
facebook

Scaling Memcached 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
4. Scale to process millions of user requests per second

Design Requirements

Support a very heavy read load

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

ACID

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

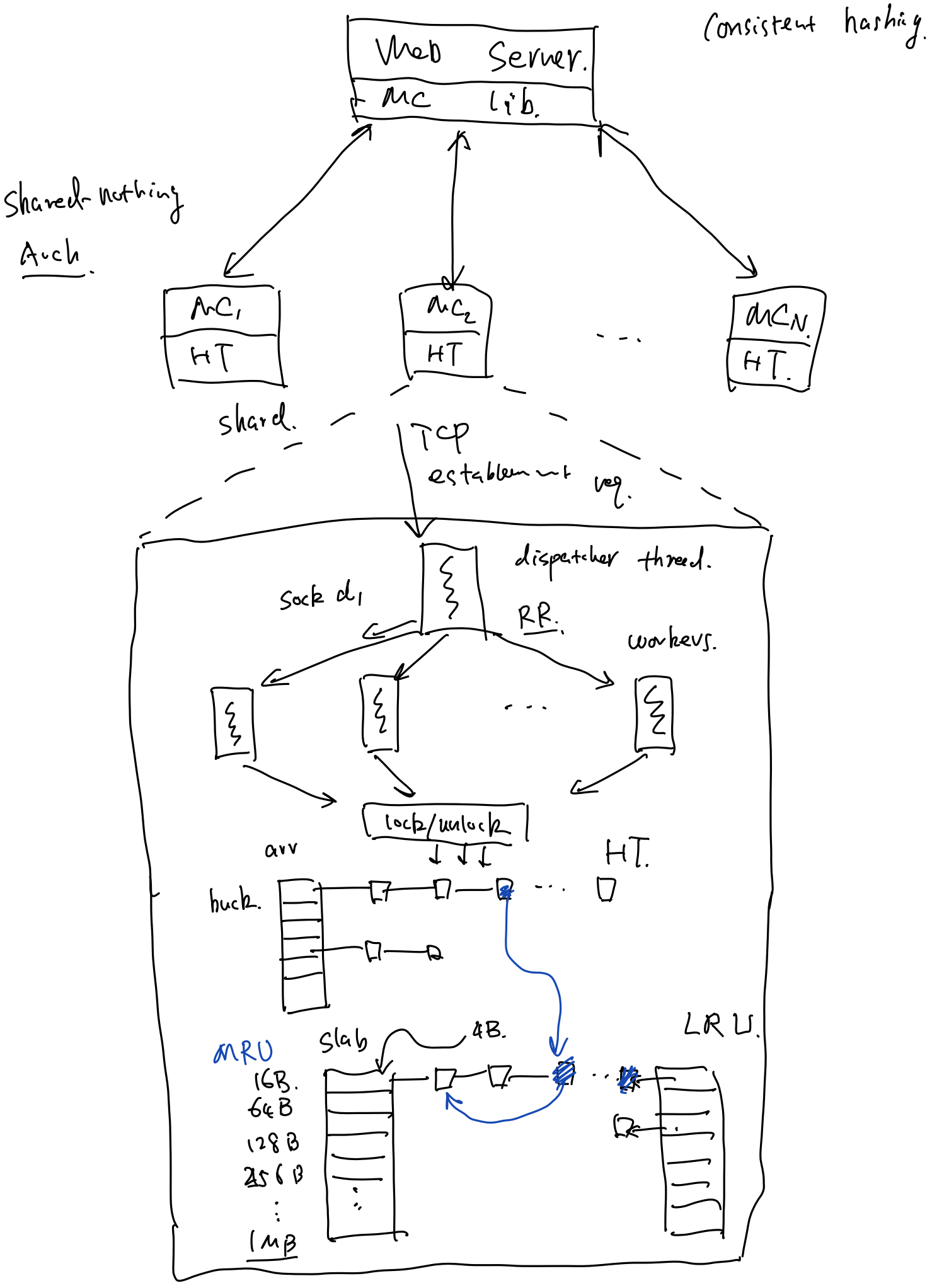
Persistence handled outside the system

- Support mechanisms to refill after updates

memcached

LRU.

- 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


Consistent hashing.
Web Server.
mc lib.
Shared-nothing
Arch.
mc1
HT
mc2
HT
...
mcN.
HT.
shared.
TCP
establishment req.
dispatcher thread.
sock d1
RR.
workers.
lock/unlock
arr
HT.
hash.
MRU
Slab
4B.
LRU.
16B.
64B
128B
256B
1MB

UserID → names

postID → text

news feed

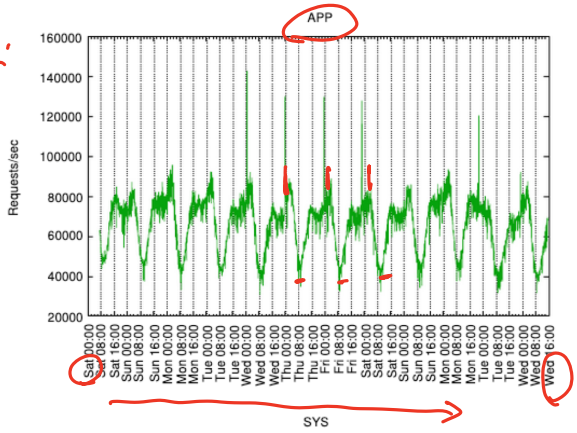
Table 1: Memcached pools sampled (in one cluster). These pools do not match their UNIX namesakes, but are used for illustrative purposes here instead of their internal names.

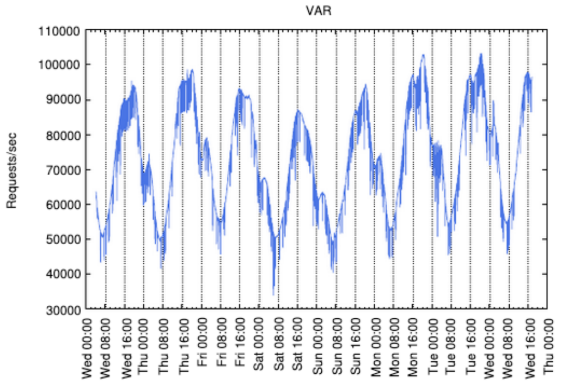
Pool	Size	Description
USR	few	user-account status information
APP	dozens	object metadata of one application
ETC	hundreds	nonspecific, general-purpose
VAR	dozens	server-side browser information
SYS	few	system data on service location

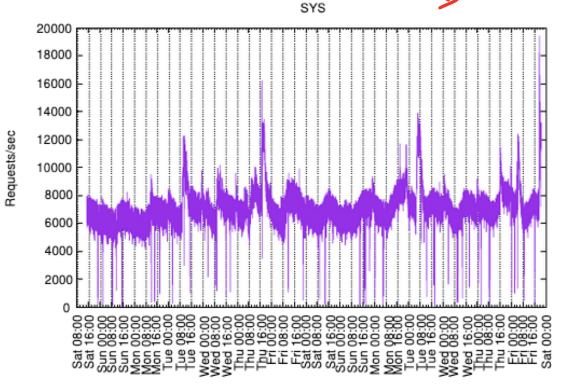
copies of data by DBMS

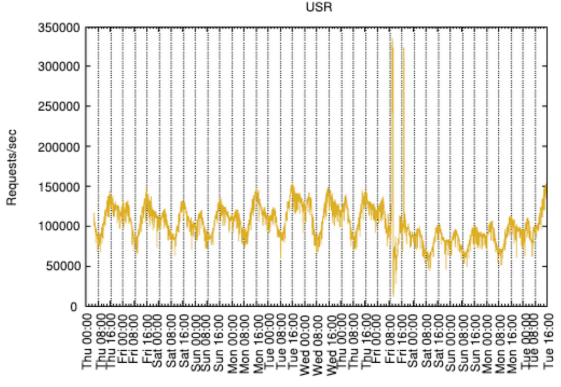
*: Workload analysis of a large-scale key-value store, Sigmetrics 2012

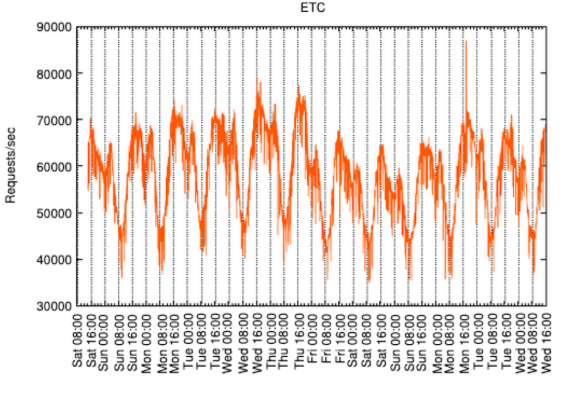
QPS.

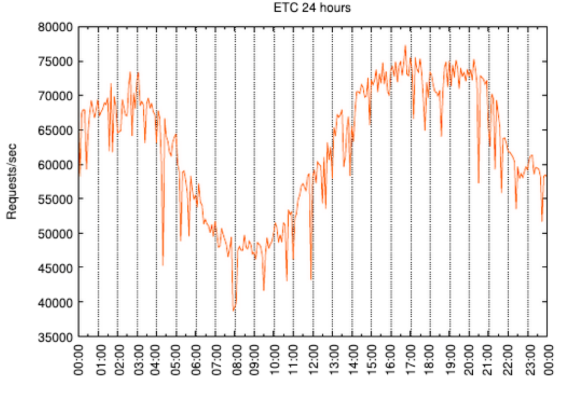

APP
Requests/sec
160000
140000
120000
100000
80000
60000
40000
20000
Sat 00:00
Sat 08:00
Sat 16:00
Sun 00:00
Sun 08:00
Sun 16:00
Mon 00:00
Mon 08:00
Mon 16:00
Tue 00:00
Tue 08:00
Tue 16:00
Wed 00:00
Wed 08:00
Wed 16:00
Thu 00:00
Thu 08:00
Thu 16:00
Fri 00:00
Fri 08:00
Fri 16:00
Sat 00:00
Sat 08:00
Sat 16:00
Sun 00:00
Sun 08:00
Sun 16:00
Mon 00:00
Mon 08:00
Mon 16:00
Tue 00:00
Tue 08:00
Tue 16:00
Wed 00:00
Wed 08:00
Wed 16:00


VAR
Requests/sec
110000
100000
90000
80000
70000
60000
50000
40000
30000
Wed 00:00
Wed 08:00
Wed 16:00
Thu 00:00
Thu 08:00
Thu 16:00
Fri 00:00
Fri 08:00
Fri 16:00
Sat 00:00
Sat 08:00
Sat 16:00
Sun 00:00
Sun 08:00
Sun 16:00
Mon 00:00
Mon 08:00
Mon 16:00
Tue 00:00
Tue 08:00
Tue 16:00
Wed 00:00
Wed 08:00
Wed 16:00
Thu 00:00


SYS
Requests/sec
20000
18000
16000
14000
12000
10000
8000
6000
4000
2000
0
Sat 08:00
Sat 16:00
Sun 00:00
Sun 08:00
Sun 16:00
Mon 00:00
Mon 08:00
Mon 16:00
Tue 00:00
Tue 08:00
Tue 16:00
Wed 00:00
Wed 08:00
Wed 16:00
Thu 00:00
Thu 08:00
Thu 16:00
Fri 00:00
Fri 08:00
Fri 16:00
Sat 00:00
Sat 08:00
Sat 16:00
Sun 00:00
Sun 08:00
Sun 16:00
Mon 00:00
Mon 08:00
Mon 16:00
Tue 00:00
Tue 08:00
Tue 16:00
Wed 00:00
Wed 08:00
Wed 16:00
Thu 00:00
Thu 08:00
Thu 16:00
Fri 00:00
Fri 08:00
Fri 16:00
Sat 00:00


USR
Requests/sec
350000
300000
250000
200000
150000
100000
50000
0
Thu 00:00
Thu 08:00
Thu 16:00
Fri 00:00
Fri 08:00
Fri 16:00
Sat 00:00
Sat 08:00
Sat 16:00
Sun 00:00
Sun 08:00
Sun 16:00
Mon 00:00
Mon 08:00
Mon 16:00
Tue 00:00
Tue 08:00
Tue 16:00
Wed 00:00
Wed 08:00
Wed 16:00
Thu 00:00
Thu 08:00
Thu 16:00
Fri 00:00
Fri 08:00
Fri 16:00
Sat 00:00
Sat 08:00
Sat 16:00
Sun 00:00
Sun 08:00
Sun 16:00
Mon 00:00
Mon 08:00
Mon 16:00
Tue 00:00
Tue 08:00
Tue 16:00

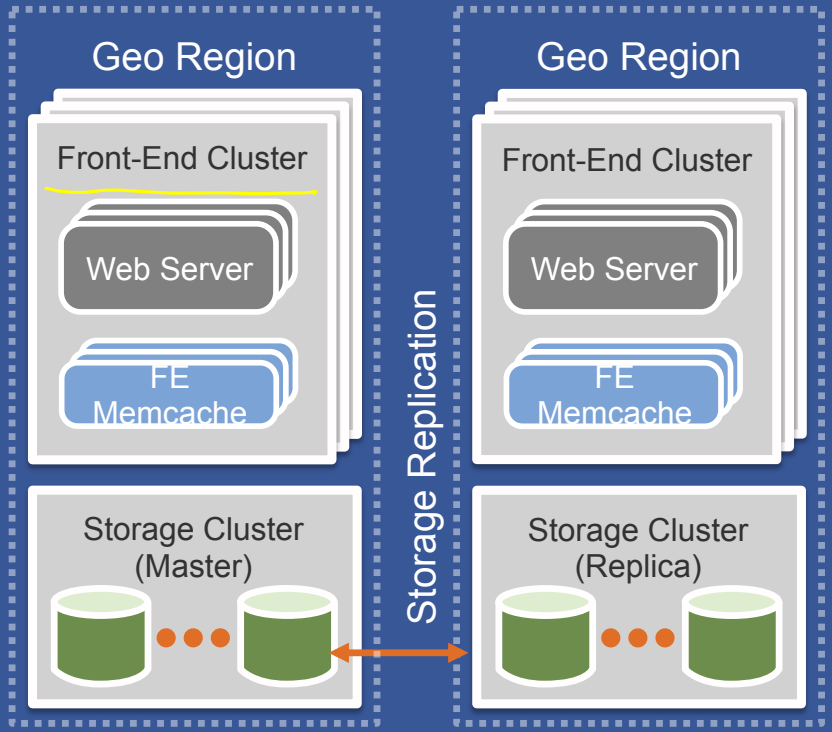

ETC
Requests/sec
90000
80000
70000
60000
50000
40000
30000
Sat 08:00
Sat 16:00
Sun 00:00
Sun 08:00
Sun 16:00
Mon 00:00
Mon 08:00
Mon 16:00
Tue 00:00
Tue 08:00
Tue 16:00
Wed 00:00
Wed 08:00
Wed 16:00
Thu 00:00
Thu 08:00
Thu 16:00
Fri 00:00
Fri 08:00
Fri 16:00
Sat 00:00
Sat 08:00
Sat 16:00
Sun 00:00
Sun 08:00
Sun 16:00
Mon 00:00
Mon 08:00
Mon 16:00
Tue 00:00
Tue 08:00
Tue 16:00
Wed 00:00
Wed 08:00
Wed 16:00


ETC 24 hours
Requests/sec
80000
75000
70000
65000
60000
55000
50000
45000
40000
35000
00:00
01:00
02:00
03:00
04:00
05:00
06:00
07:00
08:00
09:00
10:00
11:00
12:00
13:00
14:00
15:00
16:00
17:00
18:00
19:00
20:00
21:00
22:00
23:00
00:00

Roadmap

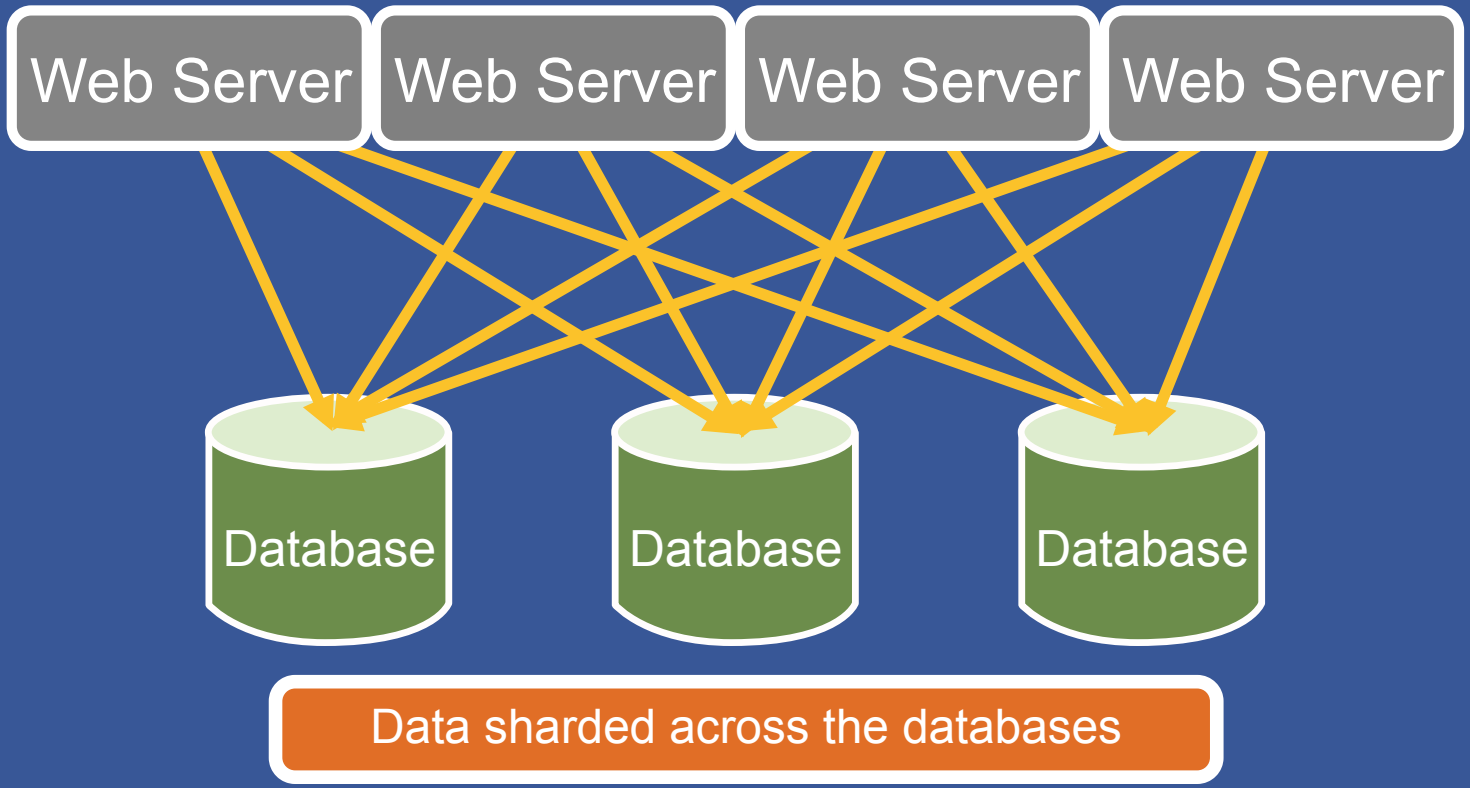
0. Pre-memcached. Era.

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

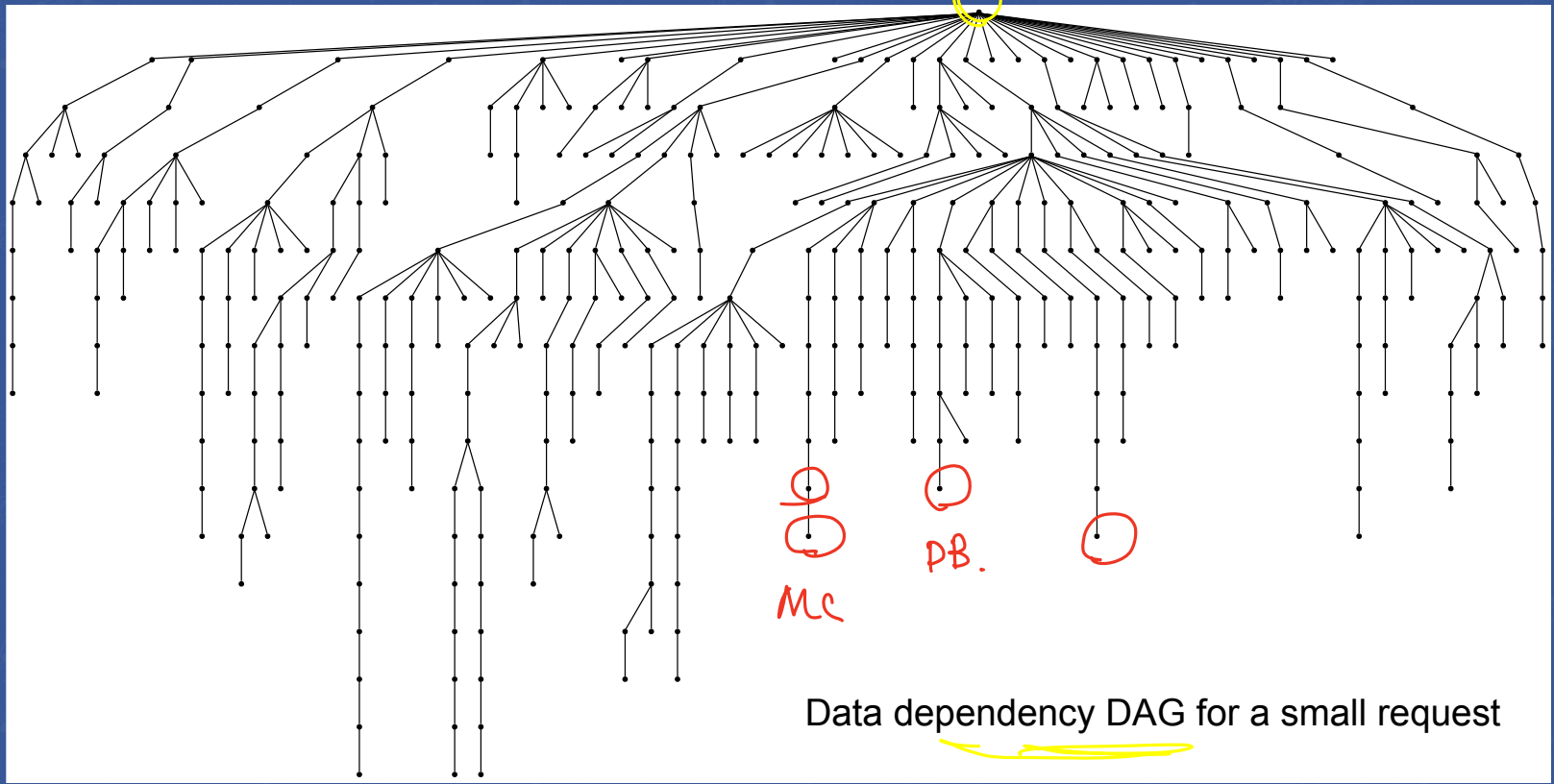


Why Separate Cache?

multi-get.

High fanout and multiple rounds of data fetching

Page Rendering Req



Data dependency DAG for a small request

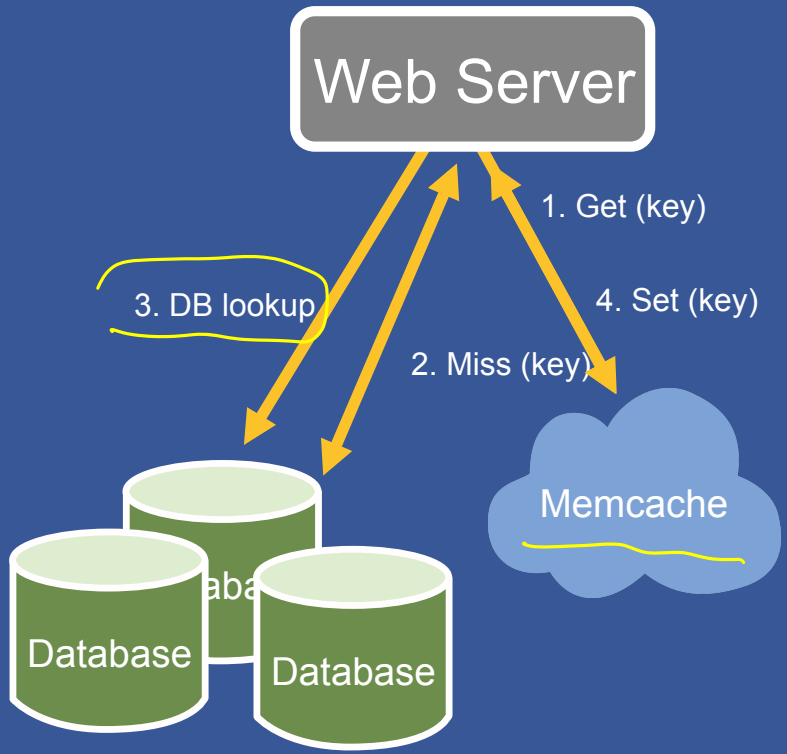
Scaling memcache in 4 “~~easy~~” steps

10s of servers & millions of operations per second

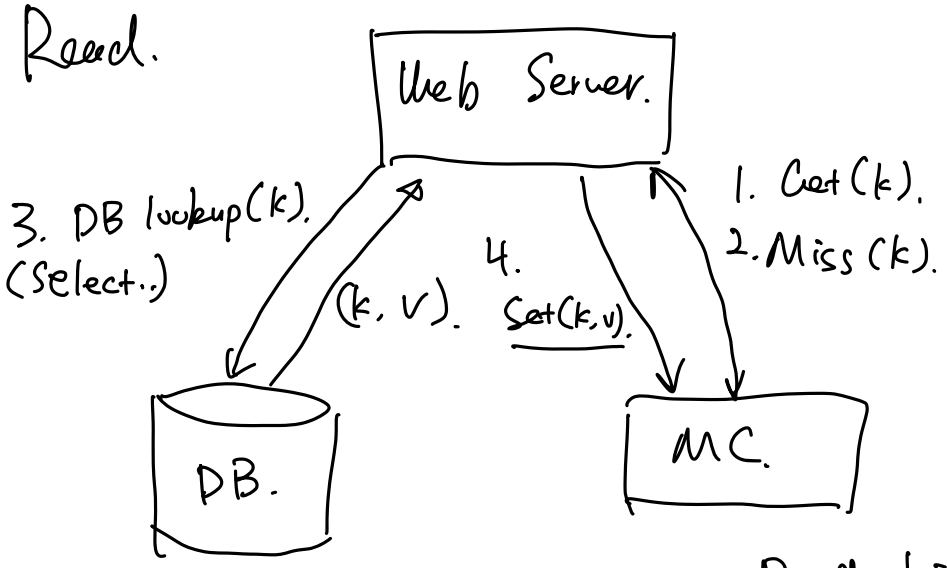
0	No memcache servers
1	A few memcache servers
2	Many memcache servers in one cluster
3	Many memcache servers in multiple clusters
4	Geographically distributed clusters

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



Read.

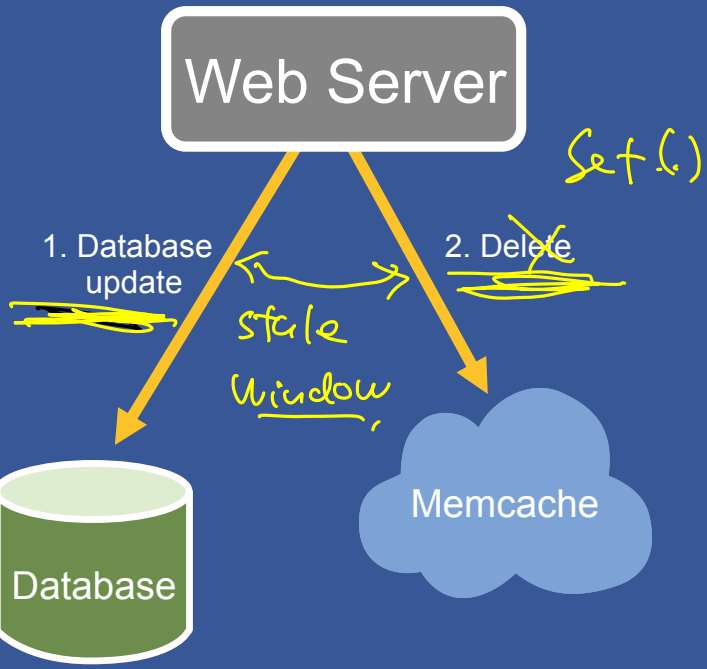


Read hits ratio > 95%

misses are rare!

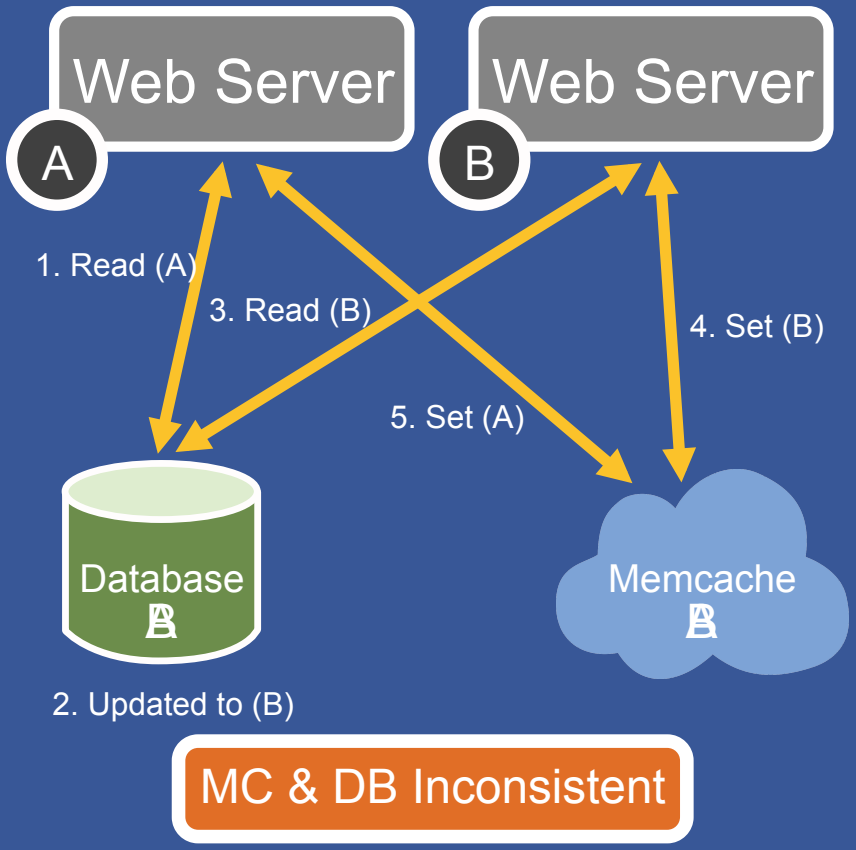
Handling updates

- Memcache needs to be invalidated after DB write
- Prefer deletes to sets
  - Idempotent
  - Demand filled
- Up to web application to specify which keys to invalidate after database update



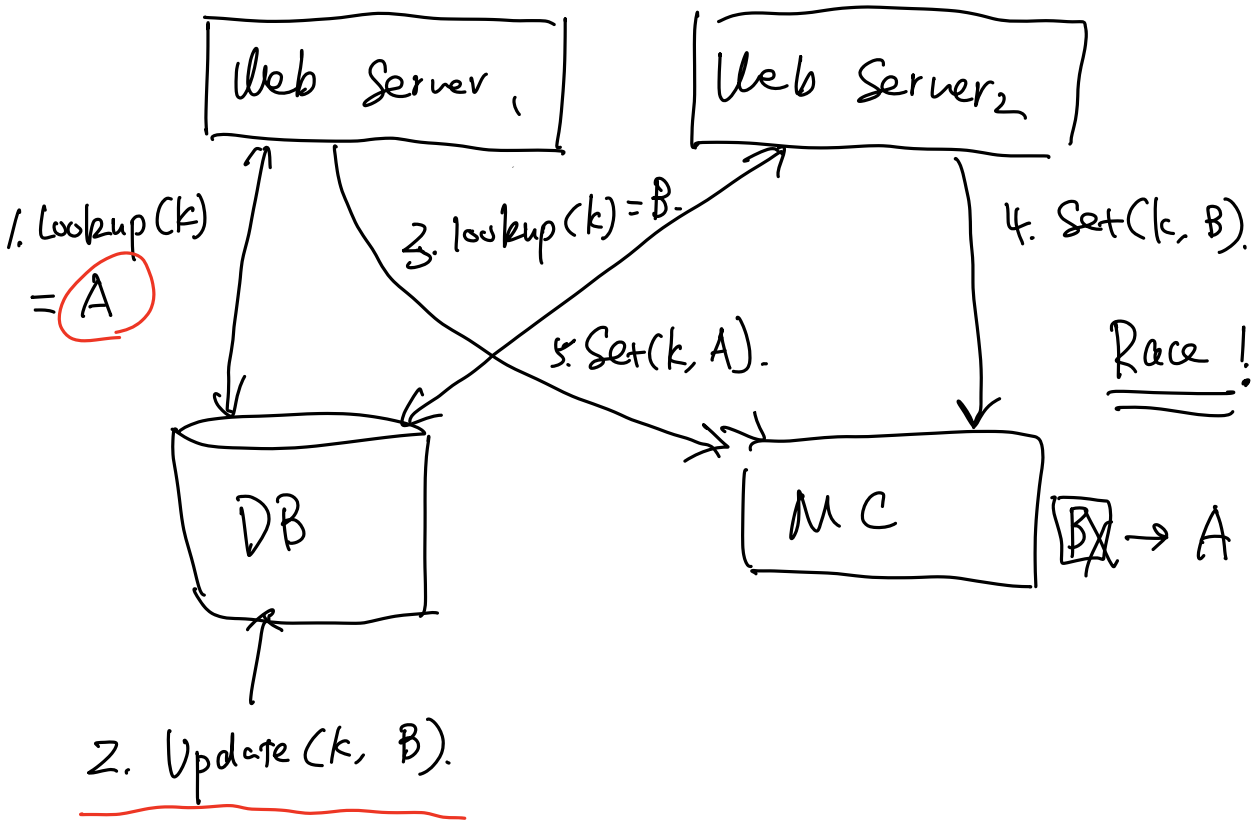
Problems with look-aside caching

Stale Sets

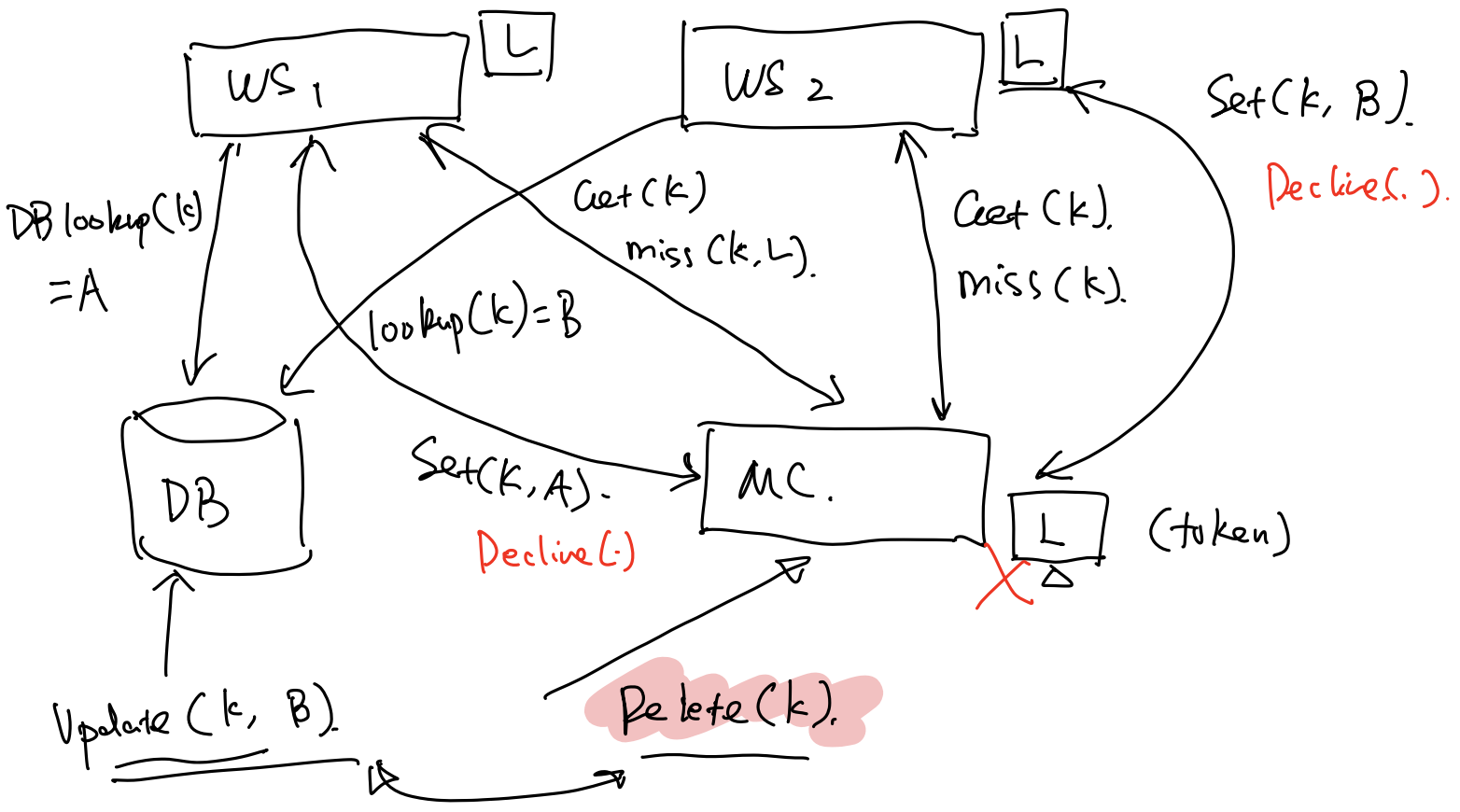


- Extend memcache protocol with "leases"
  - Return and attach a lease-id with every miss
  - Lease-id is invalidated inside server on a delete
  - Disallow set if the lease-id is invalid at the server

Issue ① Stale Sets.

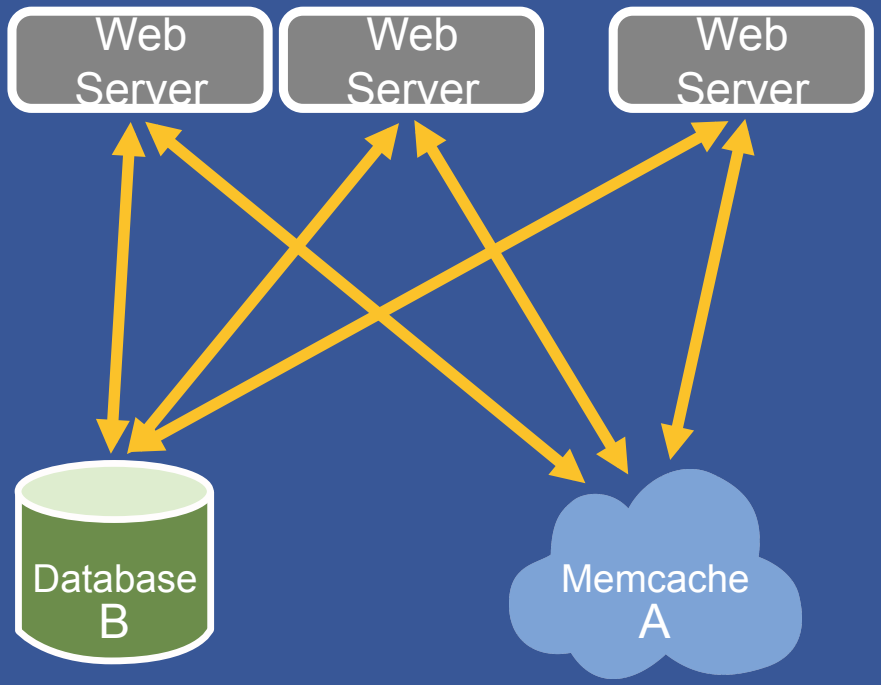


Fix: Leases. → best effort solution



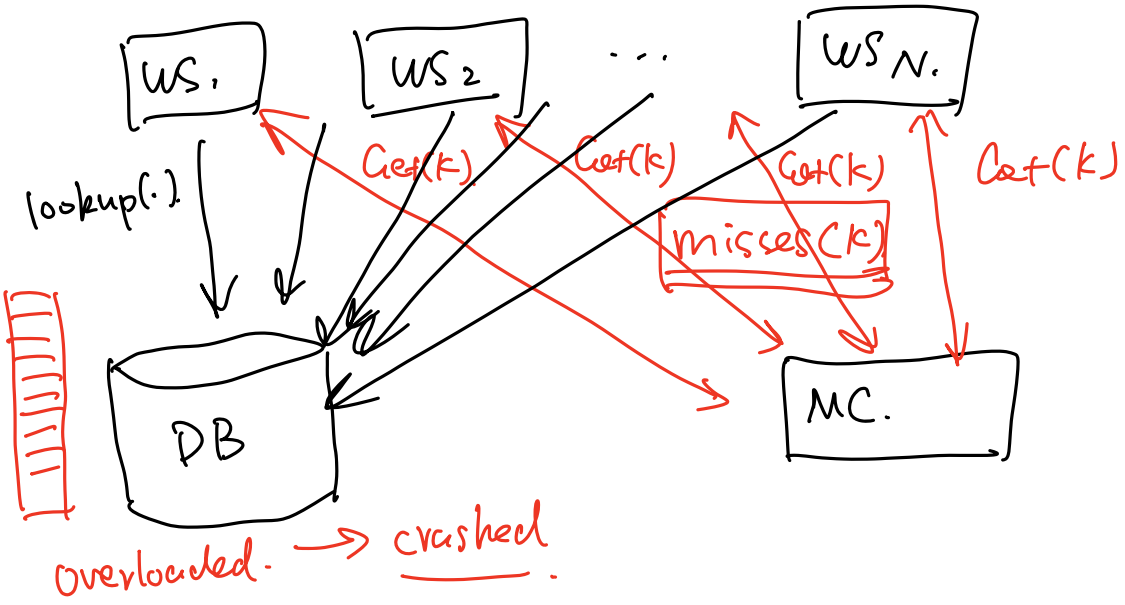
Problems with look-aside caching

Thundering Herds

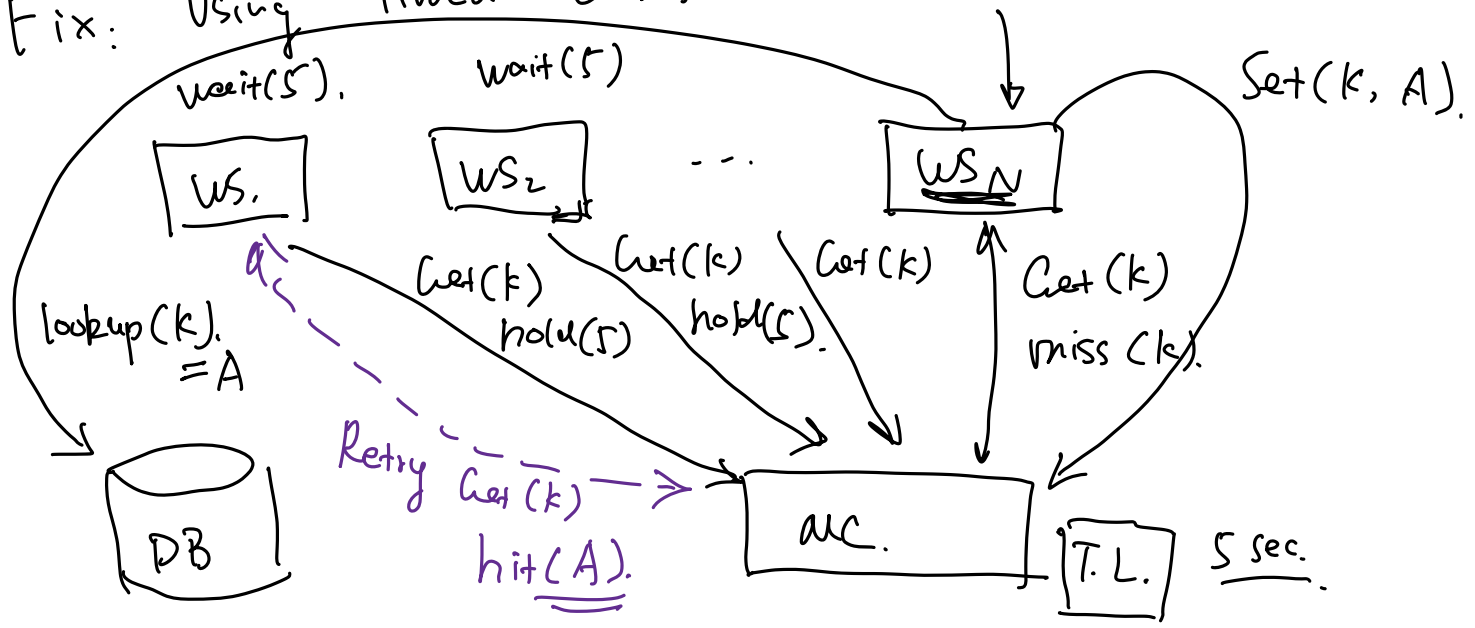


- Memcache server arbitrates access to database
  - Small extension to leases
- Clients given a choice of using a slightly stale value or waiting

Thundering Herds. (Read Storm).



Fix: Using timed leases.



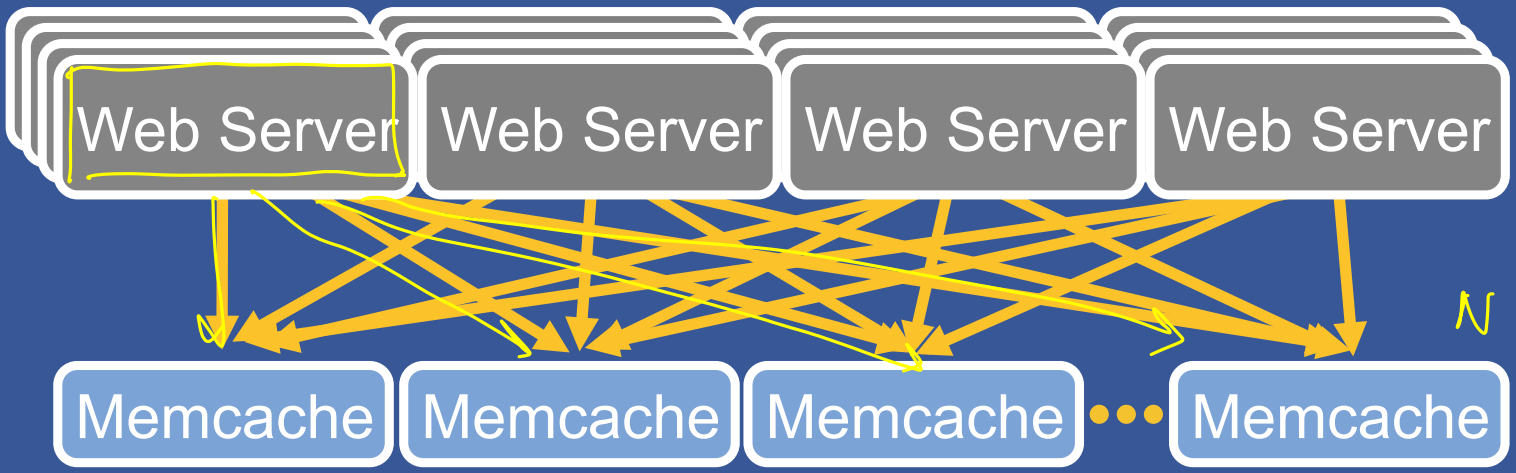
Similar → Exponential Backoff.

Scaling memcache in 4 “easy” steps

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

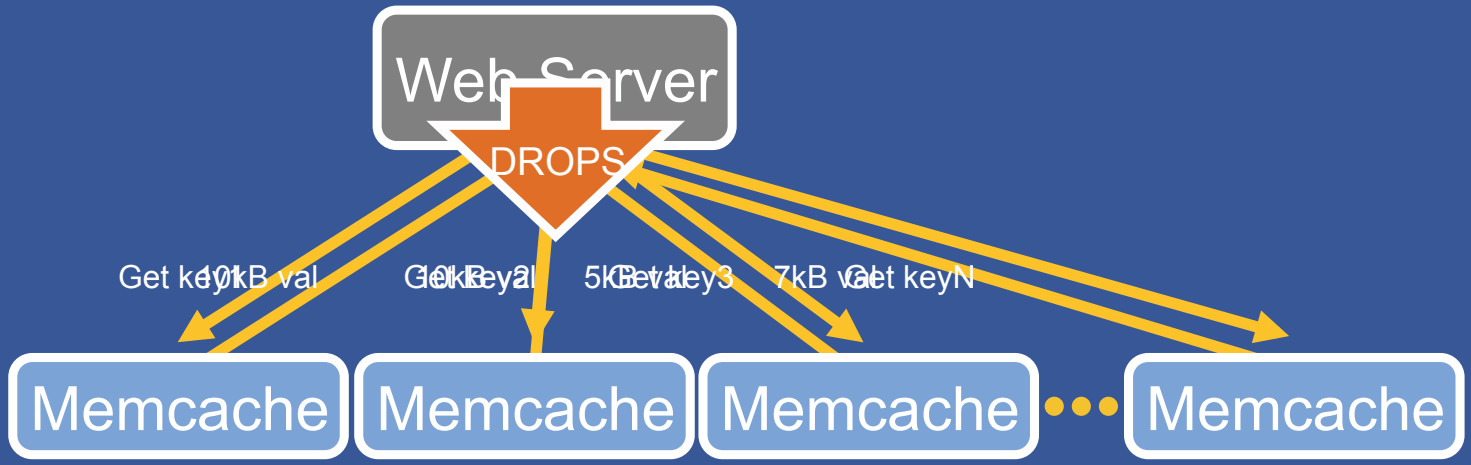
0	No memcache servers
1	A few memcache servers
2	Many memcache servers in one cluster
3	Many memcache servers in multiple clusters
4	Geographically distributed clusters

Need even more read capacity



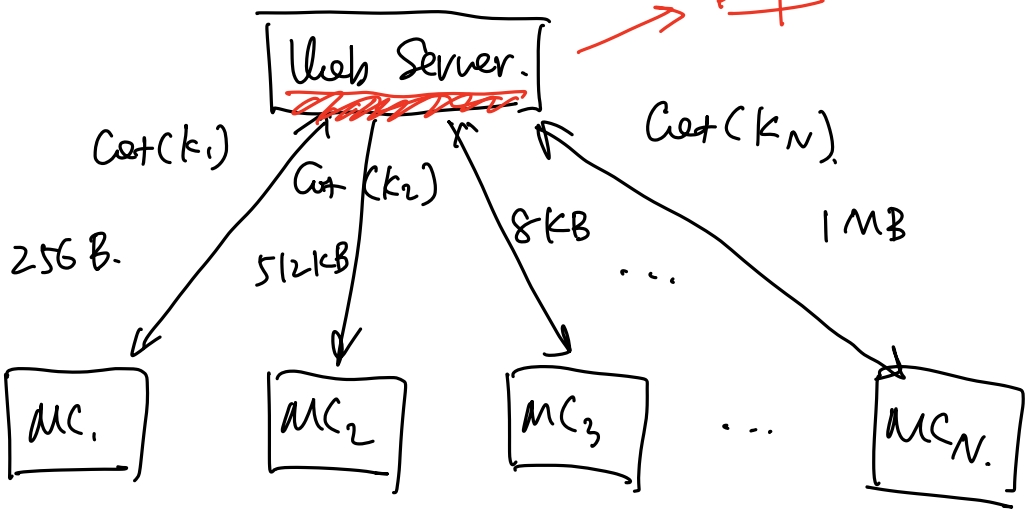
- Items are distributed across memcache servers by using consistent hashing on the key
  - Individual items are rarely accessed very frequently so over replication doesn't make sense
- All web servers talk to all memcache servers
  - Accessing 100s of memcache servers to process a user request is common

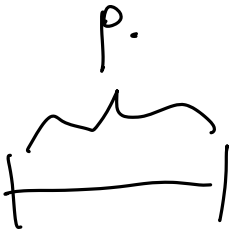
Incast congestion



- 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

Incast Congestion:

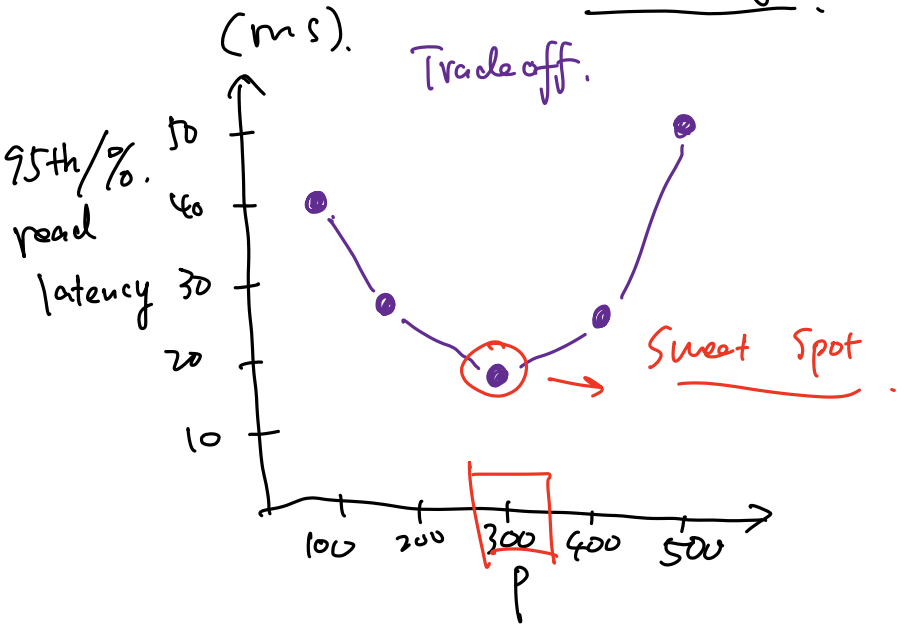




Throttling → sliding window.

P.

Performance Tuning.



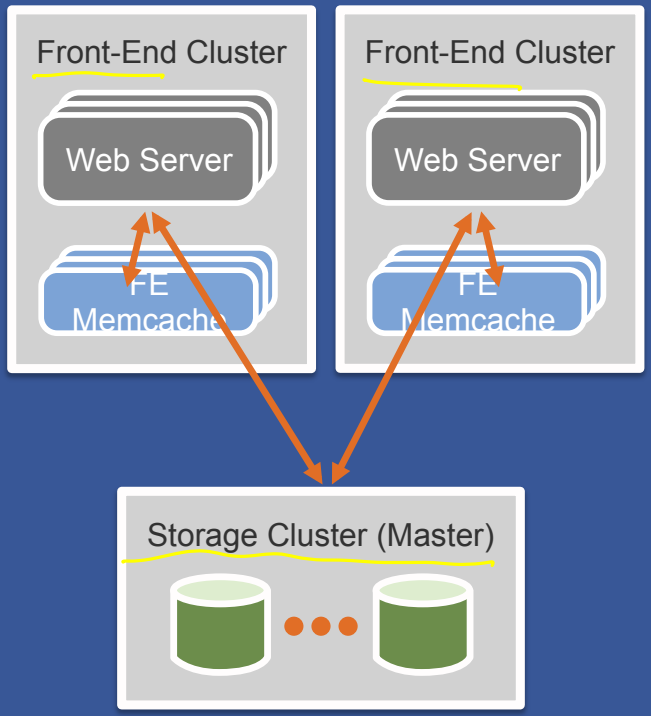
Scaling memcache in 4 “easy” steps

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

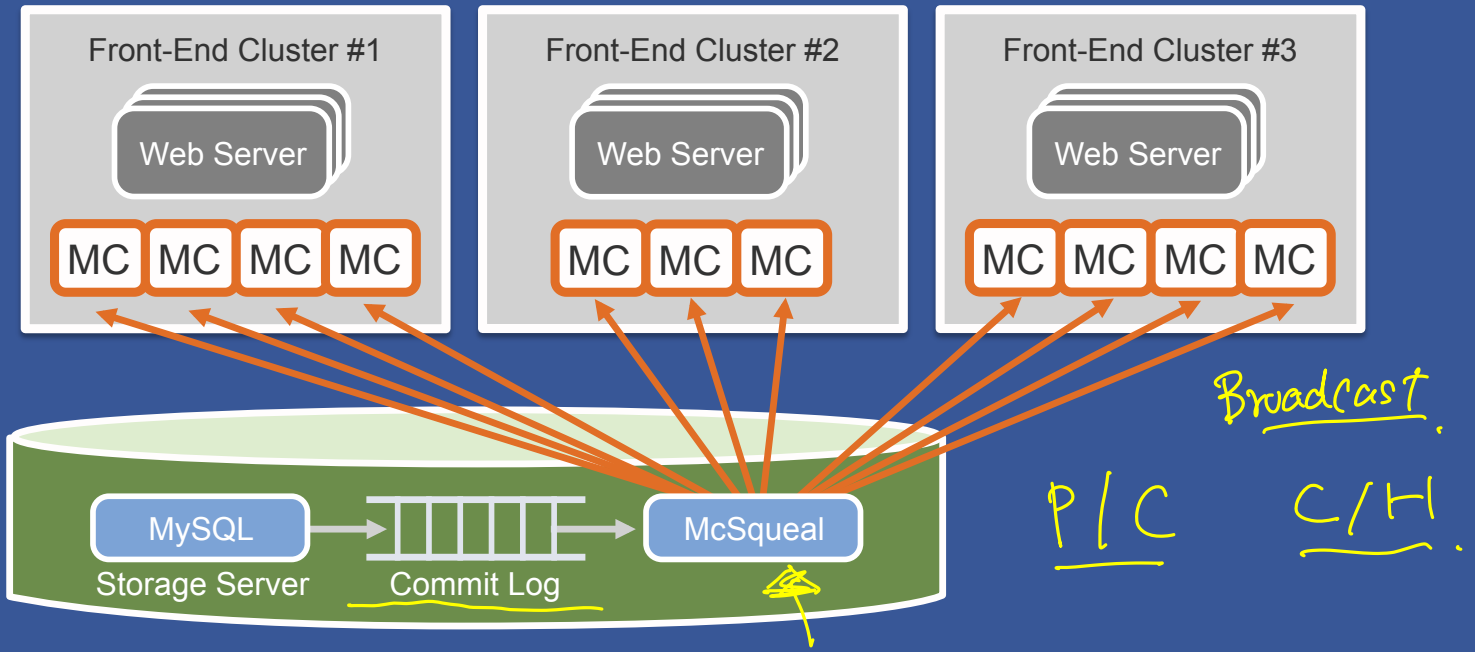
0	No memcache servers
1	A few memcache servers
2	Many memcache servers in one cluster
3	Many memcache servers in multiple clusters
4	Geographically distributed clusters

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



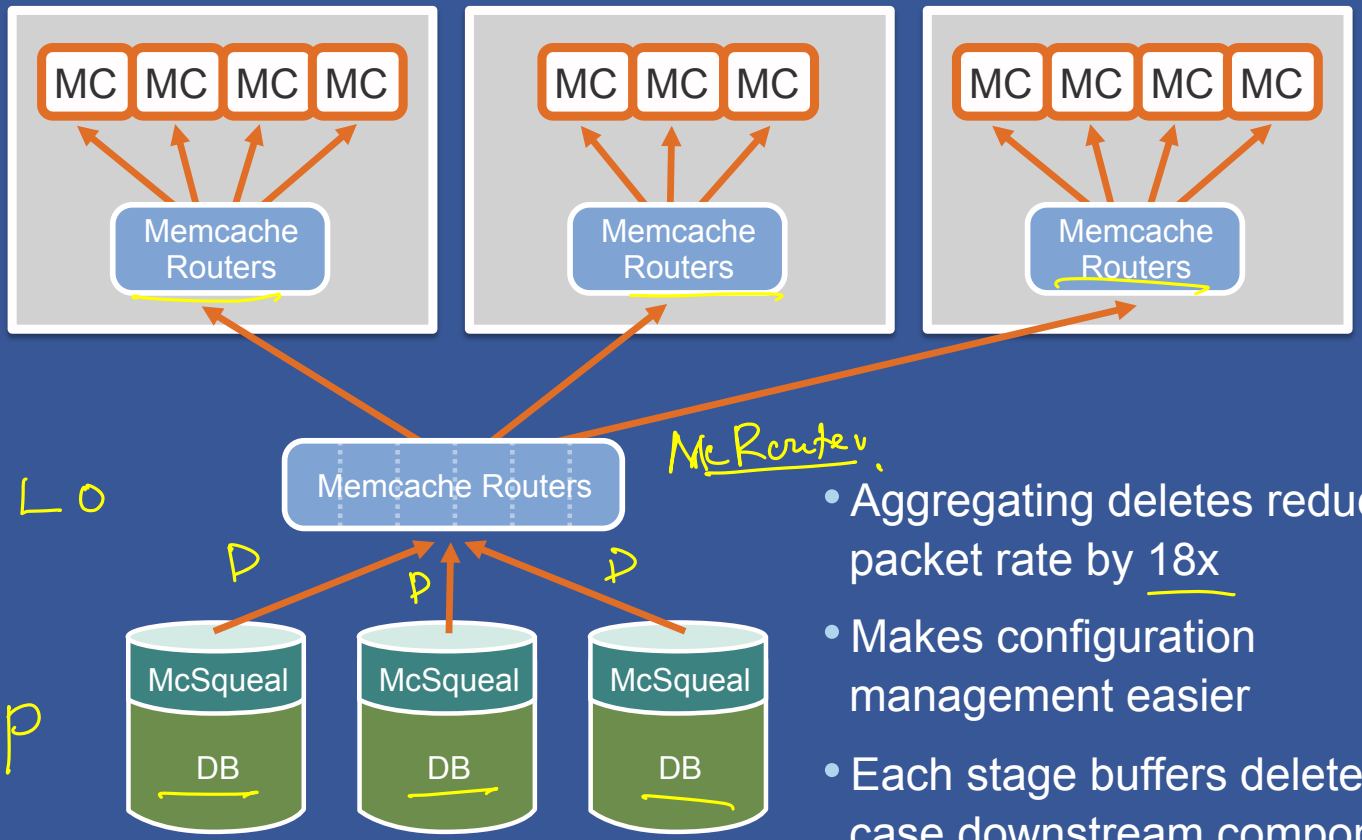
Databases invalidate caches



- 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



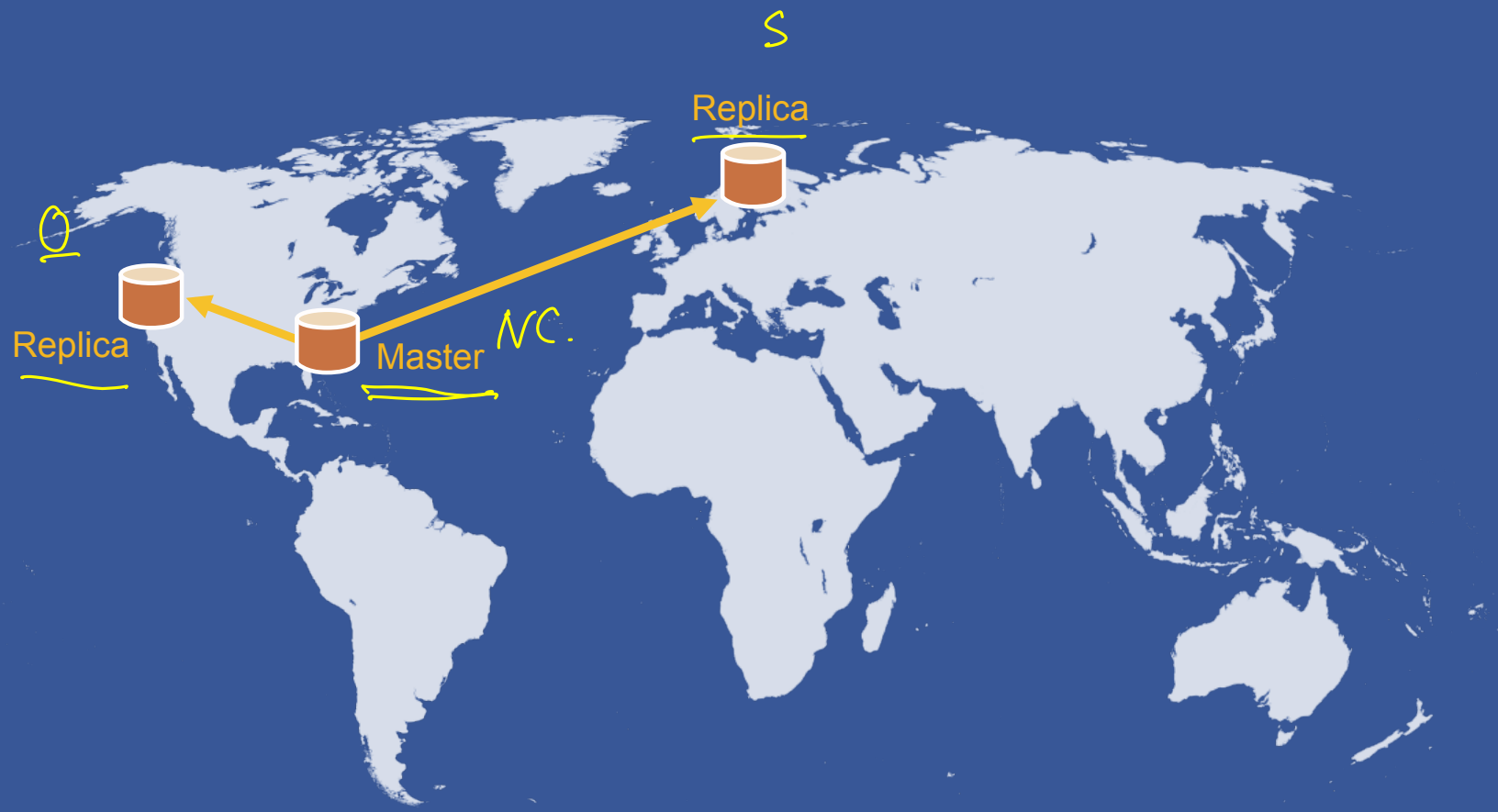
- Aggregating deletes reduces packet rate by 18x
- Makes configuration management easier
- Each stage buffers deletes in case downstream component is down

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

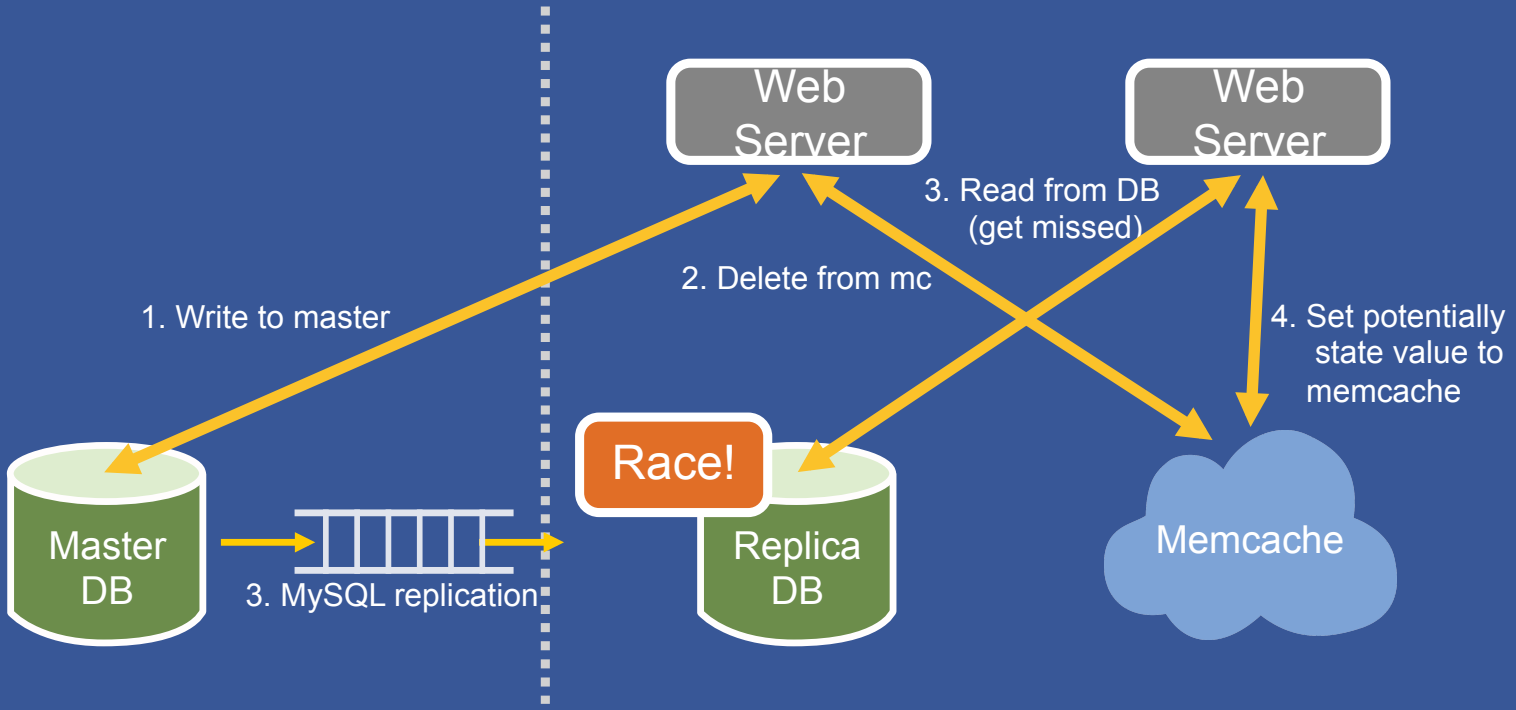
Geographically distributed clusters

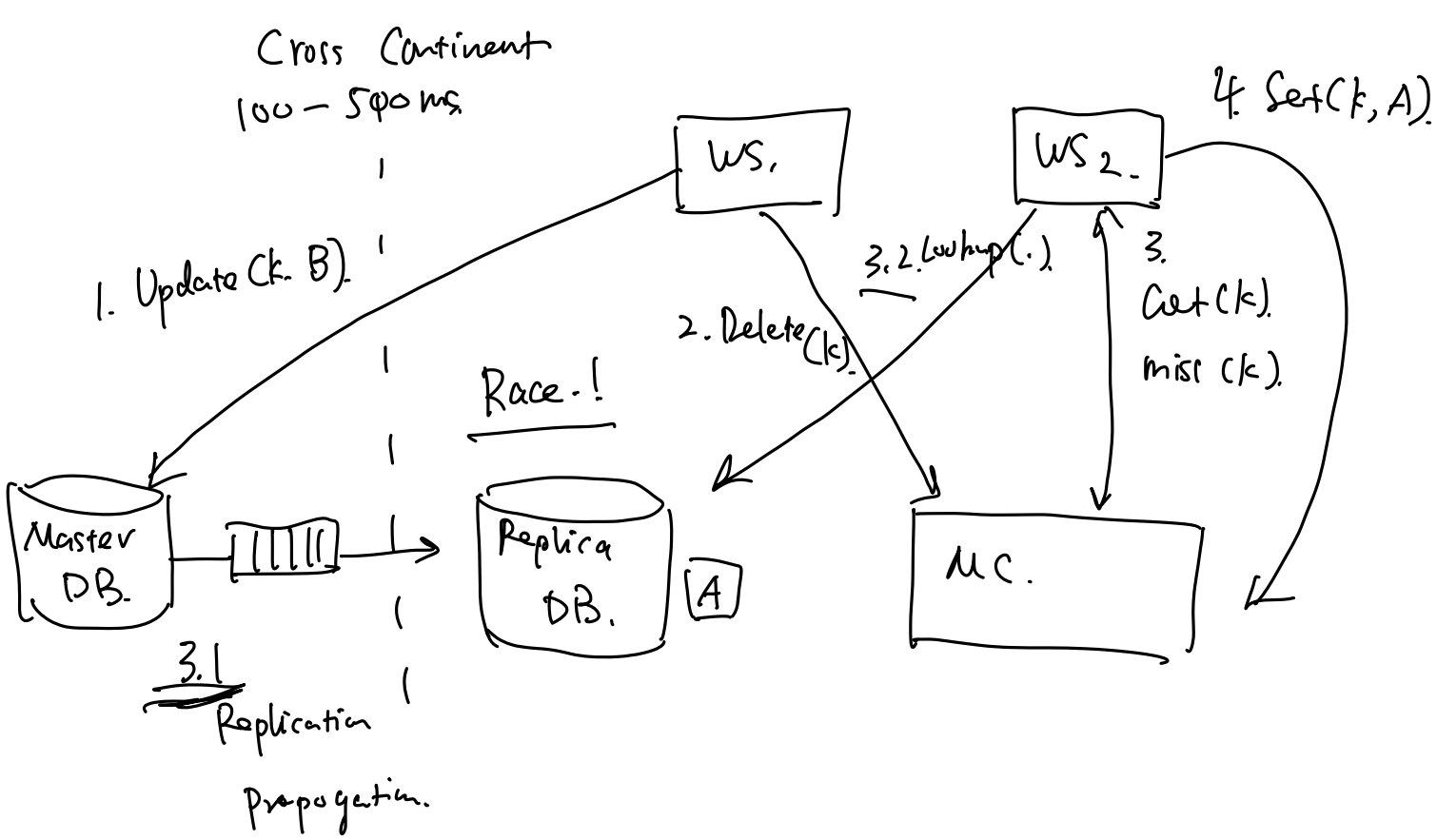


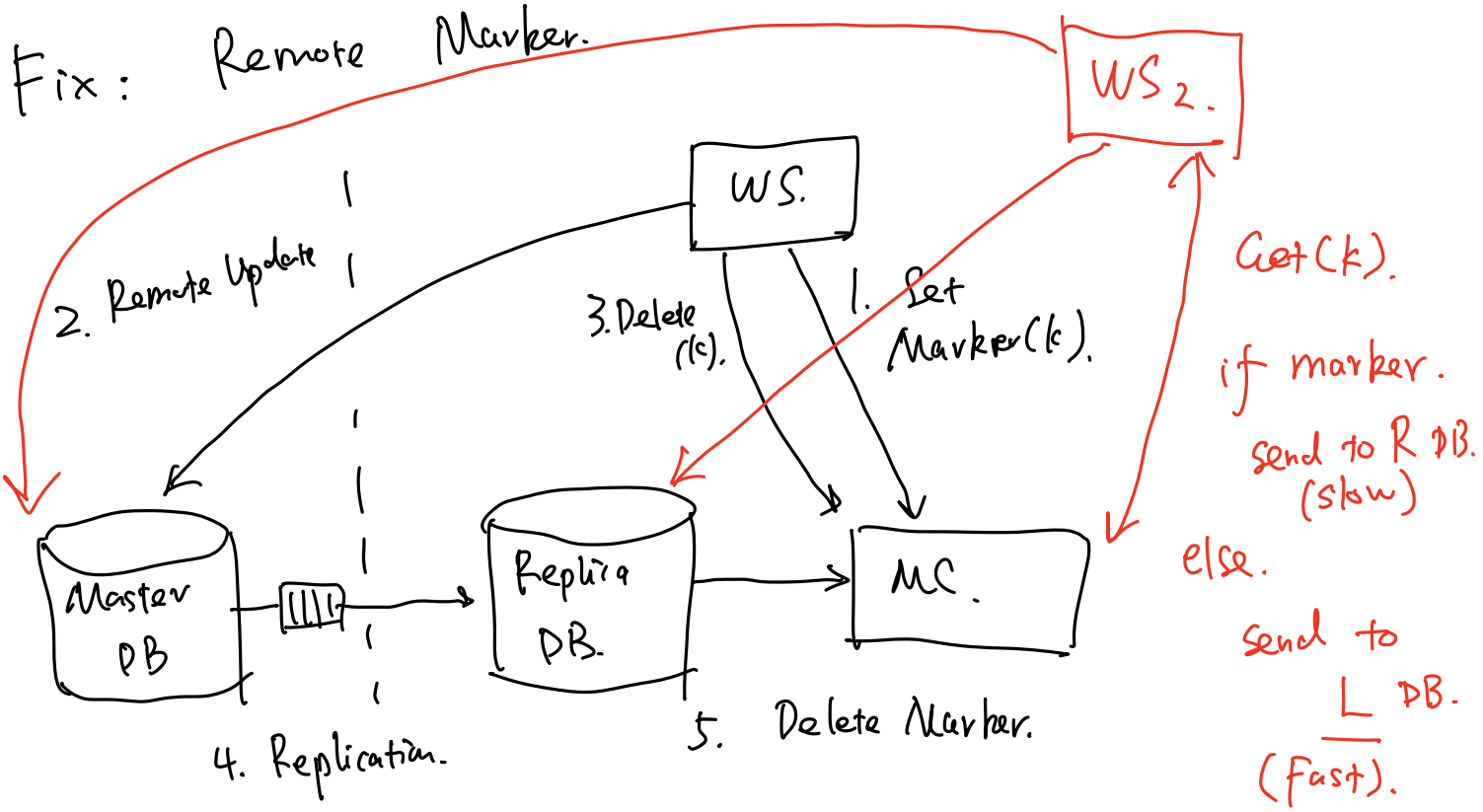
Writes in non-master

Database update directly in master

- Race between DB replication and subsequent DB read



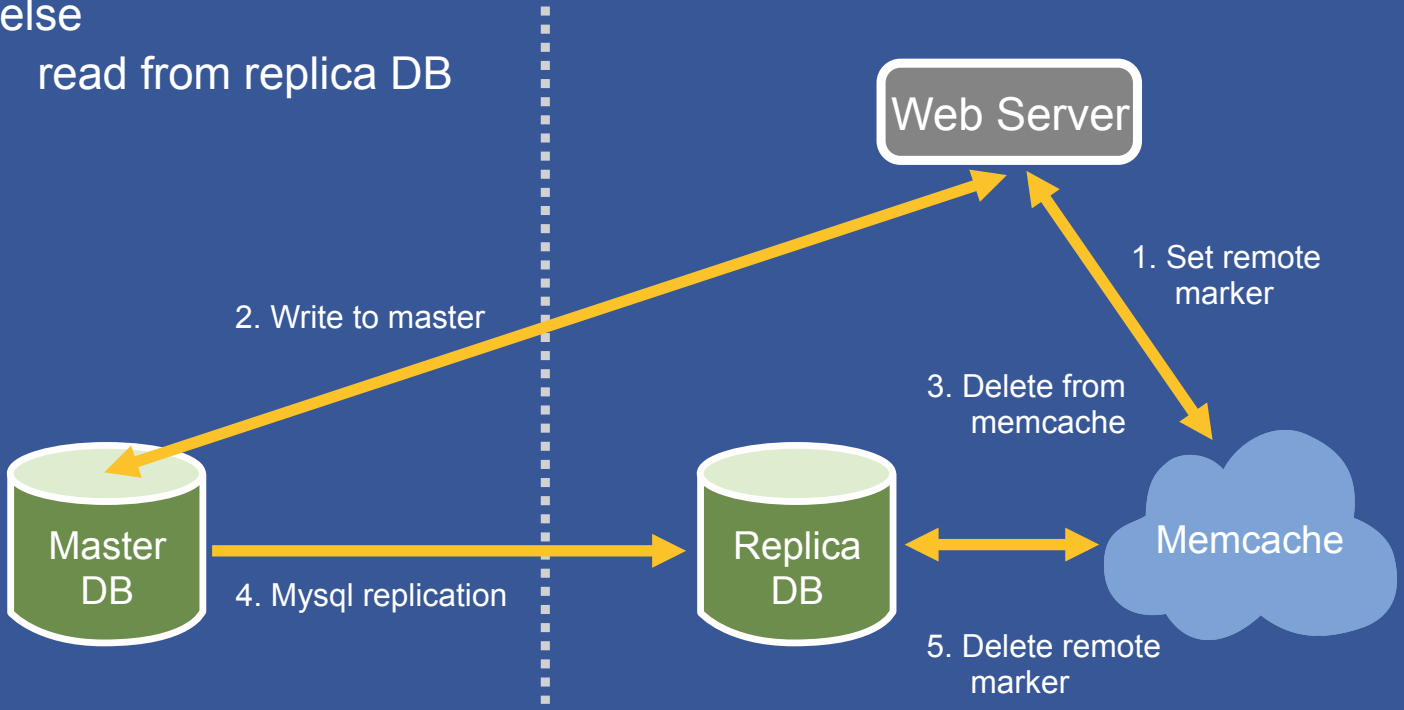

Cross Continent
100 – 500 ms
WS1
WS2
4. Set(k, A)
1. Update(k, B)
3.2 Lookup(.)
2. Delete(k)
3. Get(k)
miss(k)
Race!
Master DB
Replica DB
A
MC
3.1 Replication Propagation


Fix: Remote Marker
WS2
WS1
2. Remote Update
3. Delete(k)
1. Set Marker(k)
Get(k)
if marker
send to R DB (slow)
else
send to L DB (Fast)
Master DB
Replica DB
MC
4. Replication
5. Delete Marker

Remote markers

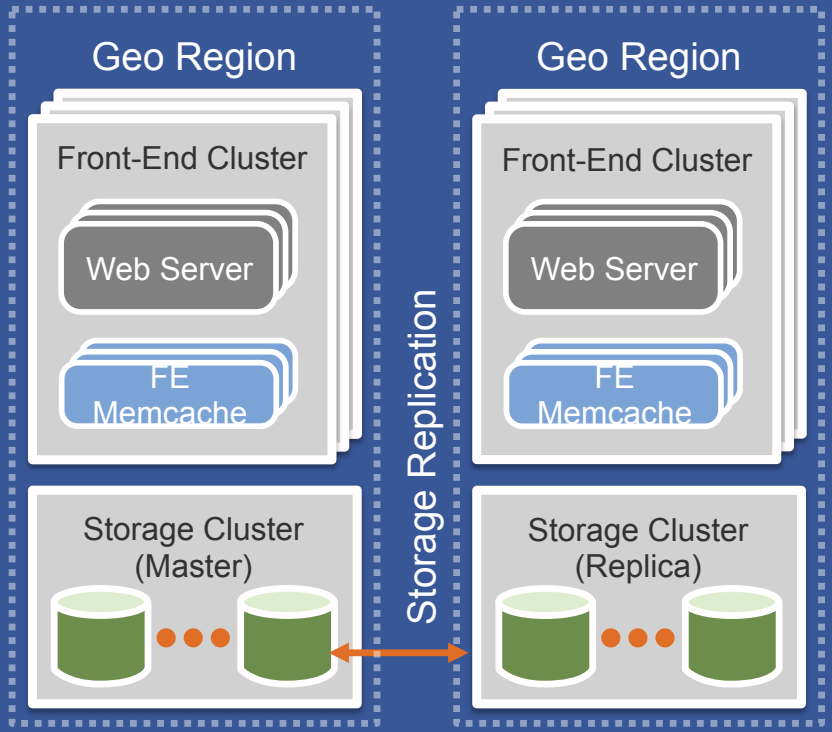
Set a special flag that indicates whether a race is likely

Read miss path:
If marker set
  read from master DB
else
  read from replica DB



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