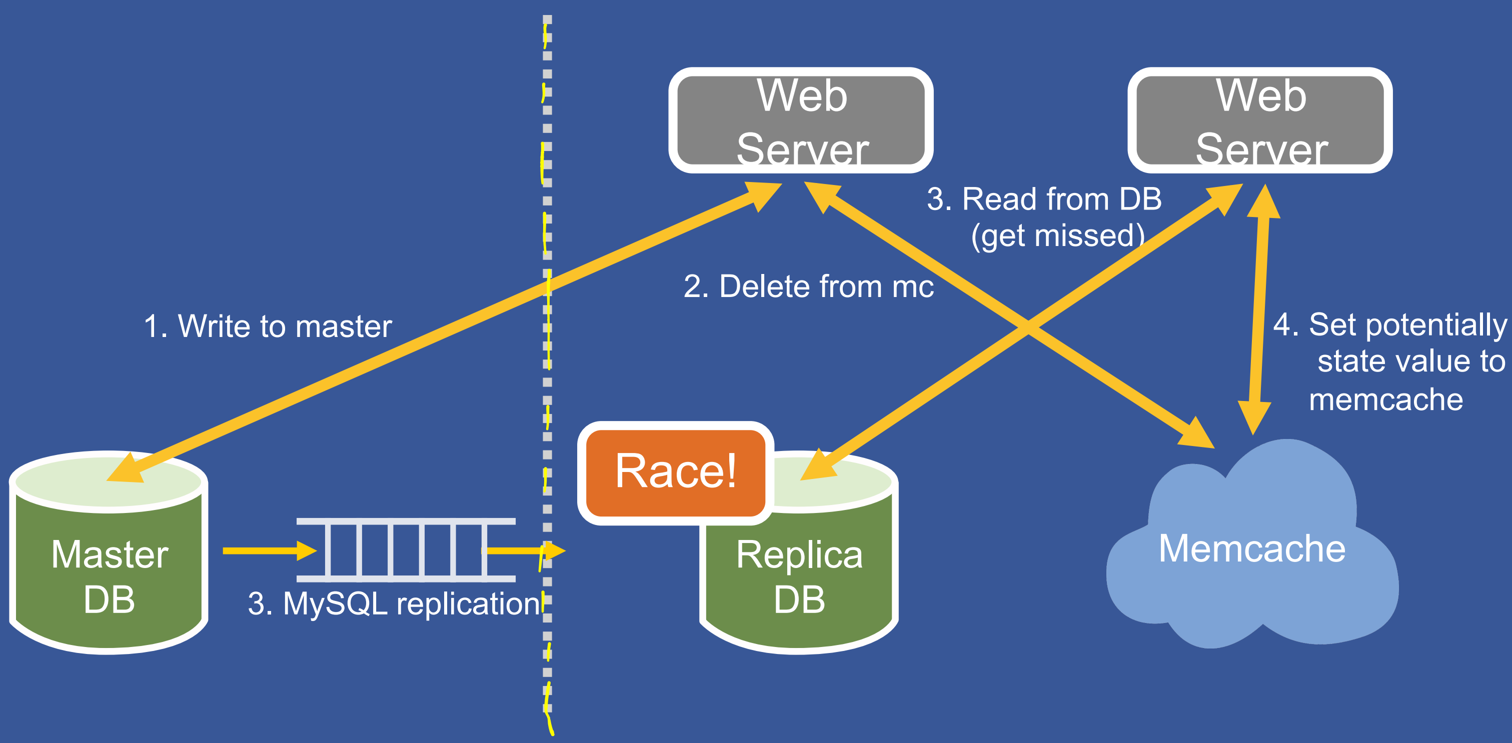


Writes in non-master

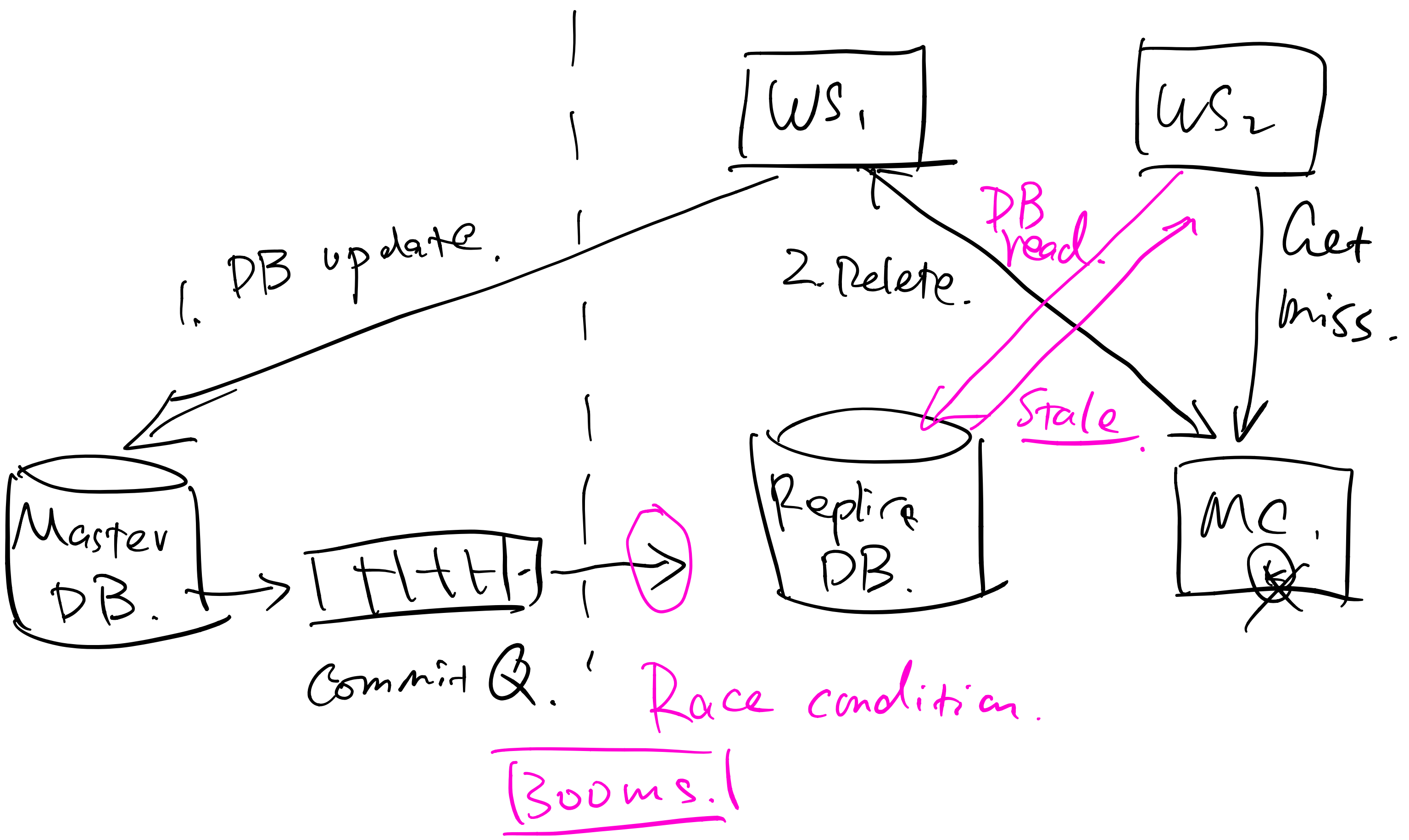
Database update directly in master

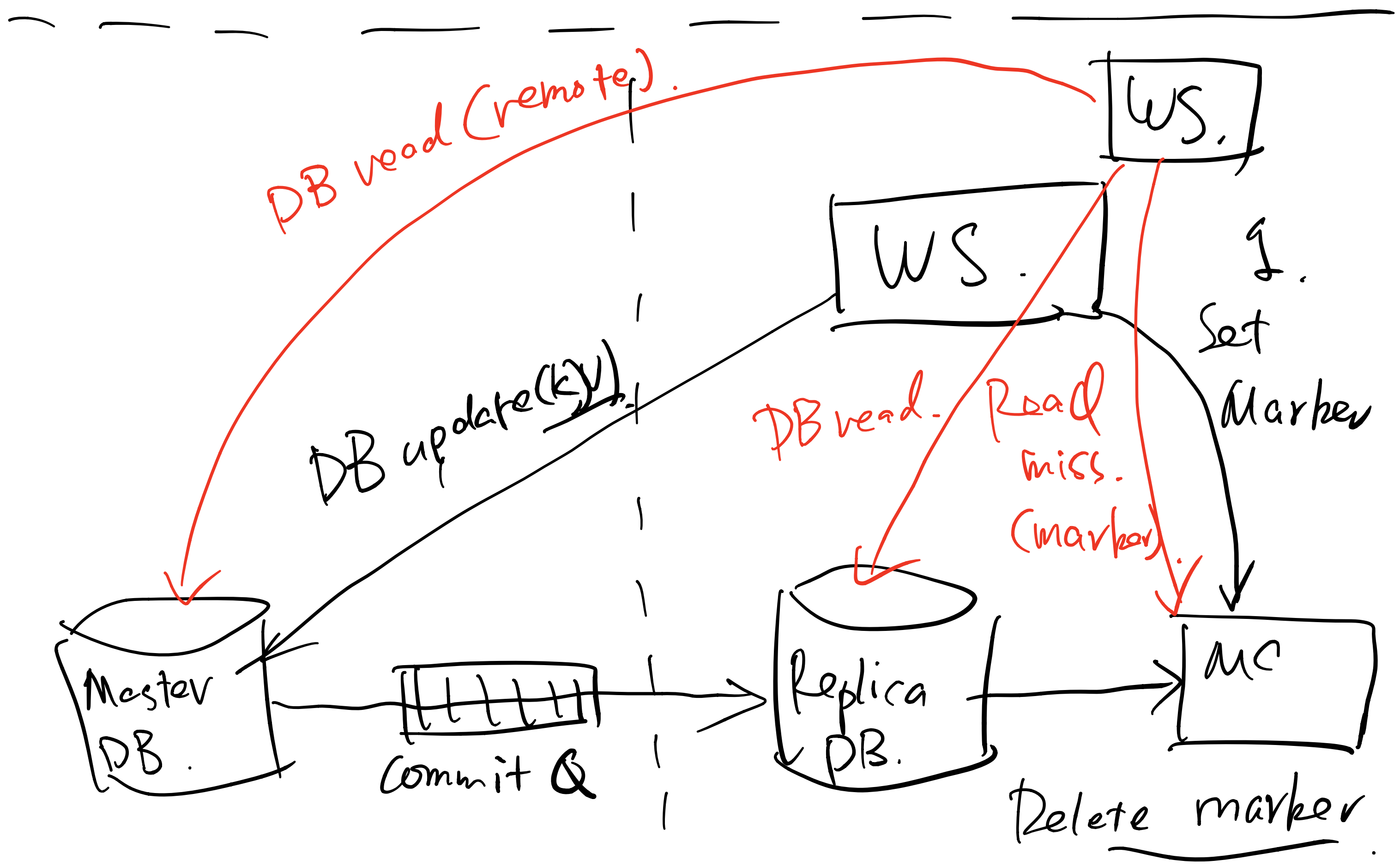
Virginia 100 – 500ms <1ms Seattle Replica 1

- Race between DB replication and subsequent DB read



Replica Site.





Remote markers

Set a special flag that indicates whether a race is likely

Lease

Eventual Consistency

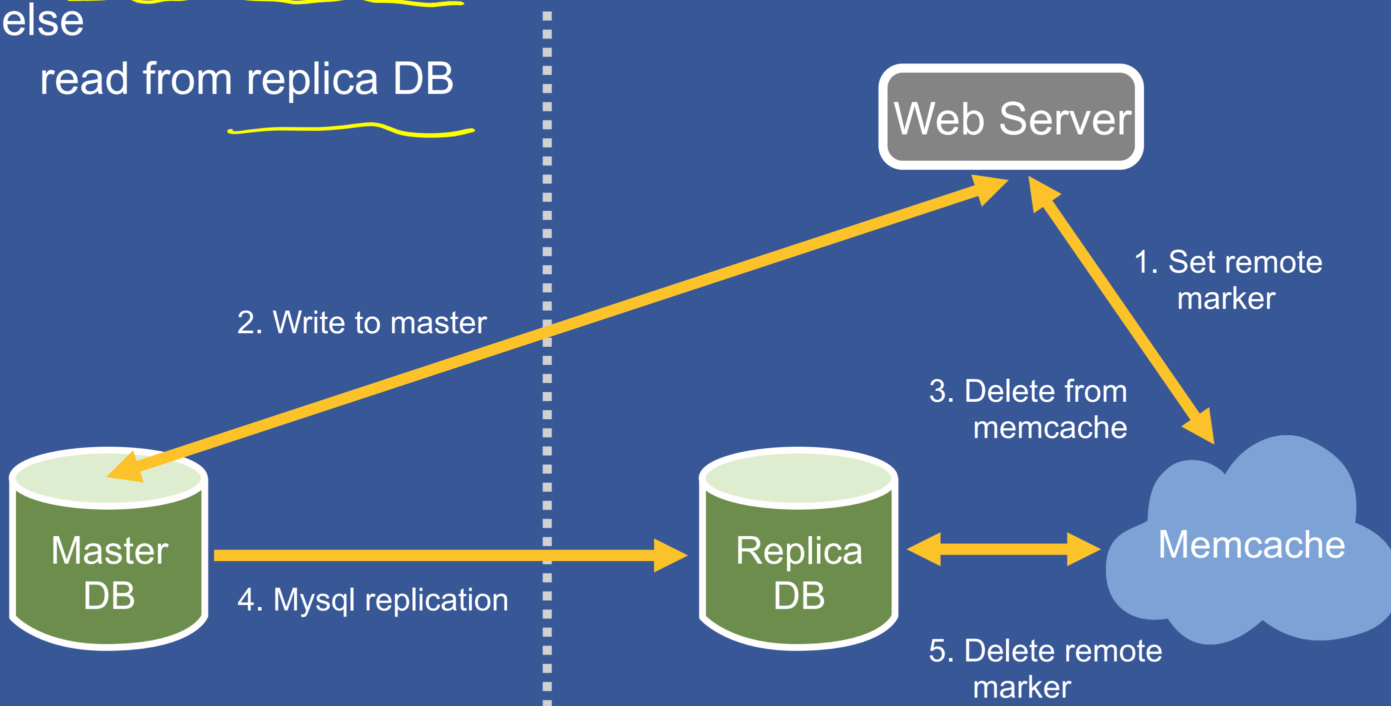
Read miss path:

If marker set

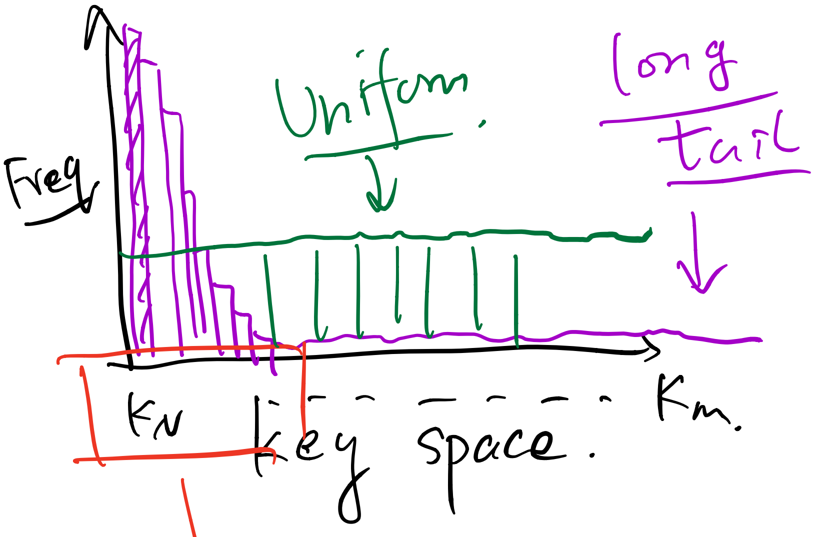
read from master DB

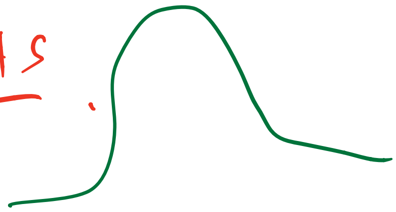
else

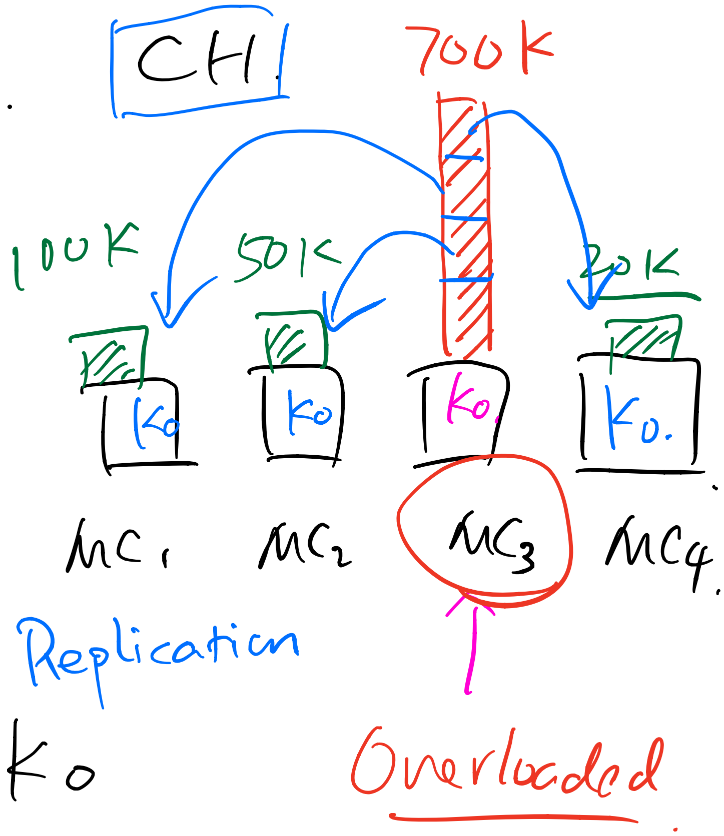
read from replica DB



Access load imbalance.







500K QPS

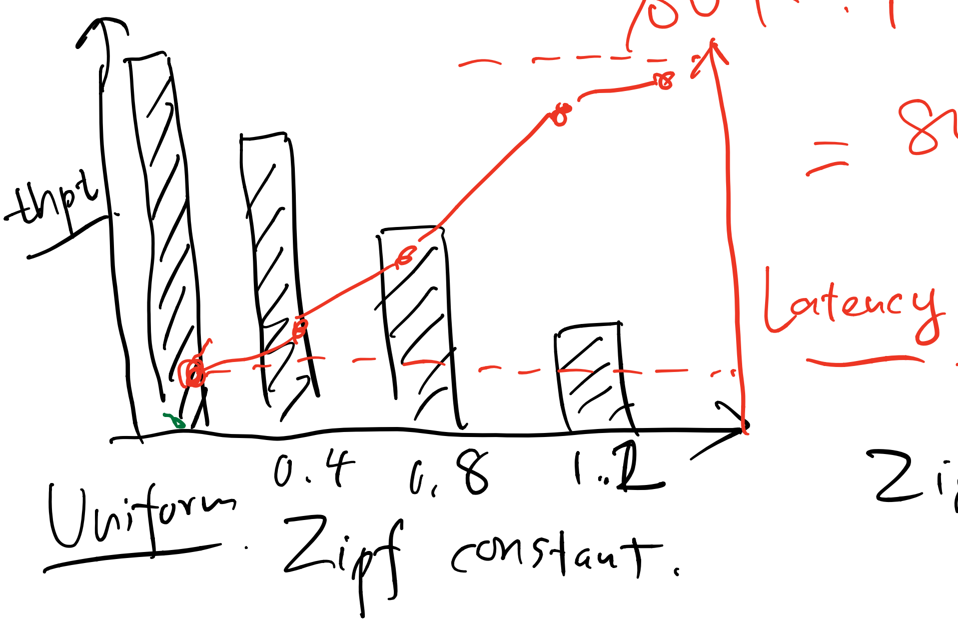
Normal:

500K × 4 = 2M QPS

700K + 100K + 50K + 20K

= 800 + 70K

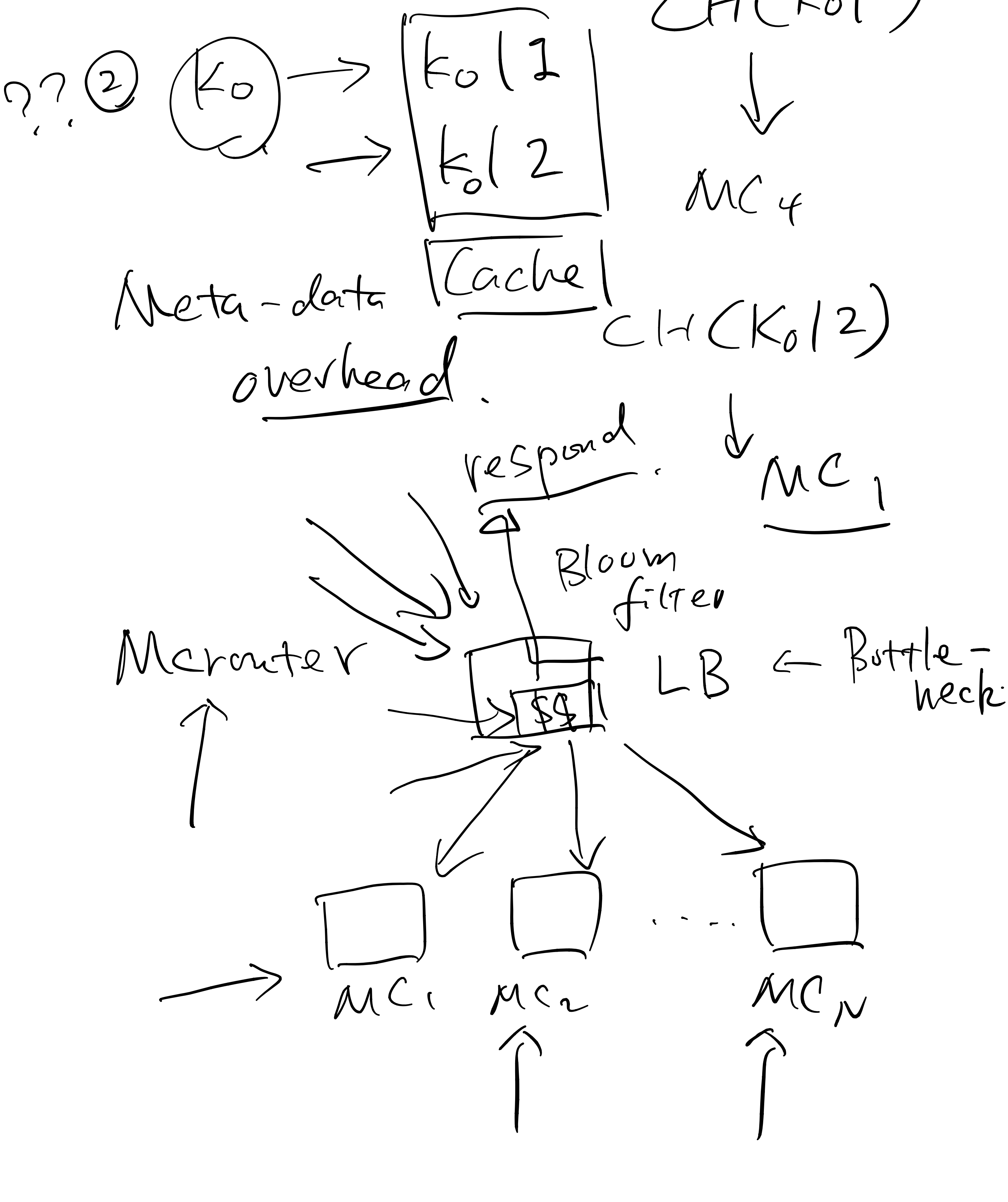
= 870K.



Zipfian Distribution.

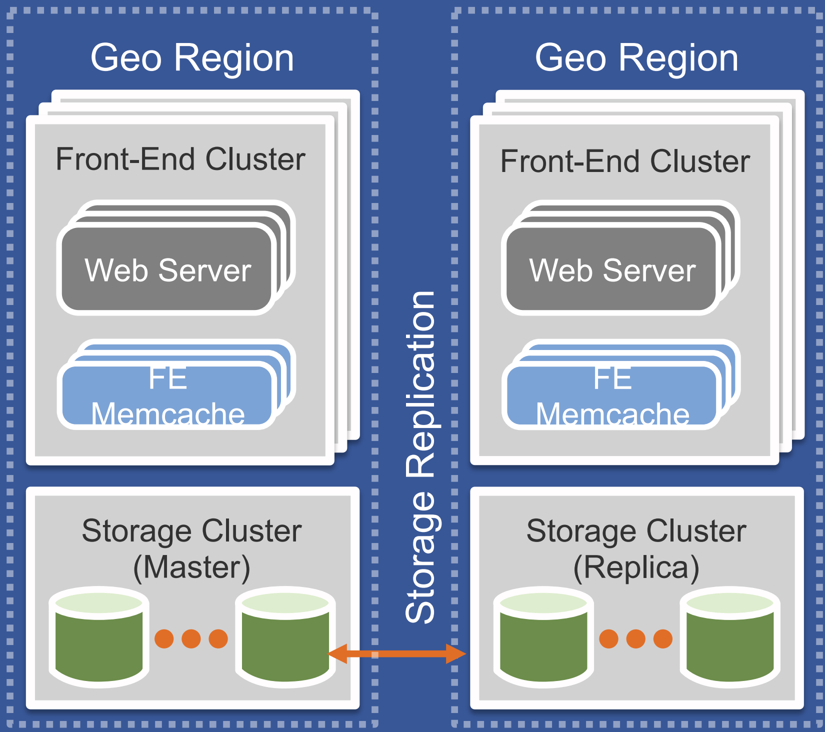
How to re-hash?

① Meta-data (LL).



Putting it all together

1. Single front-end cluster
   - Read heavy workload
   - Wide fanout
   - Handling failures
2. Multiple front-end clusters
   - Controlling data replication
   - Data consistency
3. Multiple Regions
   - Data consistency



Lessons Learned

- Push complexity into the client whenever possible
- Operational efficiency is as important as performance
- Separating cache and persistent store allows them to be scaled independently

Thanks! Questions?

http://www.facebook.com/careers

facebook