CS 582: Distributed Systems

Scaling Memcache at Facebook

Dr. Zafar Ayyub Qazi

Fall 2024

ACK: Slides use some material from Rajesh Nishtala (Facebook) & Robert Morris (MIT)

Specific learning outcomes

By the end of today's lecture, you should be able to:

- ❑ Analyze the architectural decisions and mechanisms that enabled Facebook to scale their Memcache system to handle billions of requests.
- ❑ Analyze the causes of the thundering herd problem in distributed caching systems and evaluate Facebook's mitigation strategies.
- ❑ Analyze how Facebook addresses the stale set
- ❑ Analyze the fault tolerance mechanisms implemented in Facebook's Memcache system and evaluate their effectiveness in handling failure scenarios.
- ❑ Analyze the process and techniques Facebook employs to seamlessly integrate new clusters into their existing Memcache infrastructure.

Recap: Problem at Facebook

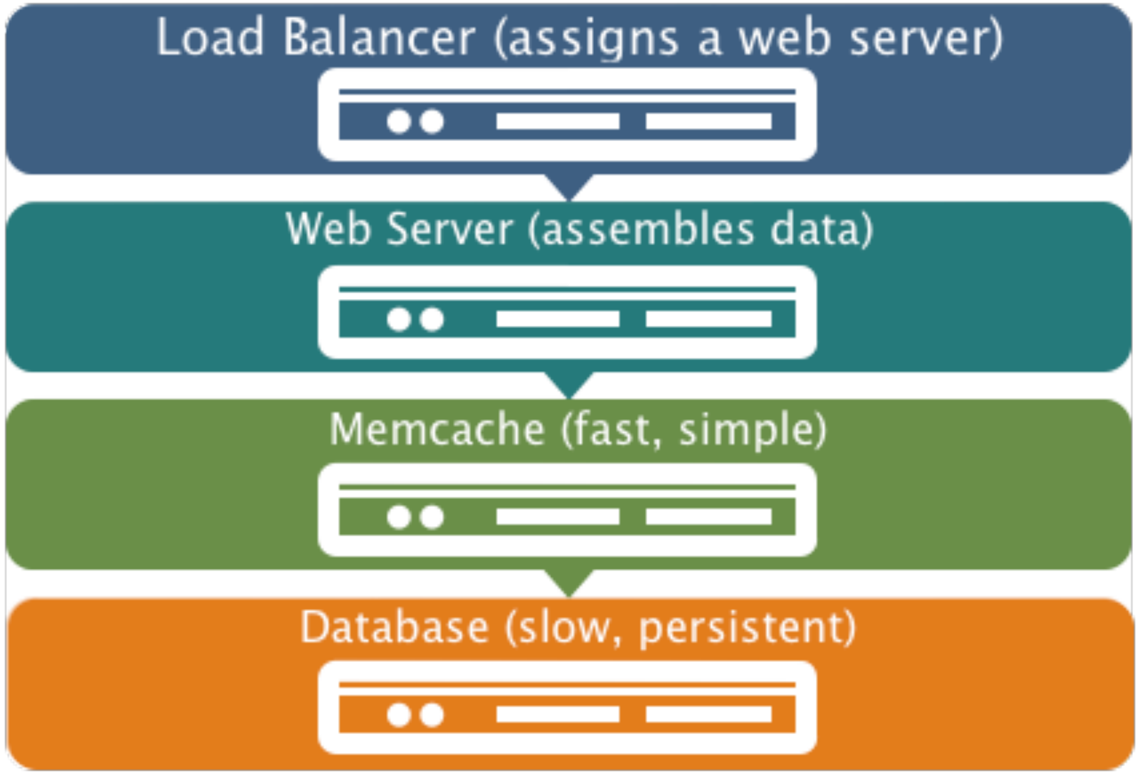
- Need a system that can handle billions of requests per second
  - Serve requests with low latency
  - Aggregate content on the fly from multiple sources
    - On average 521 distinct data items fetched for loading popular pages
    - 95^{th} percentile: 1740 data item being fetched for a page
  - Access and update very popular shared content
    - Reads >> Writes (users consume an order of magnitude more content than they create)
  - There are OK with exposing slightly stale data to users for a short time

- Traditional DB systems slow (like MySQL) and difficult to scale

Recap: Solution → Memcache

- A caching layer for fast reads and to reduce load on database servers
- Caching layer (Memcache): distributed <u>in-memory</u> hash table
  - Cache servers store key-value pairs in RAM
- Simple and fast
  - put(k,v)
  - get(k) → v
  - delete(k)

Recap: Service & Storage Architecture

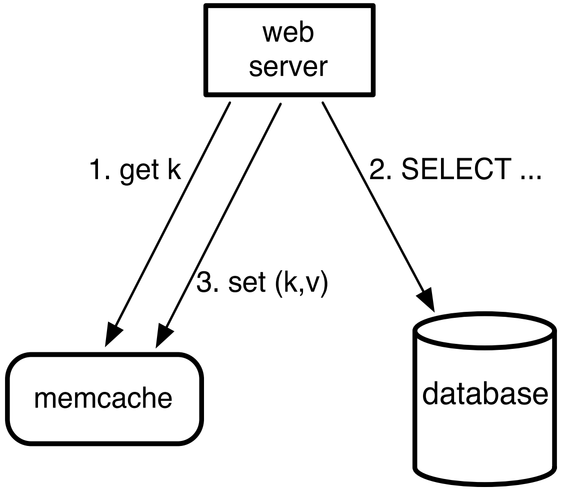


Recap: How Facebook uses Memcache?

Recap: How Facebook uses Memcache?

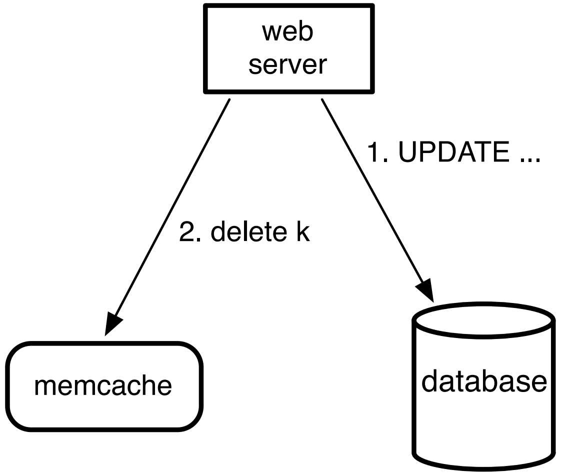
read:

```
v = get(k)      //computes hash(k) to choose mc server
if v is nil {
  v = fetch from DB
  put(k, v)   // set (k,v)
}
```



write:

```
v = new value
send k,v to DB          //write directly to DB
delete(k)       //invalidate from cache
```



Discussion: Performance

- How to do they get such high performance?

Partitioning

- They partition data using consistent hashing

- But they mention some data is extremely popular. This can lead to a high load imbalance

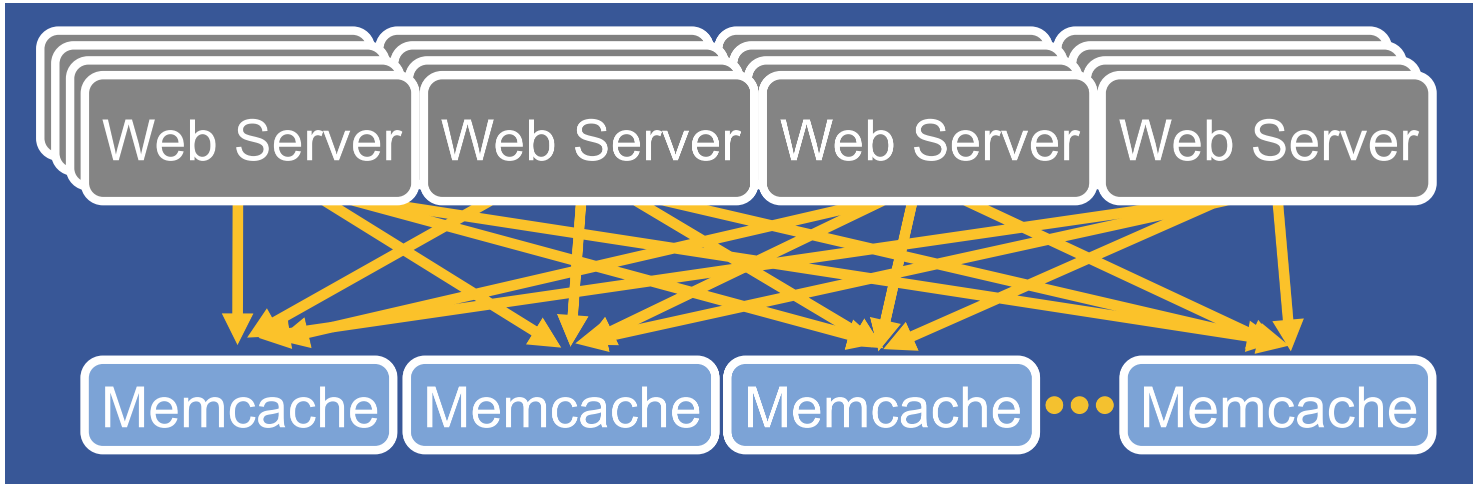
- How do they try to solve this problem?
    - Use replication

Do they replicate all objects?

All-to-All Communication

- Partitioning in Memcache leads to All-to-All Communication

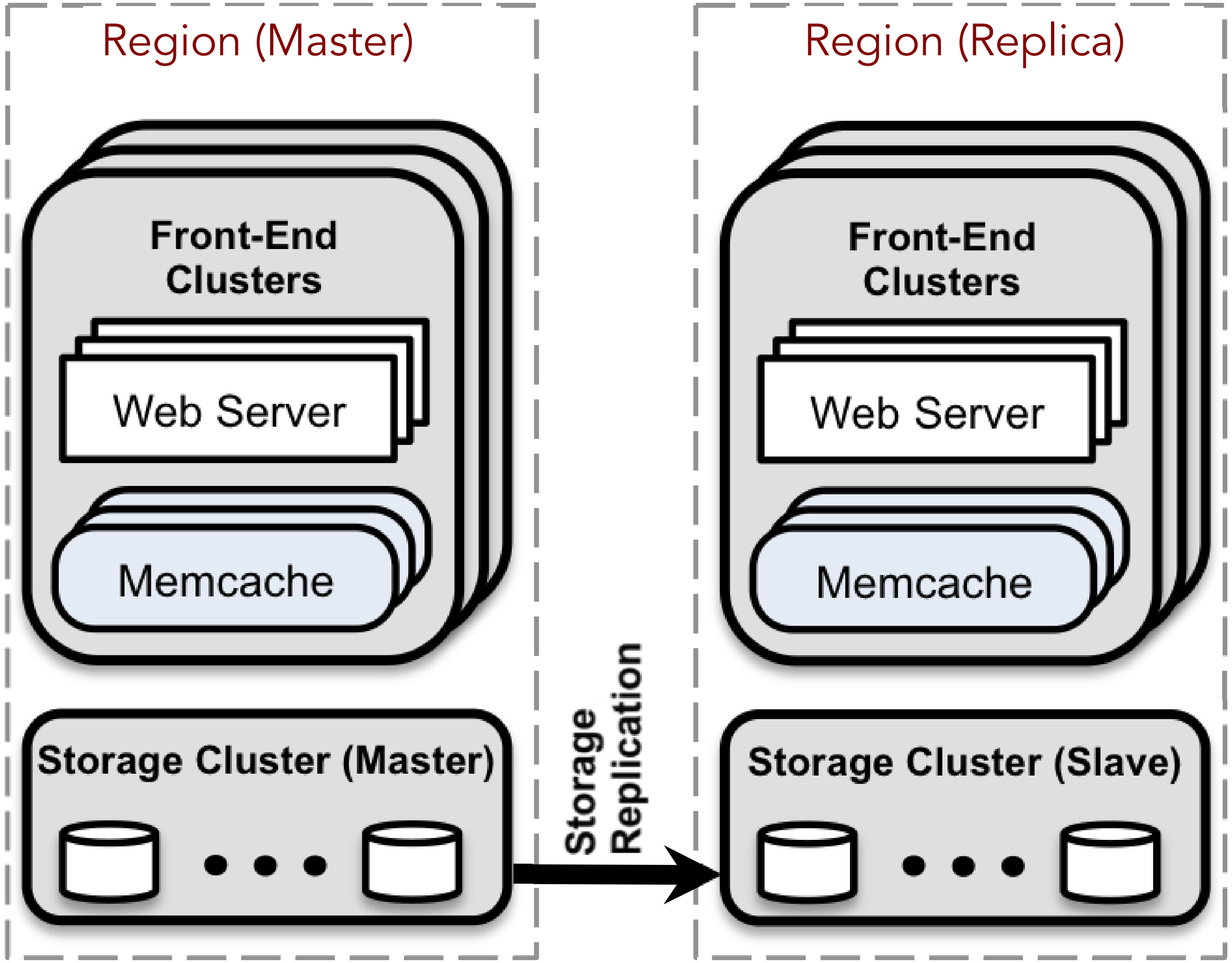
Partition forces each web server to talk to many memcache servers



Issues with all-to-all communication?

How do they address this issue?

Clusters of MC Servers and Web Servers



Summary: Partition & Replication

- Data partitioned to handle large number of requests in parallel
- Data replicated for distributing load for popular keys

Partition

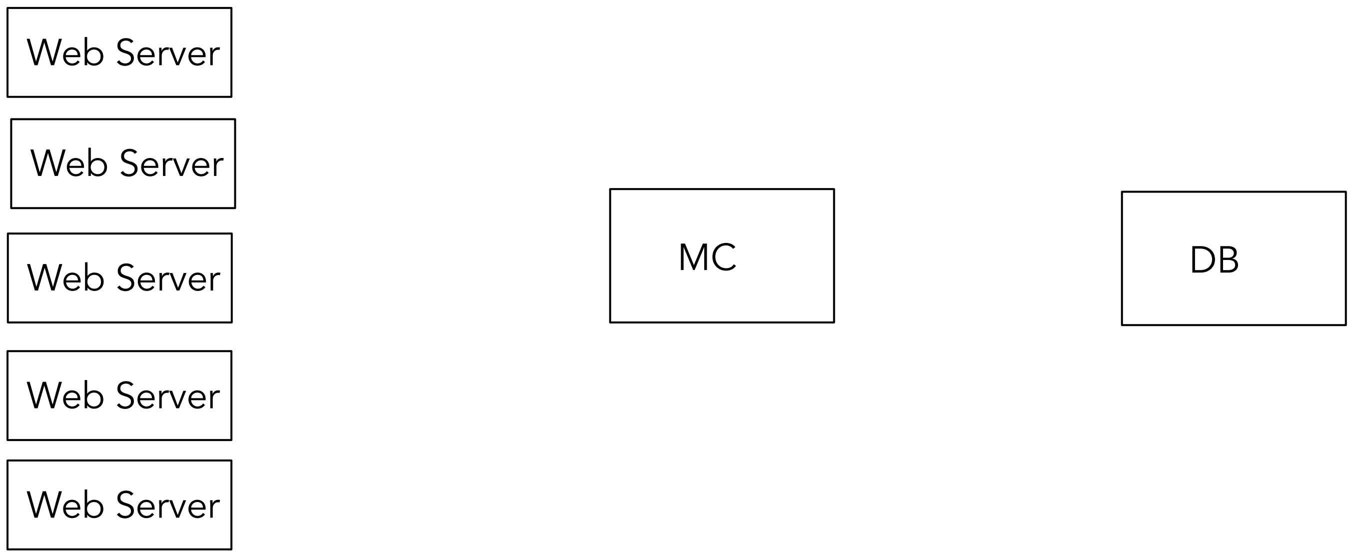
+ RAM efficient
- Not good for popular keys
- Can lead to all-to-all communication
    - Lots of TCP connections
    - Incast congestion problem

Replication

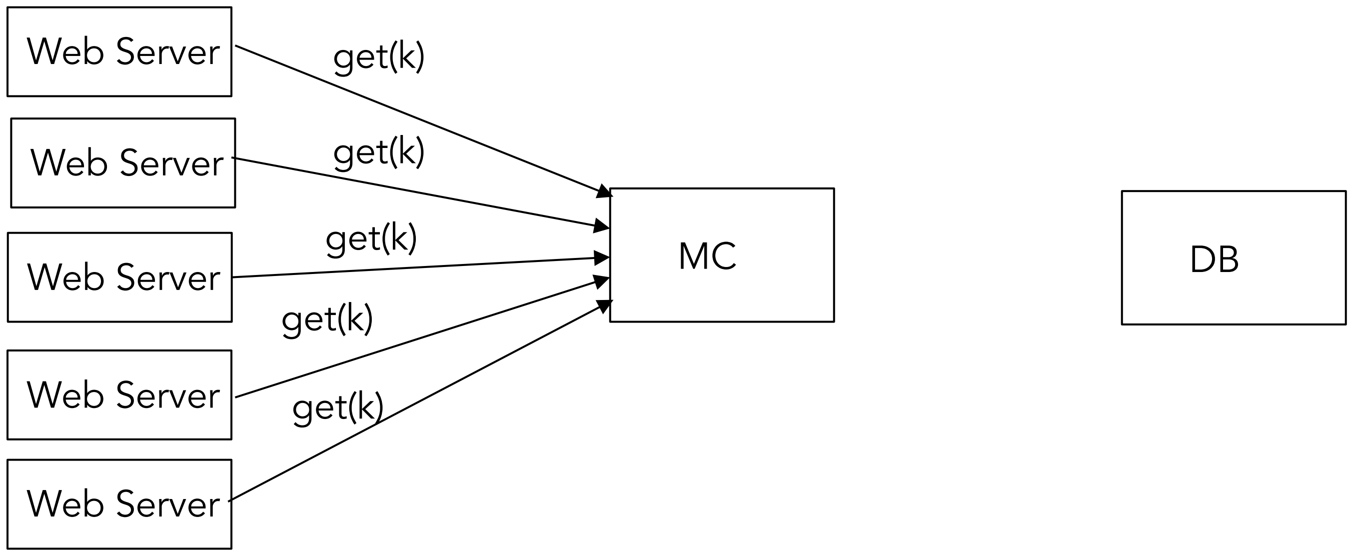
+ Good for popular keys
- Less total data can be cached

Thundering Herds Problem

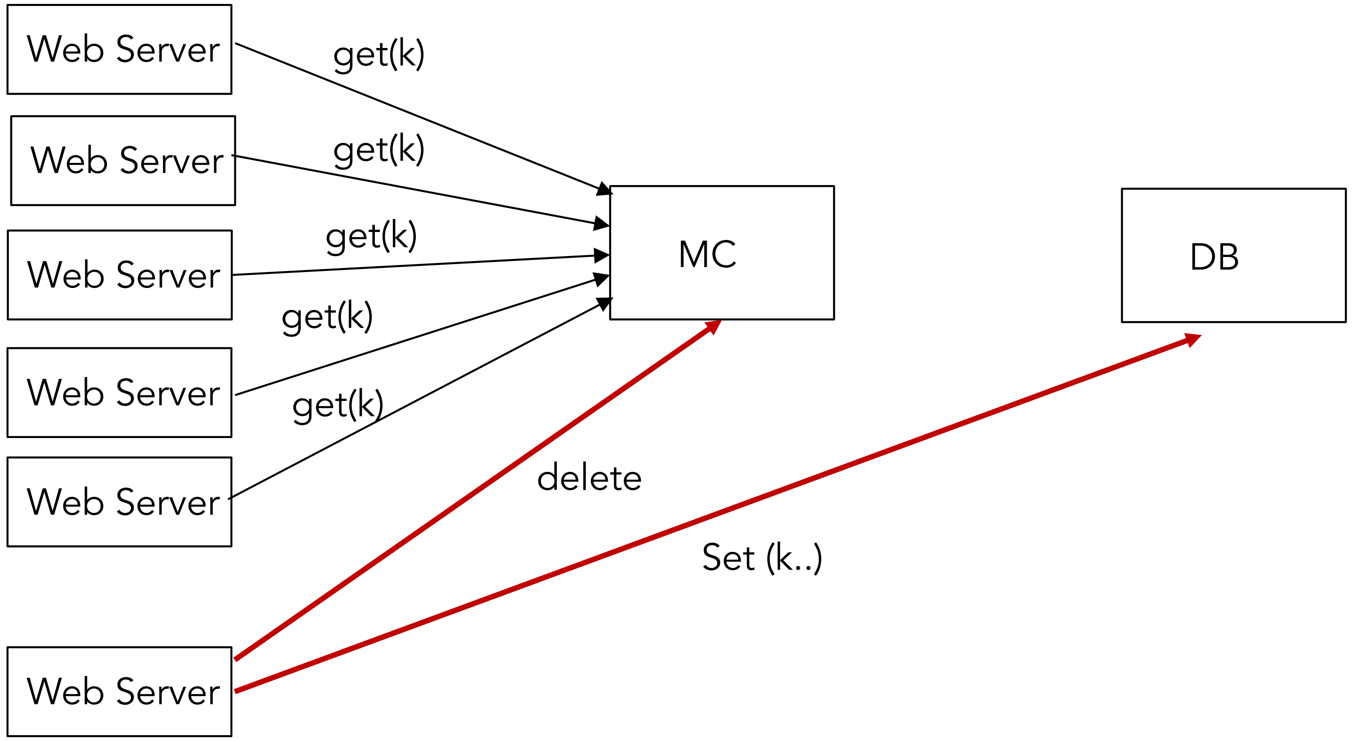
Load Issue: Thundering Herds



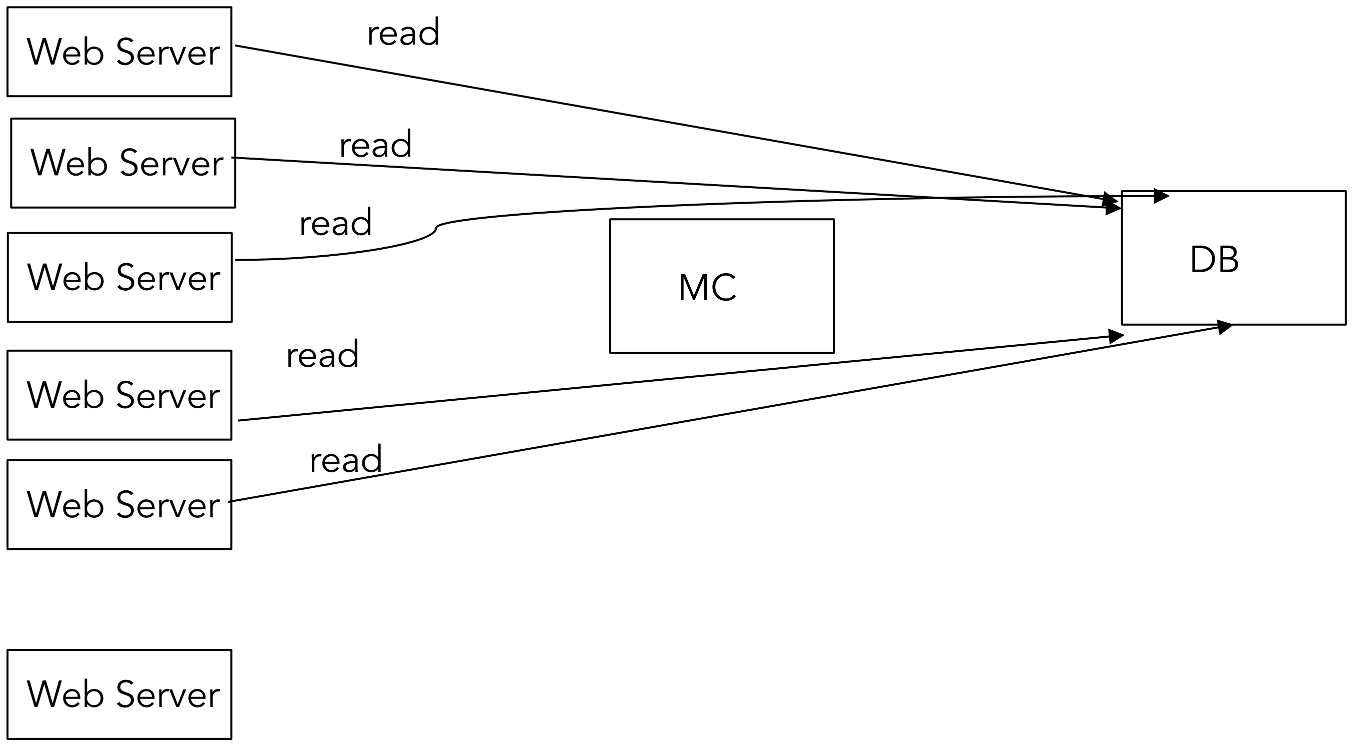
Load Issue: Thundering Herds



Load Issue: Thundering Herds

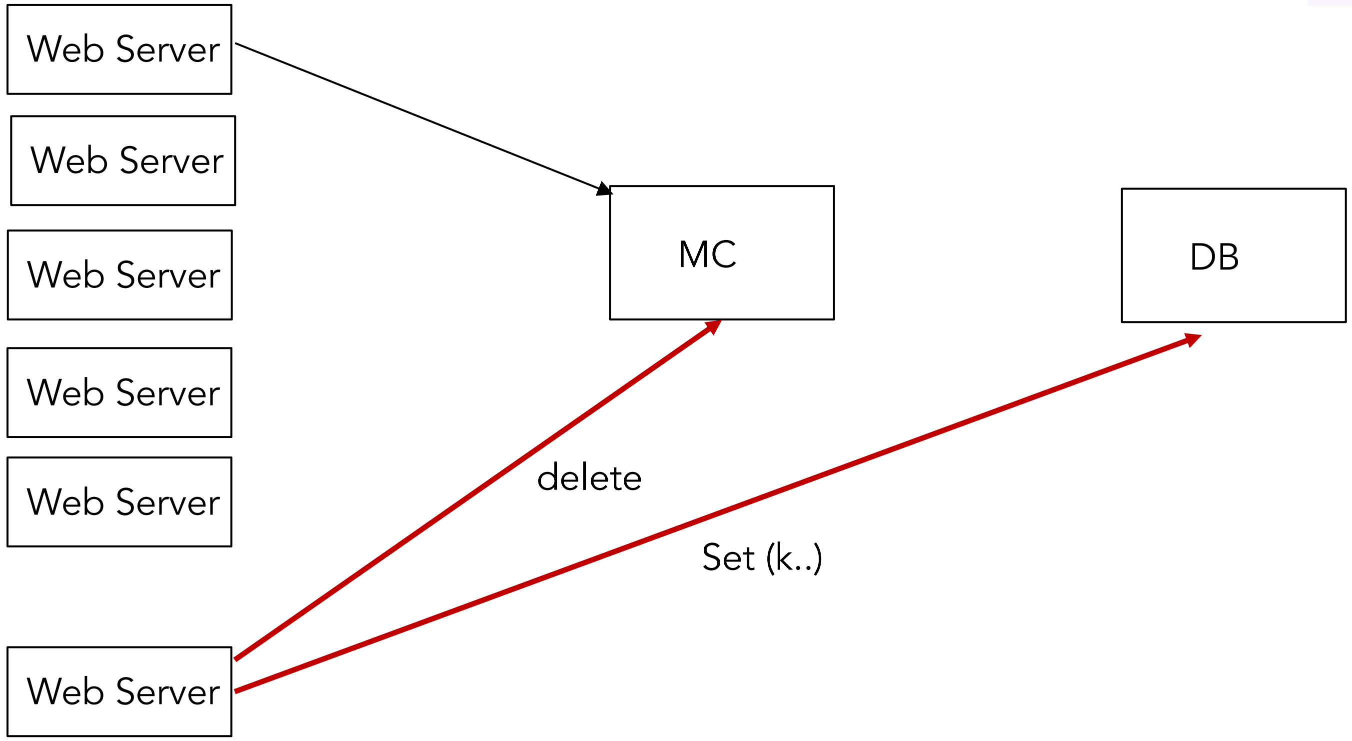


Load Issue: Thundering Herds

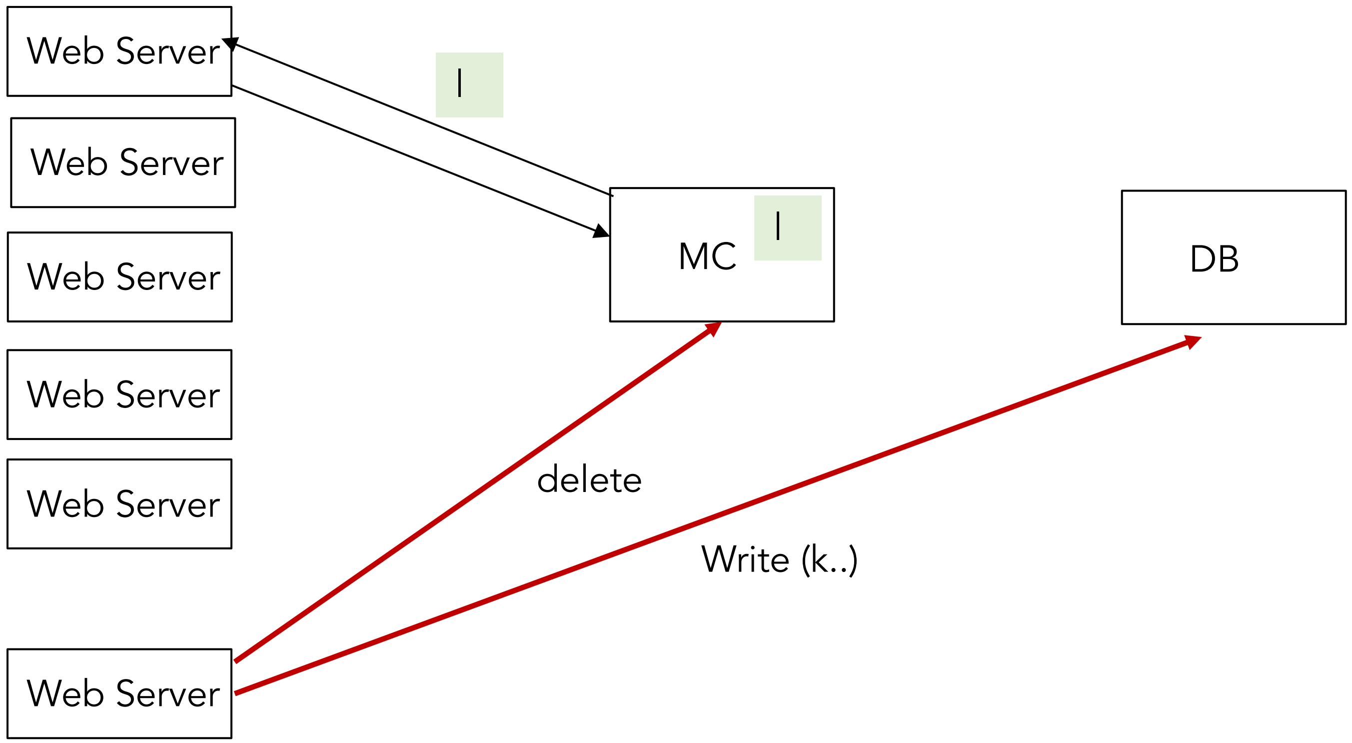


Solution: Lease

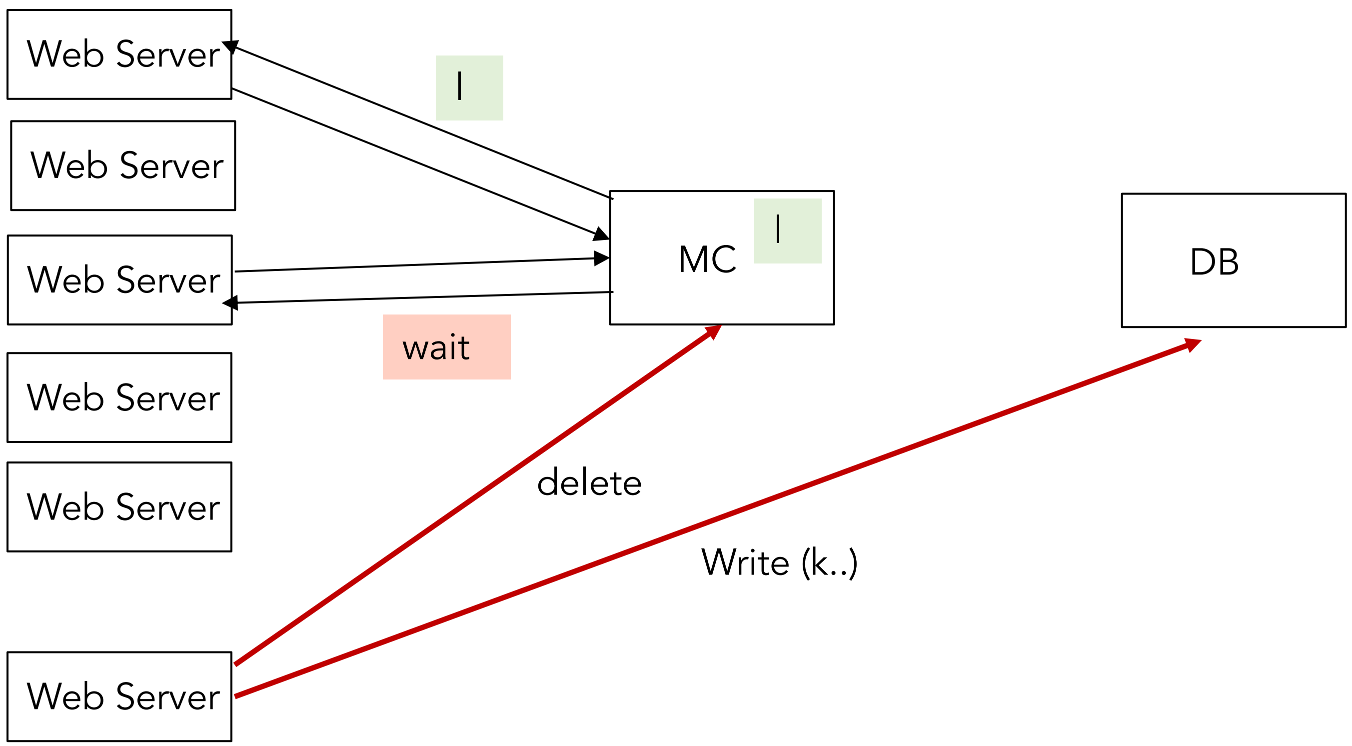
The lease is a 64-bit token bound to the specific key the client originally requested



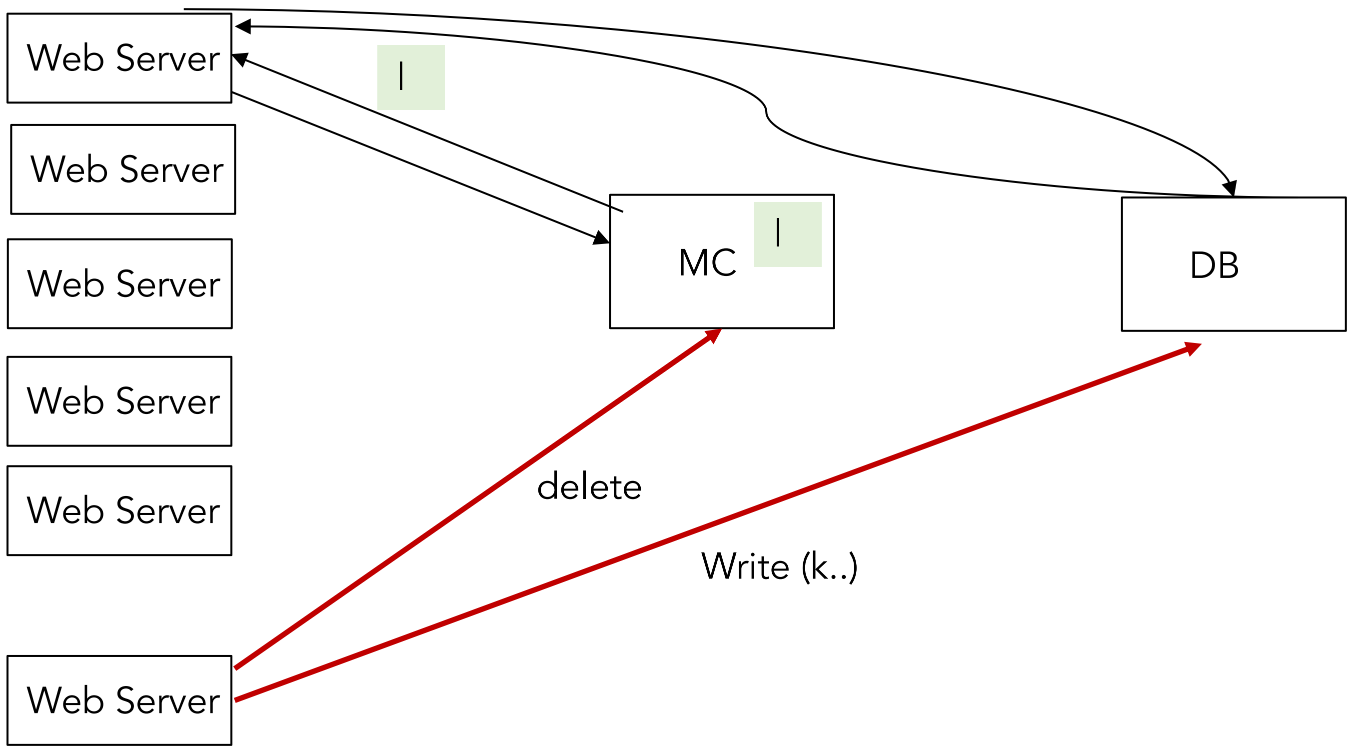
Solution: Lease



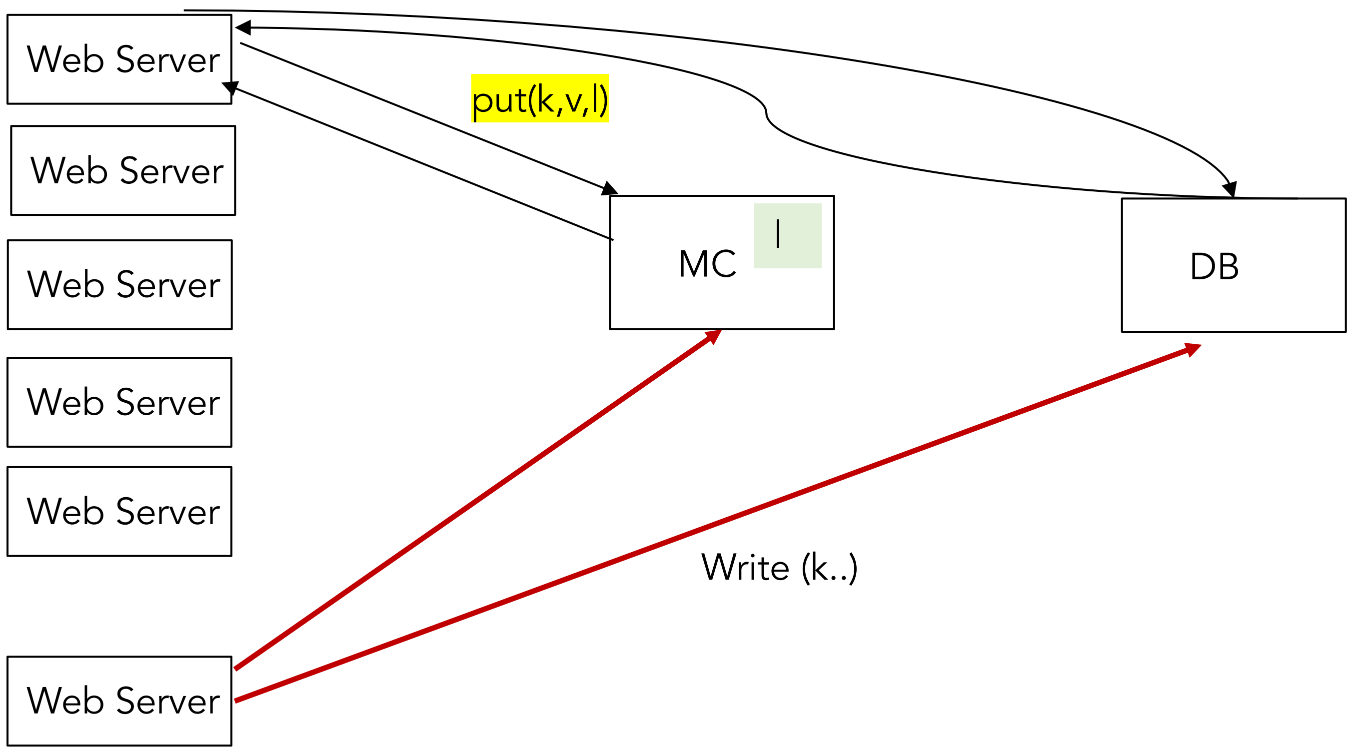
Solution: Lease



Solution: Lease



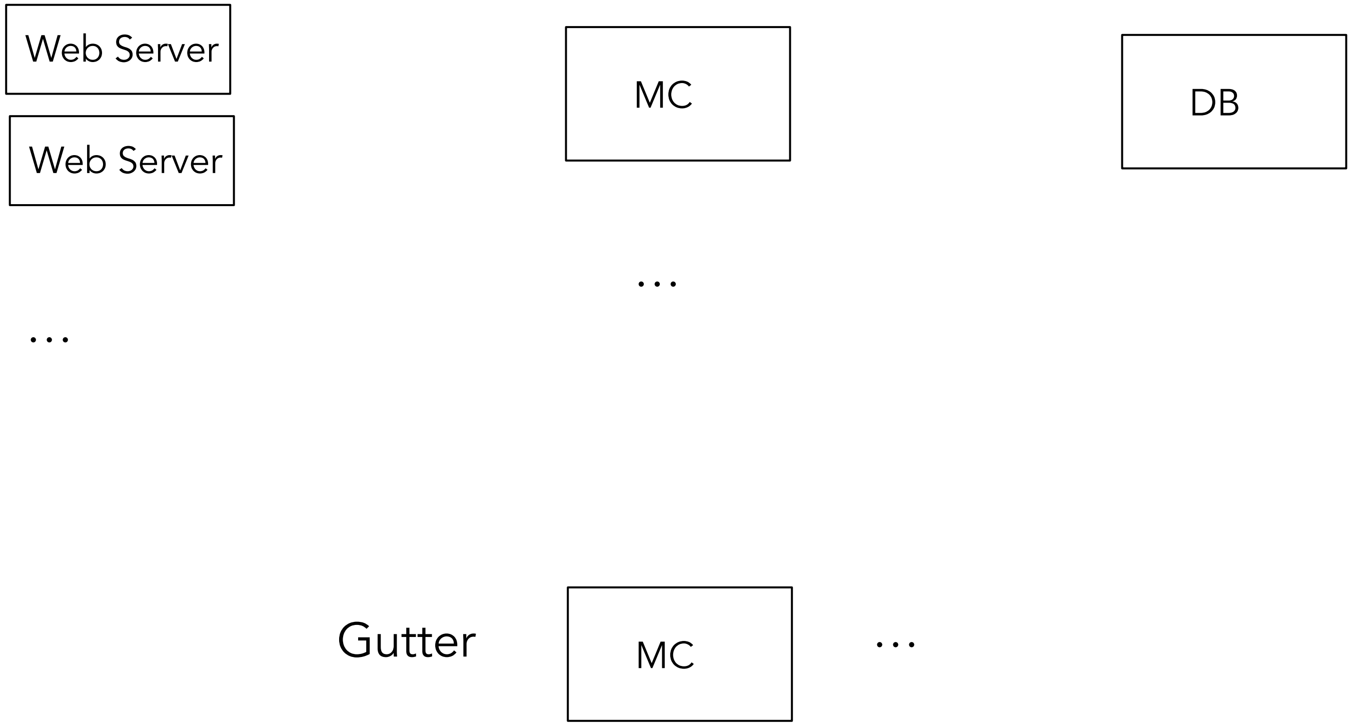
Solution: Lease



Failure Handling

- How are failures of Memcache servers handled?
  - Can't shift load to one other mc server—too much
  - Even virtual nodes might still result in load imbalance—due to popular keys
  - Can't re-partition—due to **popular keys**
  - **Gutter** - pool of idle servers, clients only use after mc server fails
    - They account for 1% of mc servers in a cluster

Gutter Pool



Bringing Up a New Cluster

- New cluster has 0% hit rate
- If clients use it, will generate big DB load
  - Serious enough that they didn't just accept the temporary hit
- Thus, the clients of new cluster first get() from existing cluster and put() into new cluster
  - Basically, copy of existing cluster to new cluster

Consistency Issues

- At high level source of problem:
  - Lots of copies of the data around
    - DB (master/slave), Memcache servers, gutter servers
- Problem:
  - You might not just have stale data
  - But stale data might be left in the system indefinitely

RACE Condition (Stale Set)

- S1: get(k) – cache miss
- S1: read from DB → v1
- S2: writes k=v2 → DB
- S2: delete(k)
- S1: put(k,v1)

- Stale version of data will be cached indefinitely

Solution: Lease

- S1: get(k) – miss (+ lease)
- S1: read from DB → v1
- S2: writes k=v2 → DB
- S2: delete(k) (also invalidate the lease)
- S1: put(k,v1, lease) (would not go through)

Summary: Issues they ran into

- Incast congestion
- Large number of TCP connections per server
- Thundering herds → high load on the DB
- Race conditions → inconsistencies between cache and DB
- Failures of Memcache servers
- Bringing up a new memcache server

In conclusion …

- Caching is vital in large web services
  - Can reduce DB load
  - Improve performance
- Prioritizes performance over consistency
  - Able to scale to very large loads
  - However, consistency could have been handled better

Next Lecture

- Distributed Transactions and Google Spanner