Scaling Memcache at Facebook

Ewa Pawłowska

Scaling Memcache at Facebook

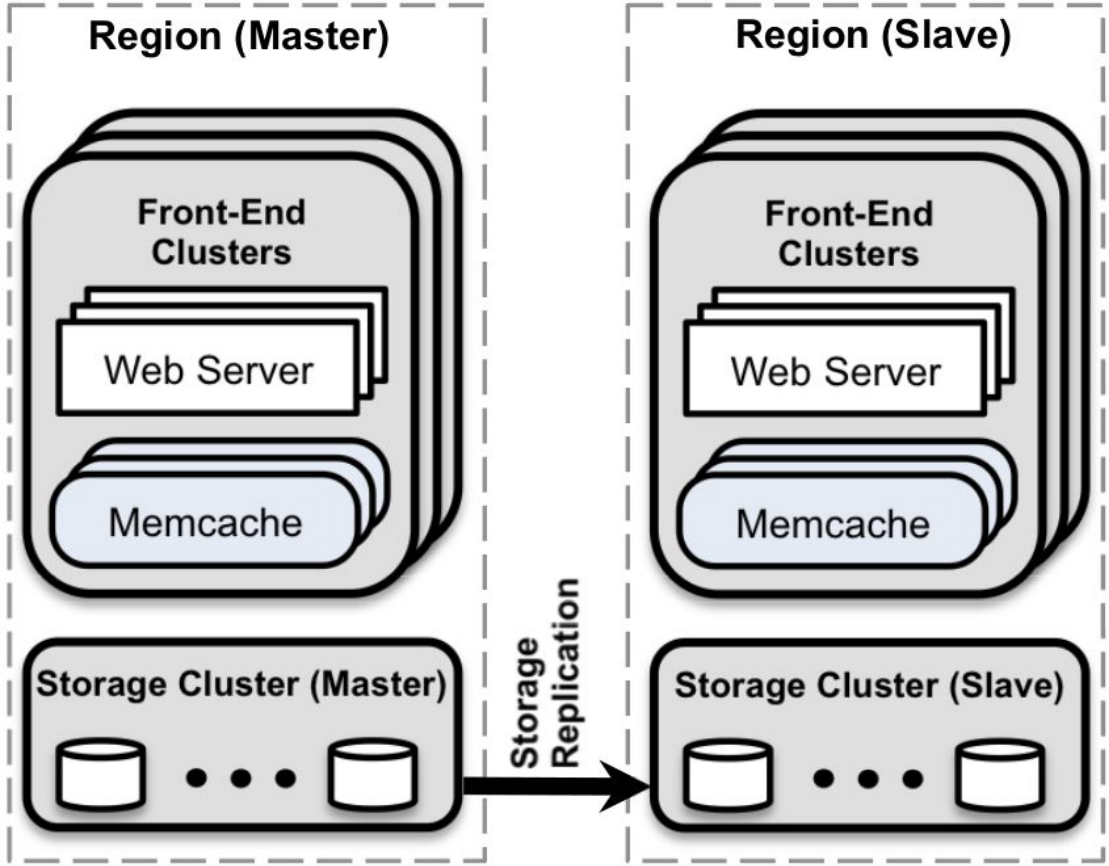
Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li,
Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung,
Venkateshwaran Venkataramani

{rajeshn,hans}@fb.com, {sgrimm, marc}@facebook.com, {herman, hcli, rm, mpal, dpeek, ps, dstaff, ttung, veeve}@fb.com

Facebook Inc.

Introduction

1. Facebook
2. Memcache
3. Scaling Memcache
   - In a Cluster
   - In a Region
   - Across Regions
4. Conclusion



Facebook - Infrastructure Requirements

Near real-time communication

Access and update very popular shared content

Scale to process millions of user requests per second

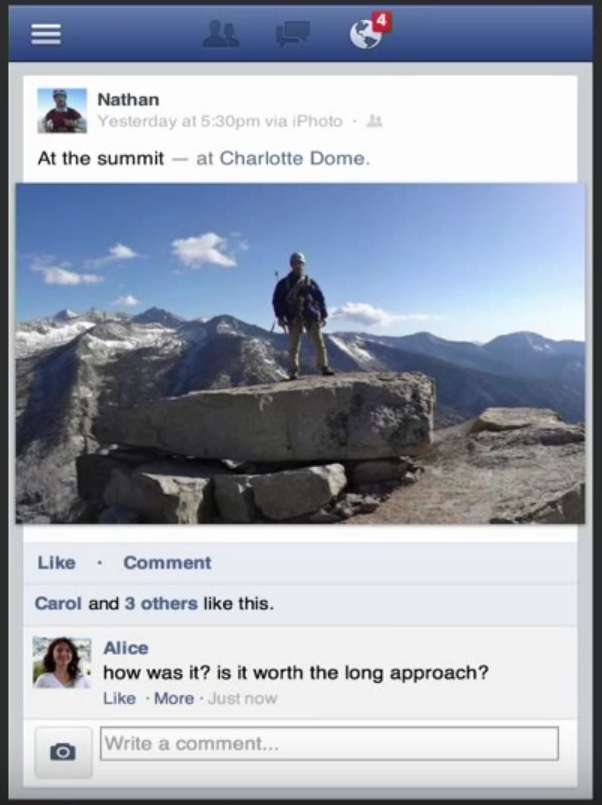
Facebook - Design Requirements

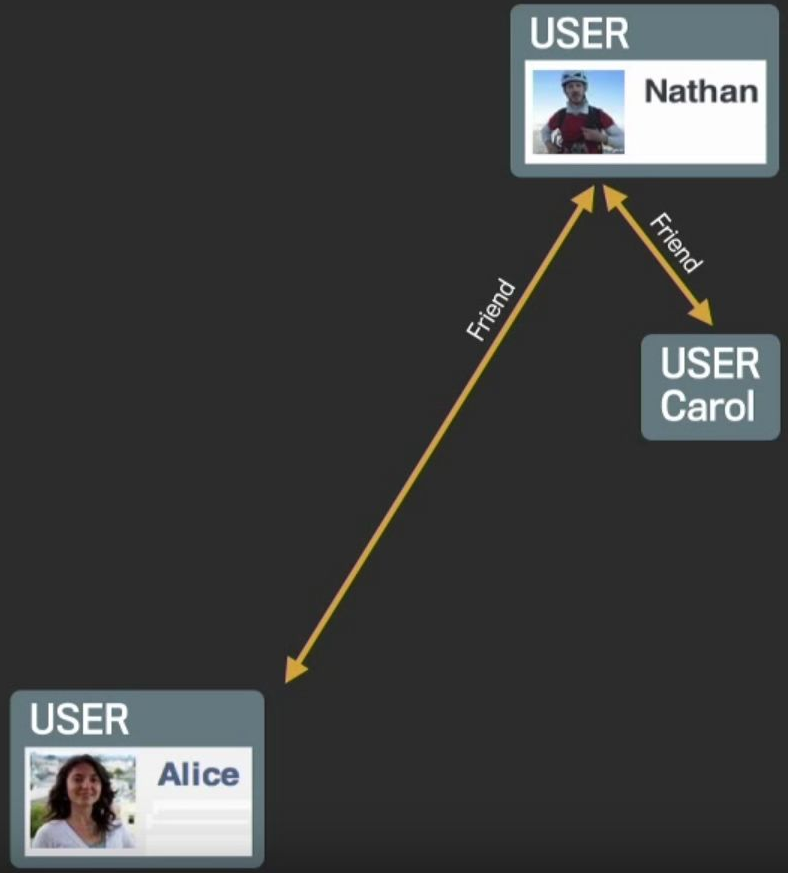
Support a very heavy read load

- Over 1 billion reads / second
- Two orders of magnitude more reads than writes
- Insulate backend service from high read rates

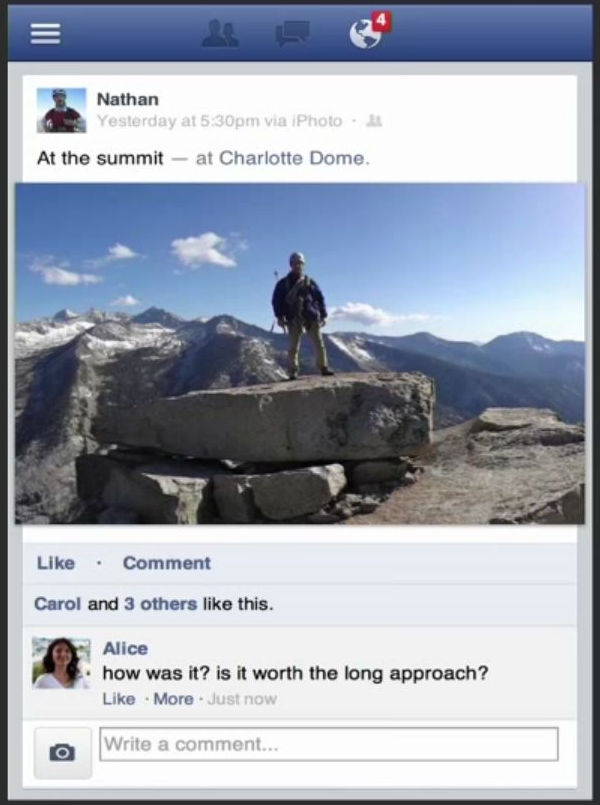
Geographically Distributed

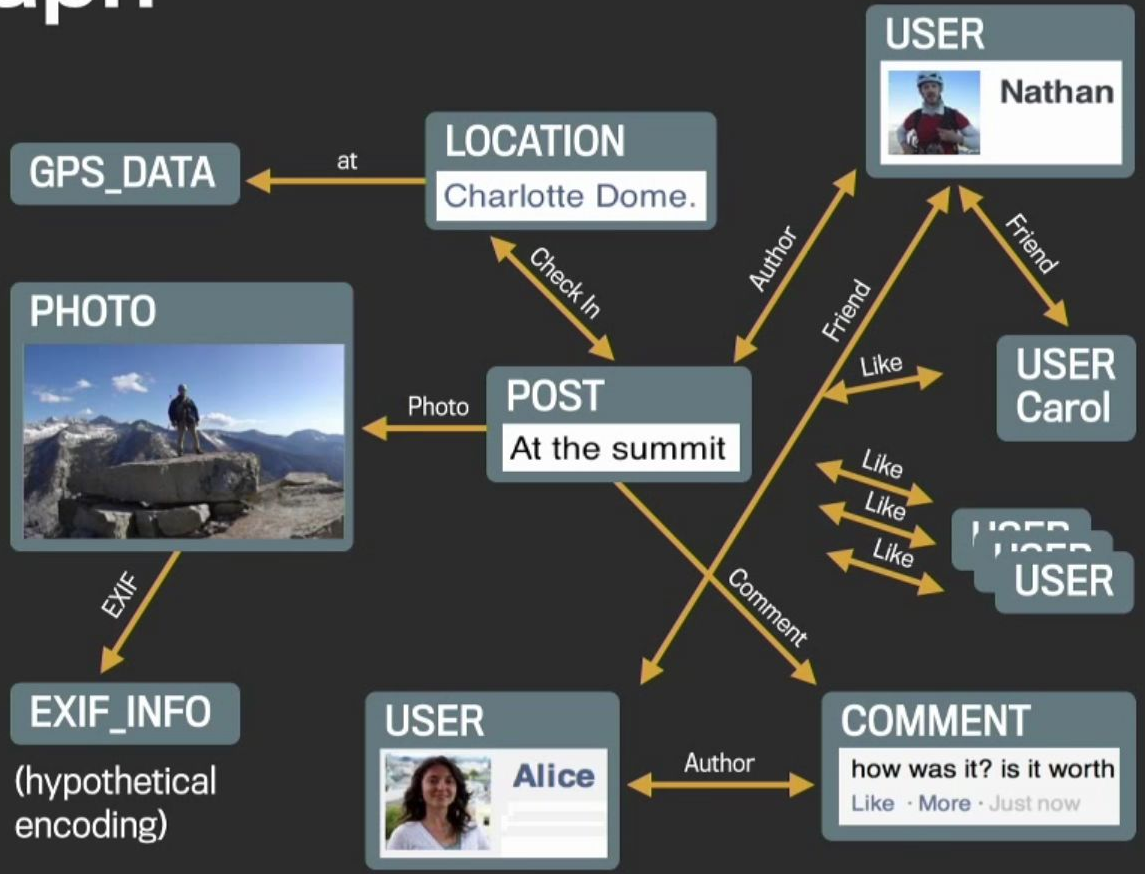
The Social Graph



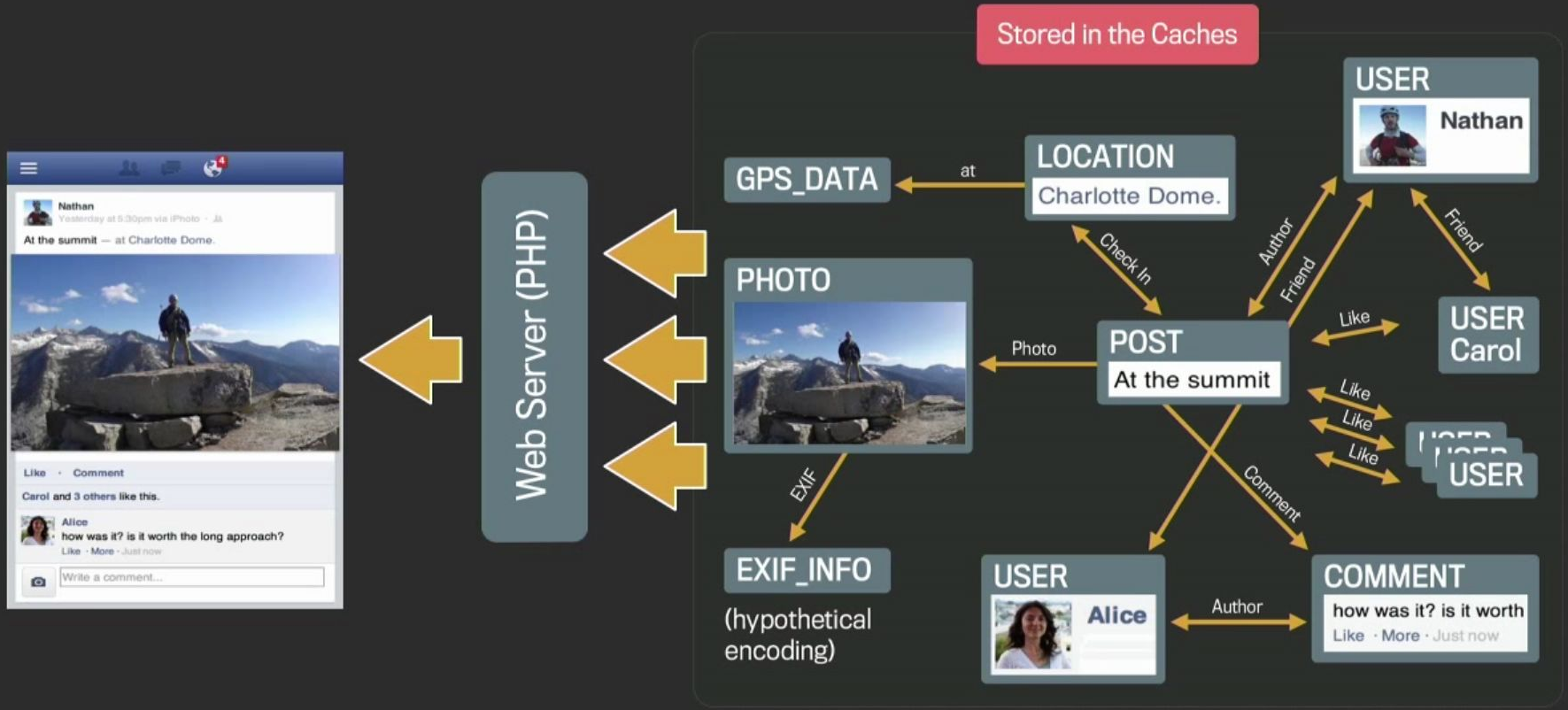


The Social Graph





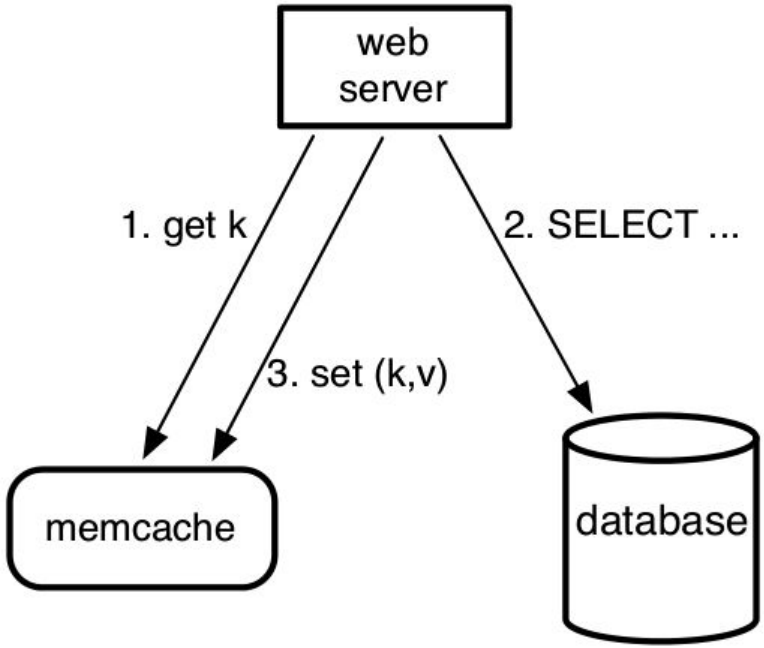
Dynamically Rendering the Graph



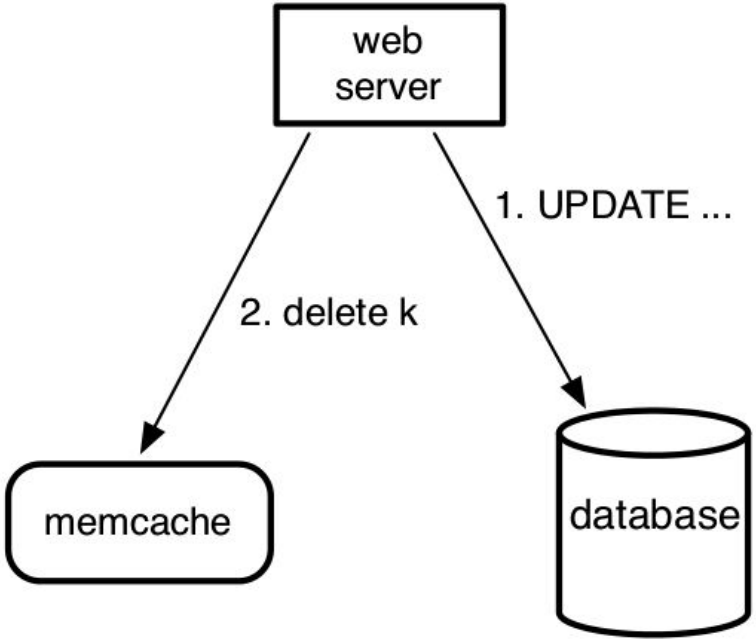
Memcache

- in memory cash
- hash table
- operations: set, get delete
- open source

Reads



Updates



- prefer deletes to sets

Scaling Memcache

1. In a cluster

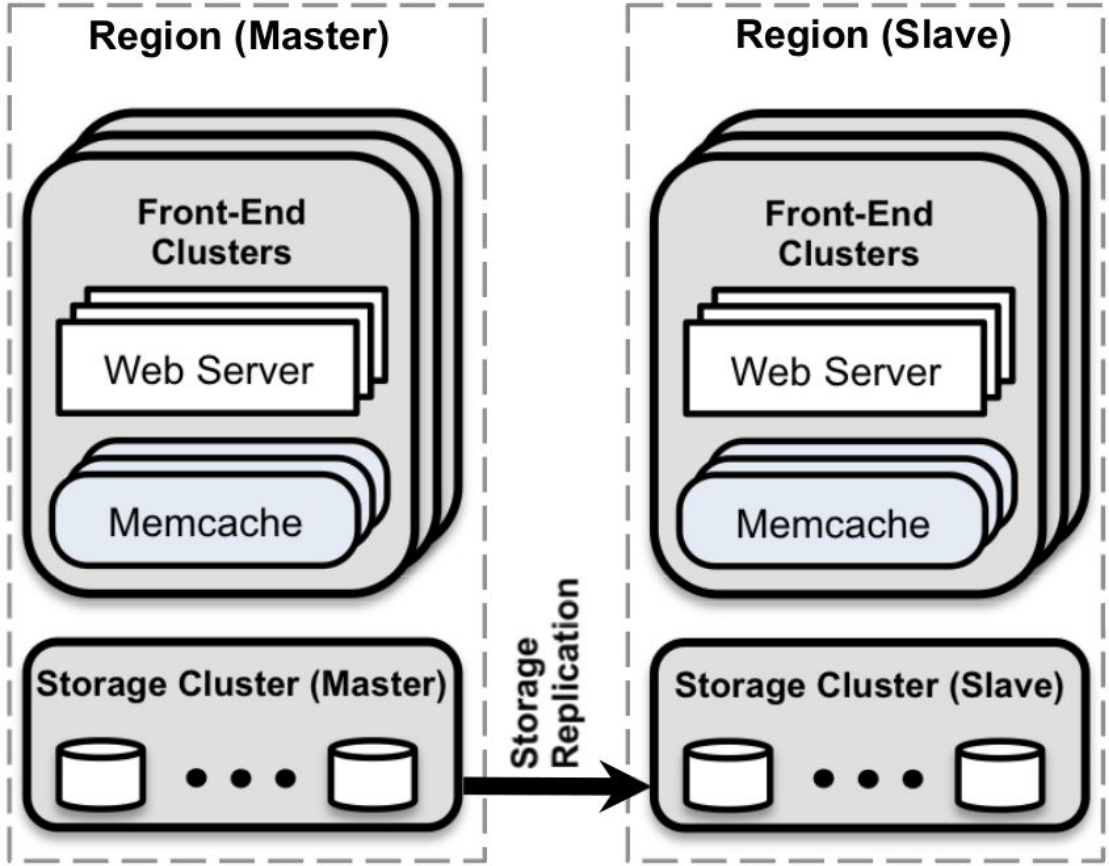
   - Read heavy workload
   - Wide fanout

2. In a region - multiple clusters

   - Controlling data replication
   - Data consistency

3. Across regions

   - Data consistency



In a Cluster

~1000 servers within a cluster

~100 memcache servers

Challenges:

- Read heavy workload
- Wide fanout

Solutions:

- Reducing latency of fetching cached data
- Reducing load due to cache miss

In a Cluster - Reducing Latency

Memcache's response - critical factor

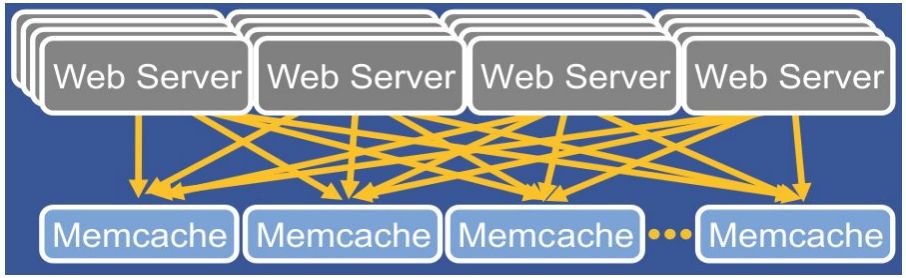
~100 memcache servers in a cluster

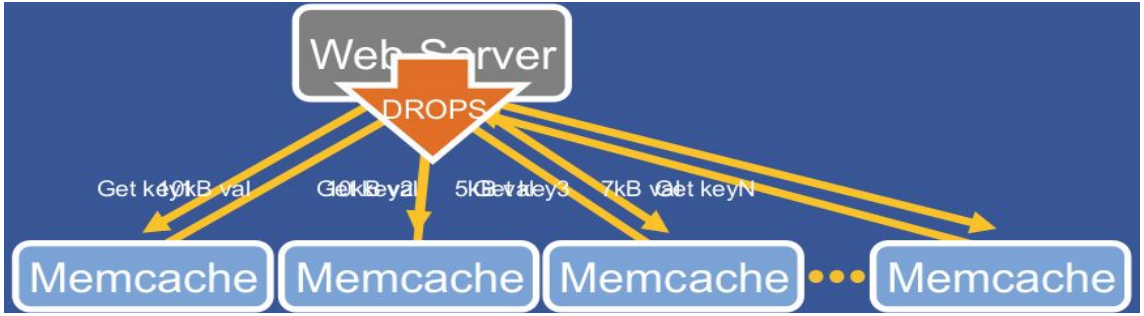
consistent hashing for items distribution

all-to-all communication

Problems:

- crowd in network
- incast congestion





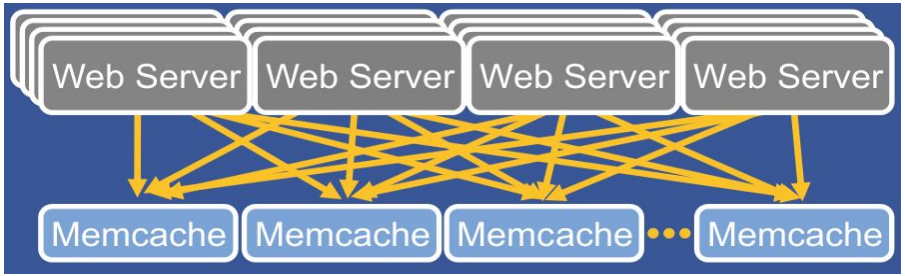
Client-Server Communication

Stateless client

- on each web server
- maintain map of all available servers
- serialization, compression, request routing, error handling, request batching

-> simplified memcached

-> simplified deployment

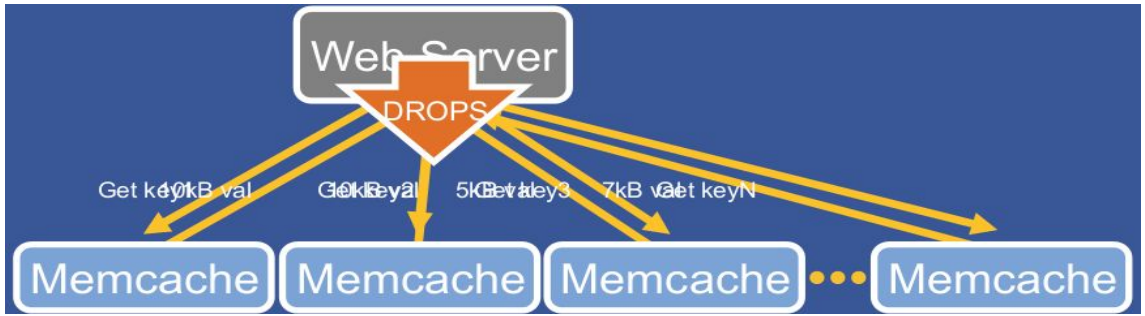


Sliding window mechanism

Control number of requests

Larger windows -> more incast congestion

Smaller windows -> more trips to the network



In a Cluster - Reducing Load

Reduce frequency of fetching data along more expensive paths

Leases

- Minimize application’s wait time in certain usecases

Leases - Stale Sets

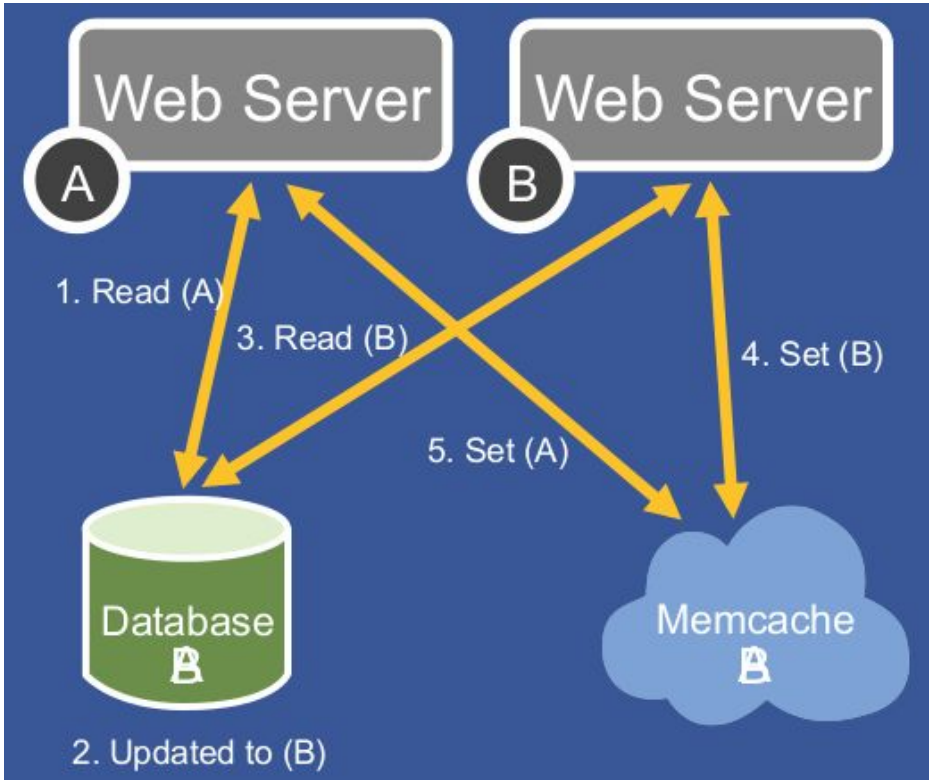
DB and Memcache inconsistency

Trestle: read and set from 2 web servers

Arbitrate concurrent writes

Solution:

- Lease token



Leases - Thundering Herds

Accessing popular content

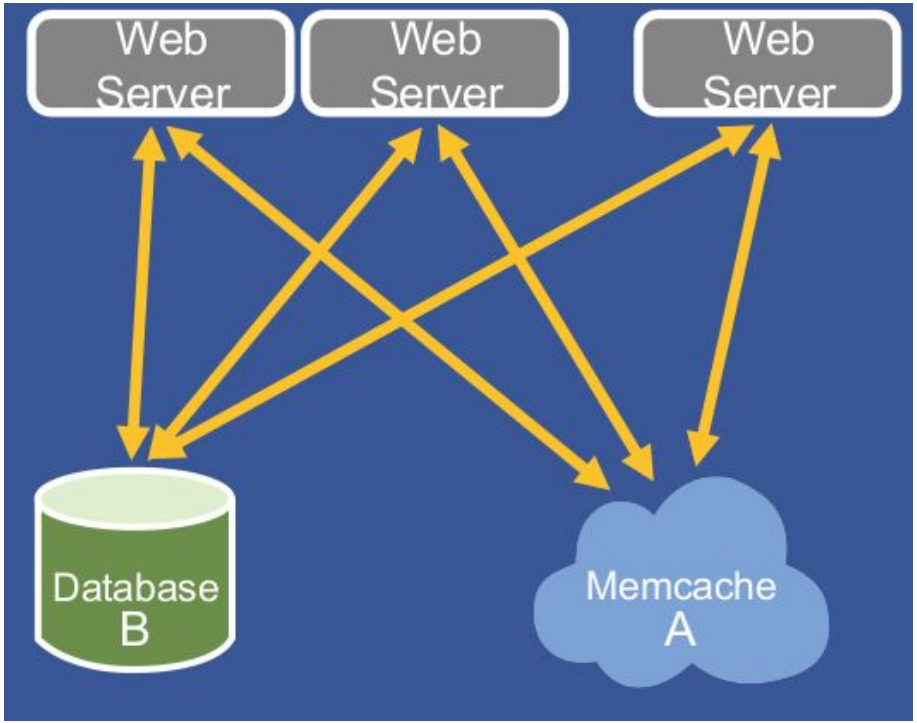
Value in database was updated

- > Value not in cache

Simultaneous access to database

Solution:

- Wait or take stale value



In a Region

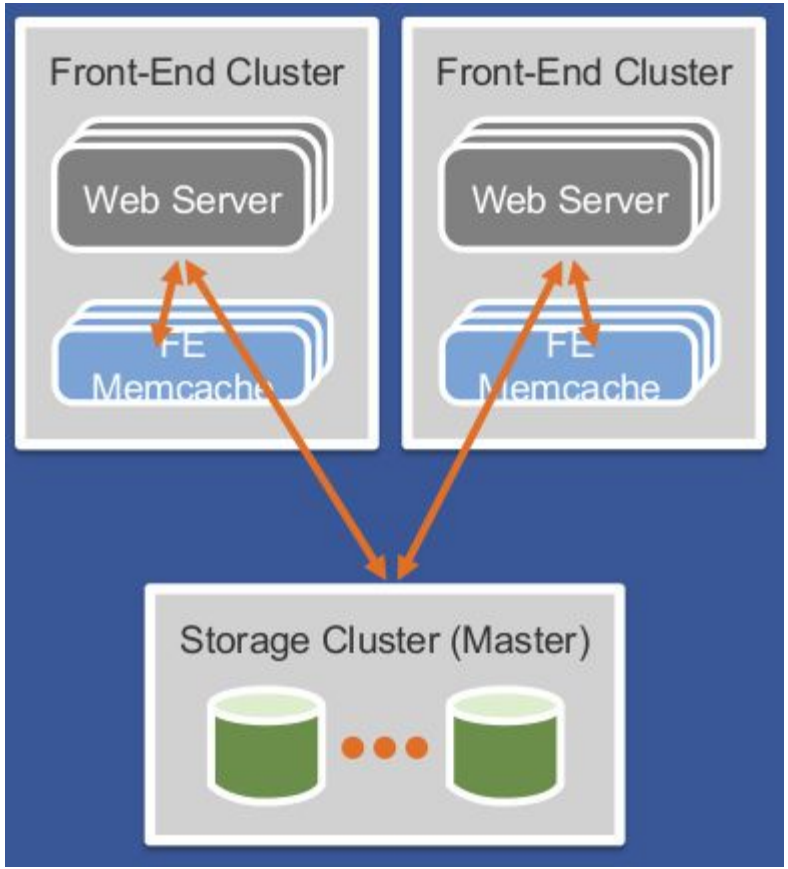
Multiple clusters

Challenges:

- Data consistency
- Controlling data replication

Solutions:

- Regional Invalidation
- Cold Cluster Warmup



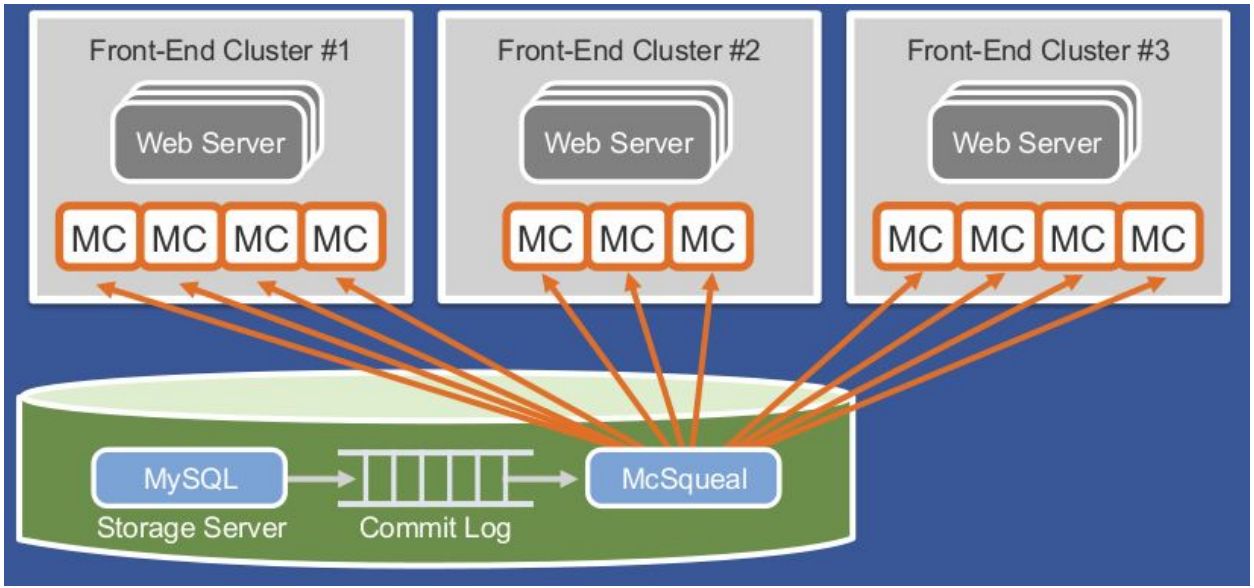
Regional Invalidations

System that looks at all database transactions that have been commited

Extract items that need to be invalidated

Broadcast invalidation

Batching deletes



Across Regions

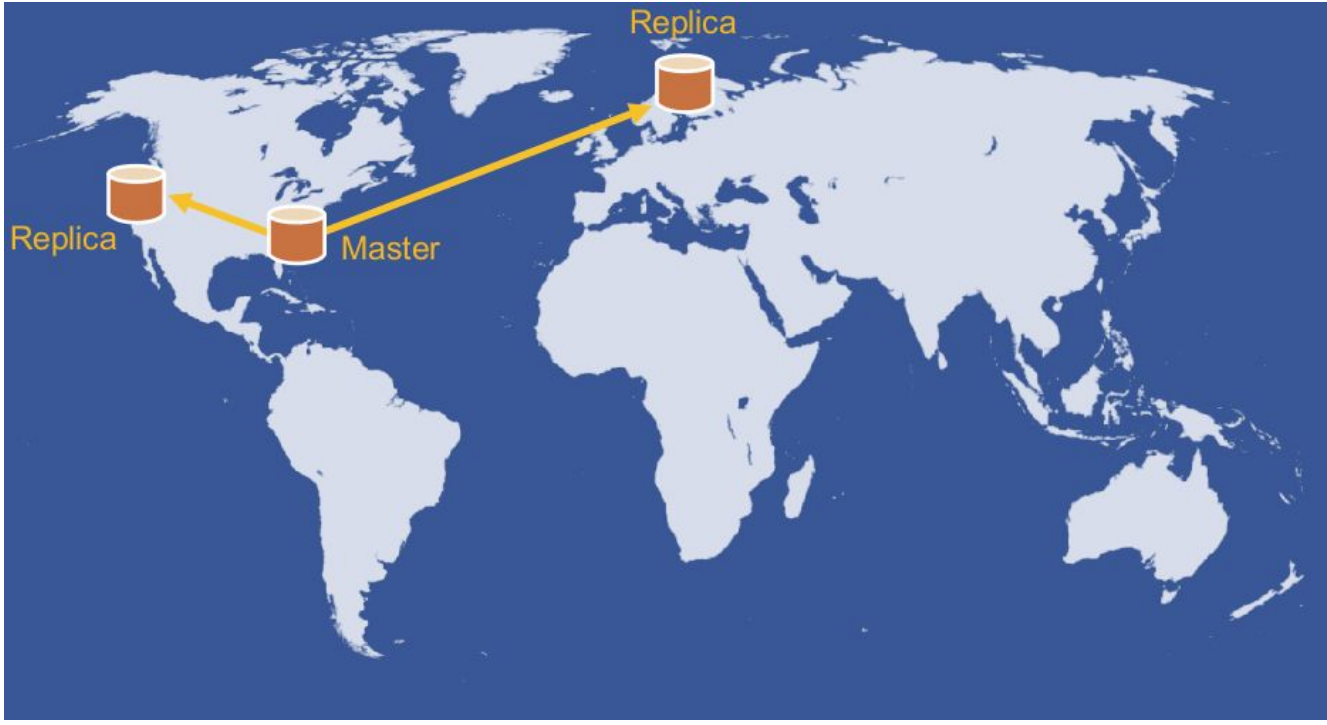
Single Master, few Replicas

Challenge

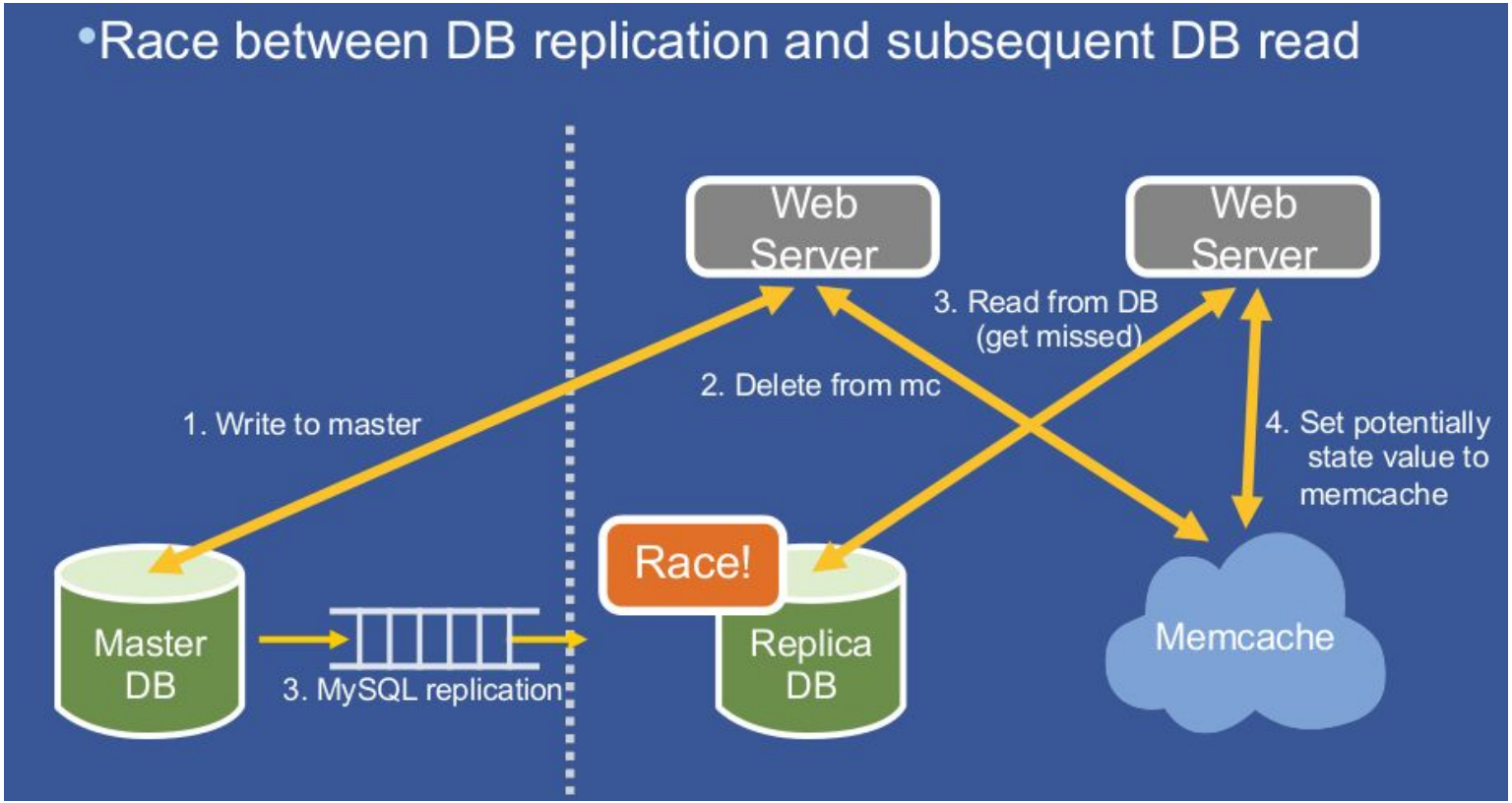
- Consistency

Solution

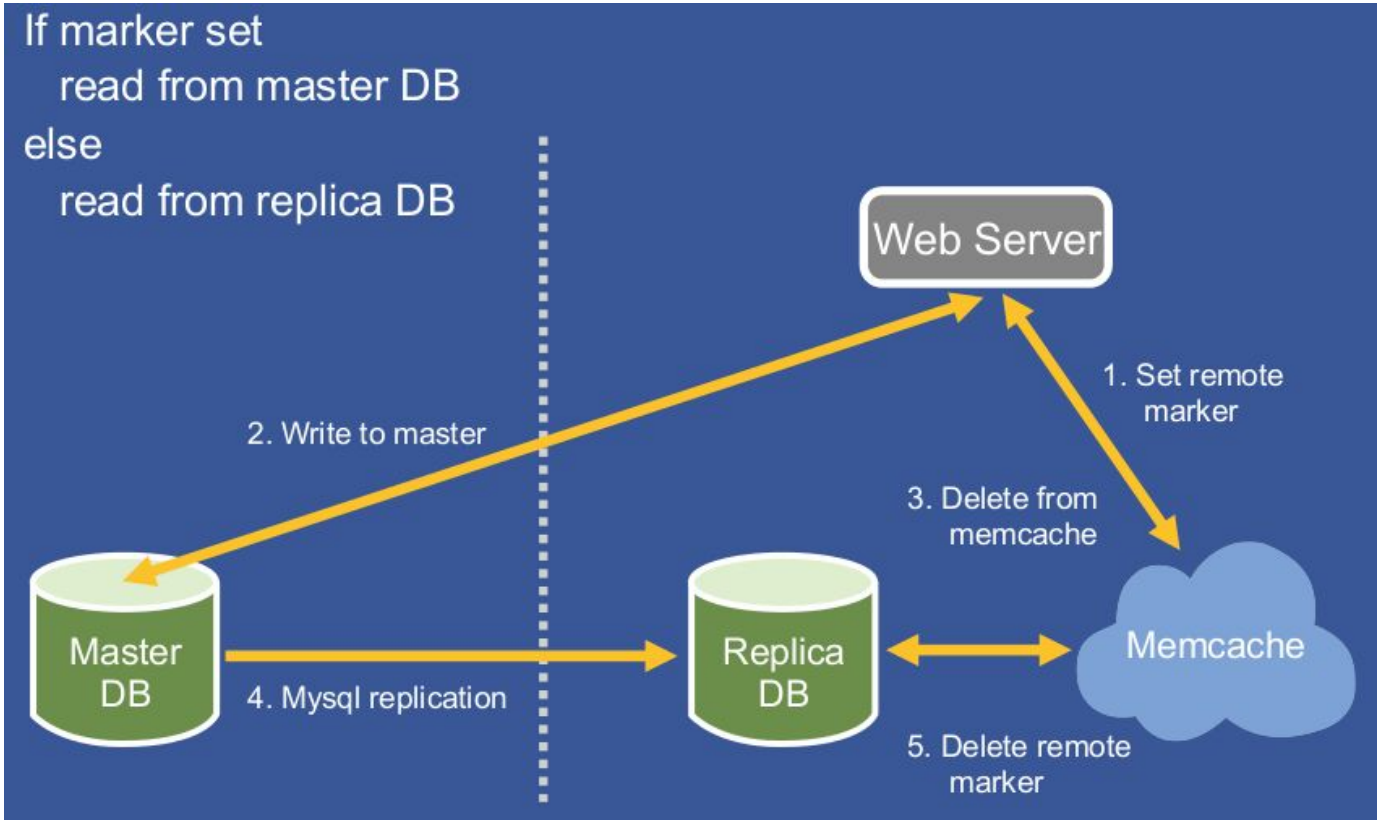
- Remote markers



Remote Markers



Remote Markers



Conclusion

1. Separating cache and persistent storage systems allows to independently scale them
2. Push complexity into the client whenever possible
3. Features that improve monitoring, debugging and operational efficiency are as important as performance

Thank you!