CS 582: Distributed Systems

Dynamo and Memcache

Dr. Zafar Ayyub Qazi

Fall 2024

Acknowledgements: Slides use some material from Rajesh Nishtala (Facebook), Robert Morris (MIT), and Ryan Stutsman (University of Utah)

Agenda

- Wrap up our discussion on Dynamo (~40mins)
- Scaling Memcache at Facebook (~30mins)

Specific learning outcomes

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

- ❑ Explain and analyze how Dynamo uses vector clocks to resolve conflict between different object versions
- ❑ Explain and analyze how Dynamo uses Merkle Trees to synchronize replicas
- ❑ Explain and analyze how Dynamo achieves high availability
- ❑ Explain how Memcache is used at Facebook
- ❑ Analyze the caching architecture of Memcache at Facebook
- ❑ Analyze how Facebook was able to scale the performance of the Memcache system

Dynamo summary of key design qns & ideas

1. How to partition and replicate data?
2. How to route and handle requests ?
3. How to cope with node failures?
4. What sort of consistency guarantees to provide?
5. How to resolve conflicts between replicas?
6. How to detect failures?

Consistent hashing + virtual nodes + preference lists

Coordinator nodes + Quorum + Gossip

Sloppy quorum + hinted handoff

Eventual Consistency

Recap: Conflicts

- Suppose N = 3, W = R = 2, nodes A, B, C
  - 1st put(k, ...) completes on A and B
  - 2nd put(k, ...) completes on B and C
  - Now get(k) arrives, completes first at A and C

- Conflicting results from A and C
  - Each has seen a different put(k, ...)

- How is this conflict handled?

Recap: Reconciling Different Versions

- Syntactic reconciliation: when new versions subsume older versions and the system itself can determine the authoritative version
  - Vector clocks used to aid reconciliation

- Semantic reconciliation: when there are conflicted versions of object
  - Application client performs reconciliation
  - E.g., a merging different versions of a customer's shopping cart
    - An "Add to cart" operation is never lost
    - However, deleted items can resurface

Recap: Object Versions & Vector Clocks

- Dynamo treats the results of each modification as a new and immutable version of the data
  - Allows for multiple versions of an object to be present at the same time

- Use vector clocks to capture (potential) causality between different versions of the same object

- Idea: Track "ancestor-descendant" relationships between different versions of data stored under the same key

Recap: Dynamo's System Interface

- get(key)→[value], context
  - Returns one value or multiple conflicting values
  - Context describes version(s) of value(s)

- put(key, context, value)→OK
  - Context indicates which versions this version supersedes or merges

Recap: Version Vectors (Vector Clocks)

- Version Vector: List of (coordinator node, counter) pairs
  - E.g., [(A,1), (B,3),…]

- Dynamo stores a version vector with each stored key-value pair

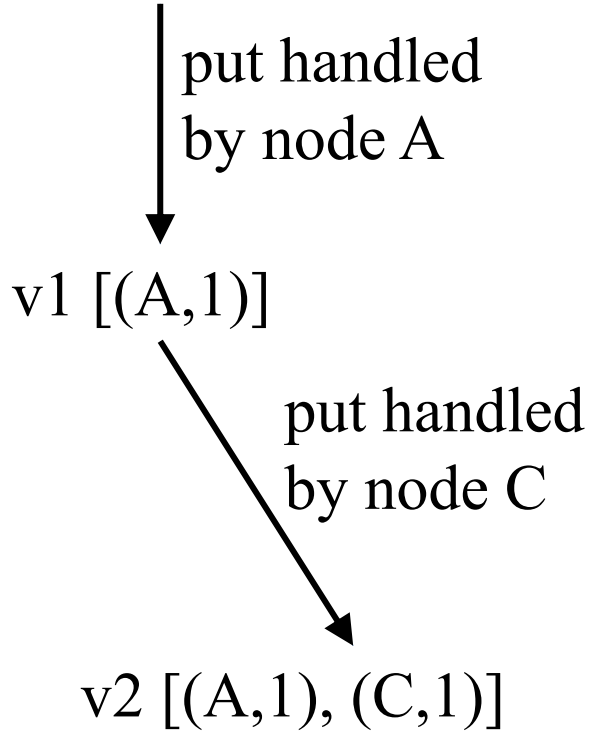
Recap: Version Vectors: Dynamo's Mechanism

- Rule: If vector clock comparison of v1 < v2, then the first is an ancestor of the second
  - Dynamo can forget v1

- Each time a put() occurs, Dynamo increments the counter in the V.V. for the coordinator node

- Each time a get() occurs, Dynamo returns the V.V. for the value(s) returned (in the "context")
  - Then users must supply that context to put()s that modify the same key

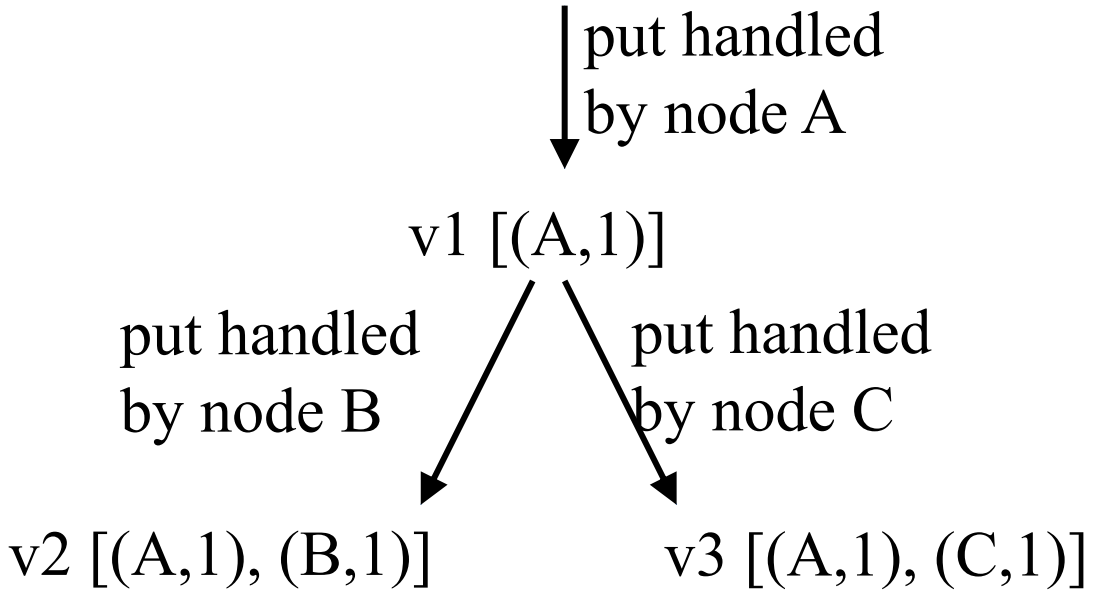
Recap: Version Vectors (auto-resolving case)



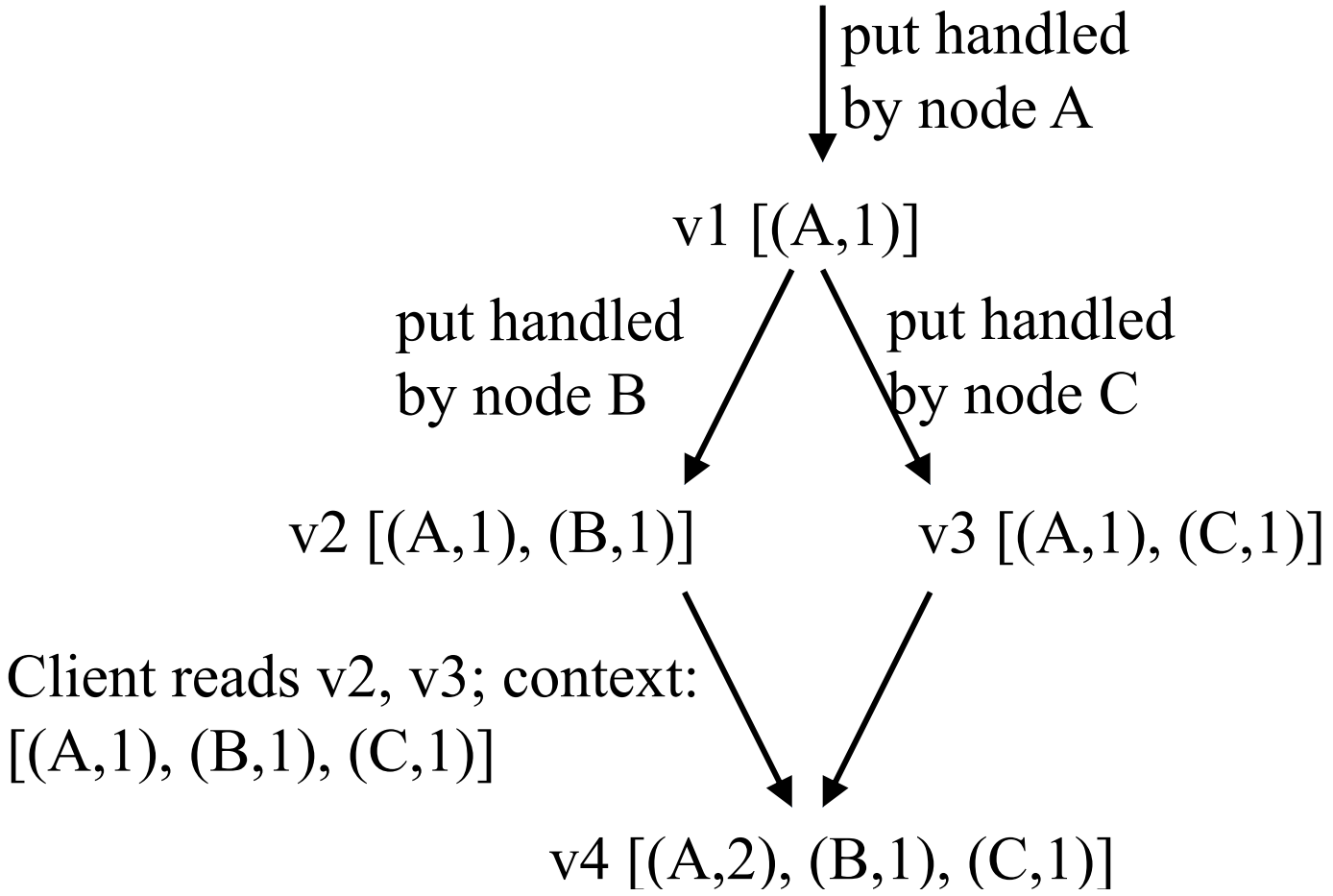
This form of reconciliation of different versions by the Dynamo system is referred to as **syntactic reconciliation**

v2 > v1, so Dynamo nodes automatically drop v1, for v2

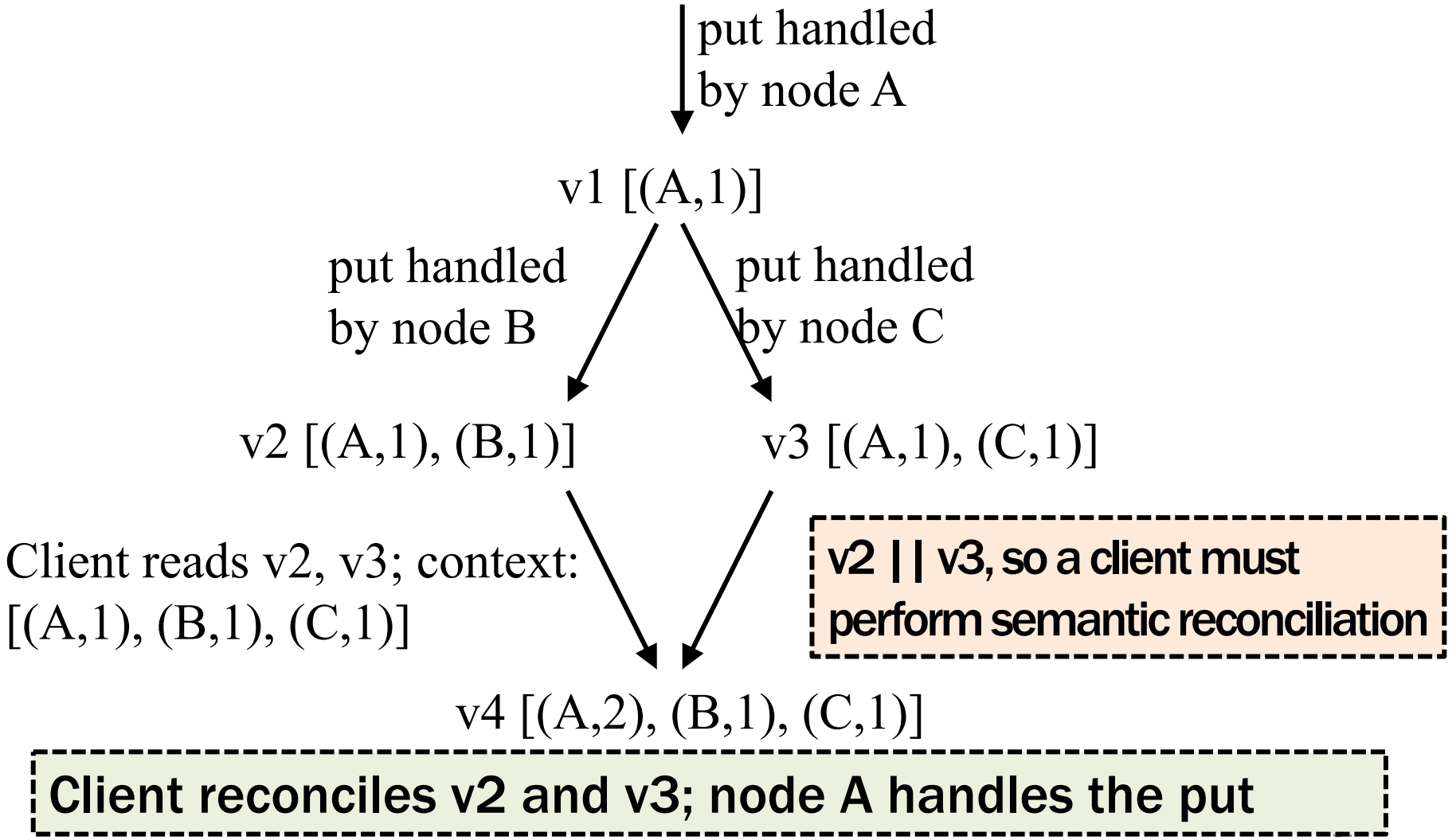
Recap: Version Vectors (app-resolving case)



Recap: Version Vectors (app-resolving case)



Recap: Version Vectors (app-resolving case)



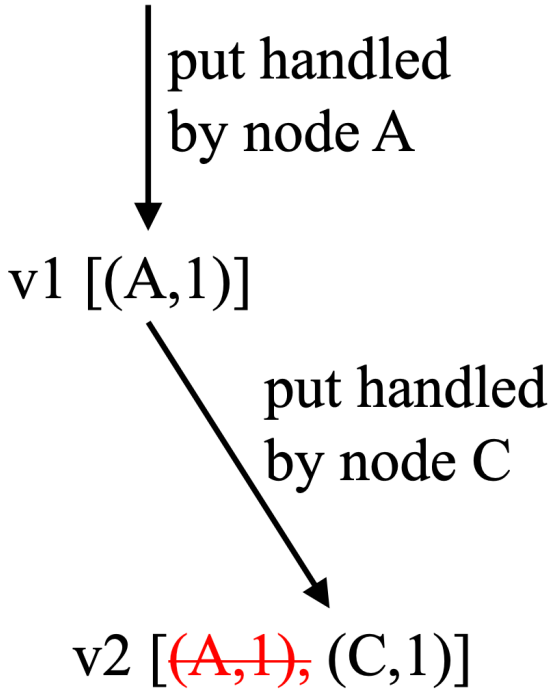
Trimming Version Vectors

- Many nodes may process a series of put()s to the same key
  - Version vectors may get long – do they grow forever?

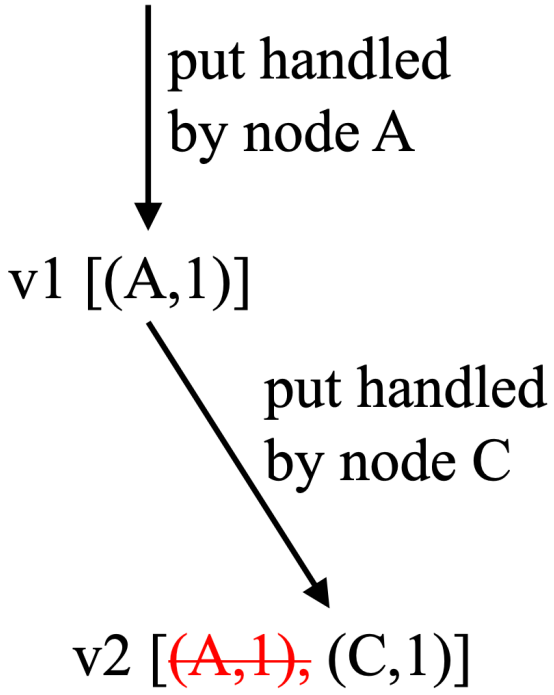
- To reduce size, clock truncation scheme
  - Dynamo stores physical timestamp with each entry

- When V.V. > 10 nodes long, V.V. drops the timestamp of the node that least recently processed that key

Impact of deleting a Version Vector Entry



Impact of deleting a Version Vector Entry



v2 || v1, so looks like application resolution is required

How to resolve data inconsistencies?

- For example, hinted handoff node crashes before it can replicate data to a node in the preference list

- Mechanism: Replica synchronization

- Two questions:
  - How to detect data inconsistencies quickly?
  - How to minimize data transfer cost when synchronizing replicas?

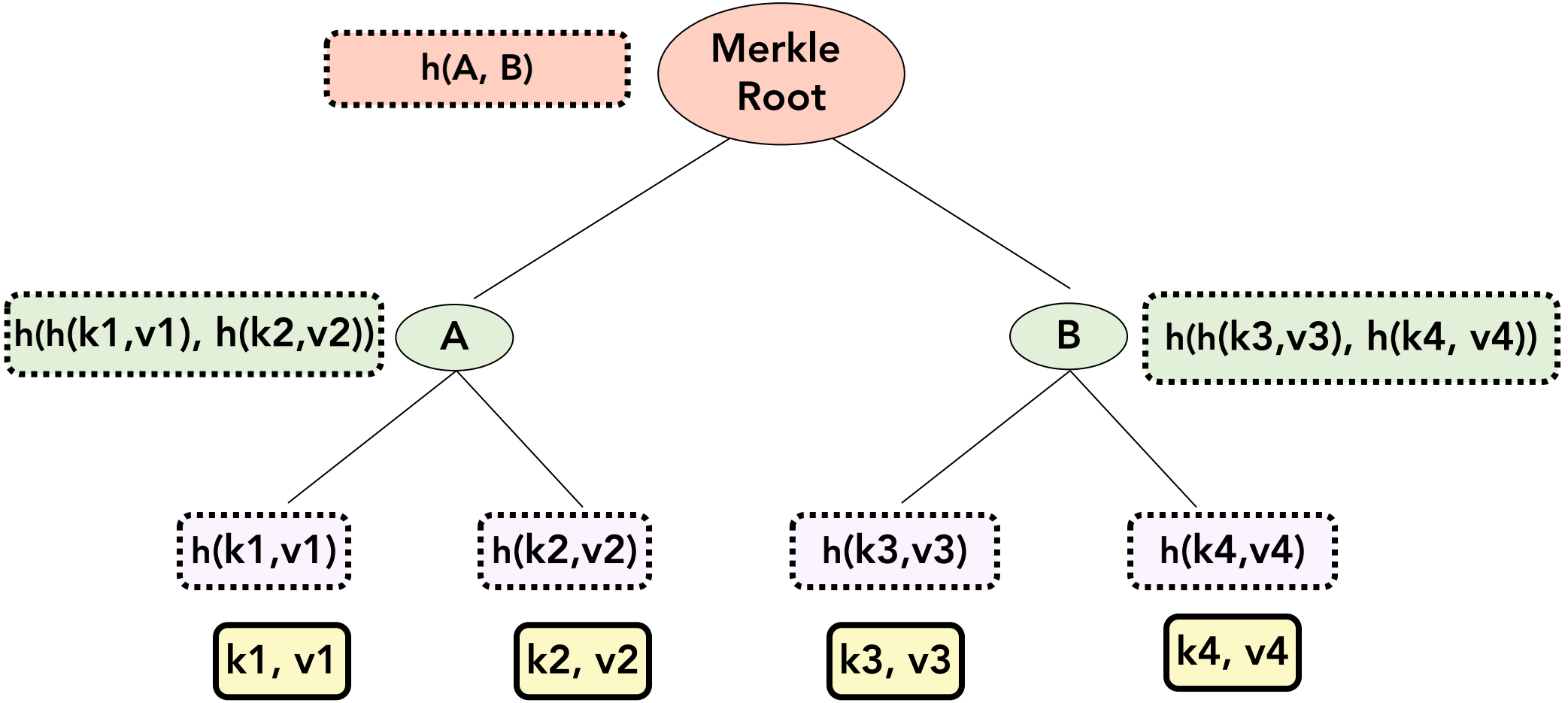
Efficient Synchronization with Merkle Trees

Merkle Trees Background

- Invented by Ralph Merkle around 1979
- Ralph was co-inventor of public key cryptography
  - With Martin Hellman and Whitefield Diffie
- Merklee trees used in many systems
  - Cryptocurrencies: e.g., Bitcoin and Ethereum
  - Data stores: e.g., Amazon's Dynamo, Apache Cassandra
  - And many others

Merkle Trees

Merkle Trees



Merkle Trees

- Each node maintains a separate Merkle tree for each key range
  - This allows nodes to compare whether the keys within a key range are up-to-date
- Nodes exchange root of the Merkle tree
  - Corresponding to the key ranges they host in common
  - If the root match → synchronized
  - Otherwise, traverse the tree
- Advantages:
  - You can quickly detect if you need synchronizing by comparing the root
  - You don't need to send all the data when synchronizing
    - You can identify the data that mismatches and needs synchronizing

For more on Merkle trees in Dynamo, checkout this interesting blog post: https://decomposition.al/blog/2019/05/31/how-i-learned-about-merklix-trees-without-having-to-become-a-cryptocurrency-enthusiast/#fnref:1

Failure Detection

- Nodes do it locally (through timeouts)
- They don’t send any heartbeat or ping messages
  - Instead assume clients traffic is steady & can be used for timely detection

Homework: Varying N, R, W

N	R	W
3	2	2
3	3	1
3	1	3
3	3	3
3	1	1

How changing these values impacts consistency, performance, availability, and durability?

Dynamo summary of key ideas

1. How to partition and replicate data?
2. How to route and handle requests ?
3. How to cope with node failures?
4. What sort of consistency guarantees to provide?
5. How to resolve conflicts between replicas?
6. How to detect failures?

Consistent hashing + virtual nodes + preference lists

Coordinator nodes + Quorum + Gossip

Sloppy quorum + hinted handoff

Eventual consistency

Versions + V. clocks + Semantic resolution (by app) + Merkel trees

Locally through timeouts

In conclusion …

- A highly influential design
  - Used for storing the state of a number of core services at Amazon
  - Influenced many other designs, including Apache Cassandra (facebook), Project Voldemort (LinkedIn), Riak (Basho)
- Provides very high levels of availability
  - Lets writes and reads return quickly, even when partitions and failures
  - At the cost of some inconsistency
    - May not be suitable for many applications that require stronger consistency
- Version vectors allow some conflicts to be resolved automatically; others left to the application

Next …

Scaling Memcache at Facebook

Why are we discussing this system?

- It's an experience paper, not about new ideas/techniques
- Three ways to read it:
  - Impressive story of super high capacity from mostly-off-the-shelf software
  - Cautionary tale about not taking consistency seriously from the start
  - Fundamental struggle between performance and consistency
- We can argue with their design, but not their success

Problem at Facebook

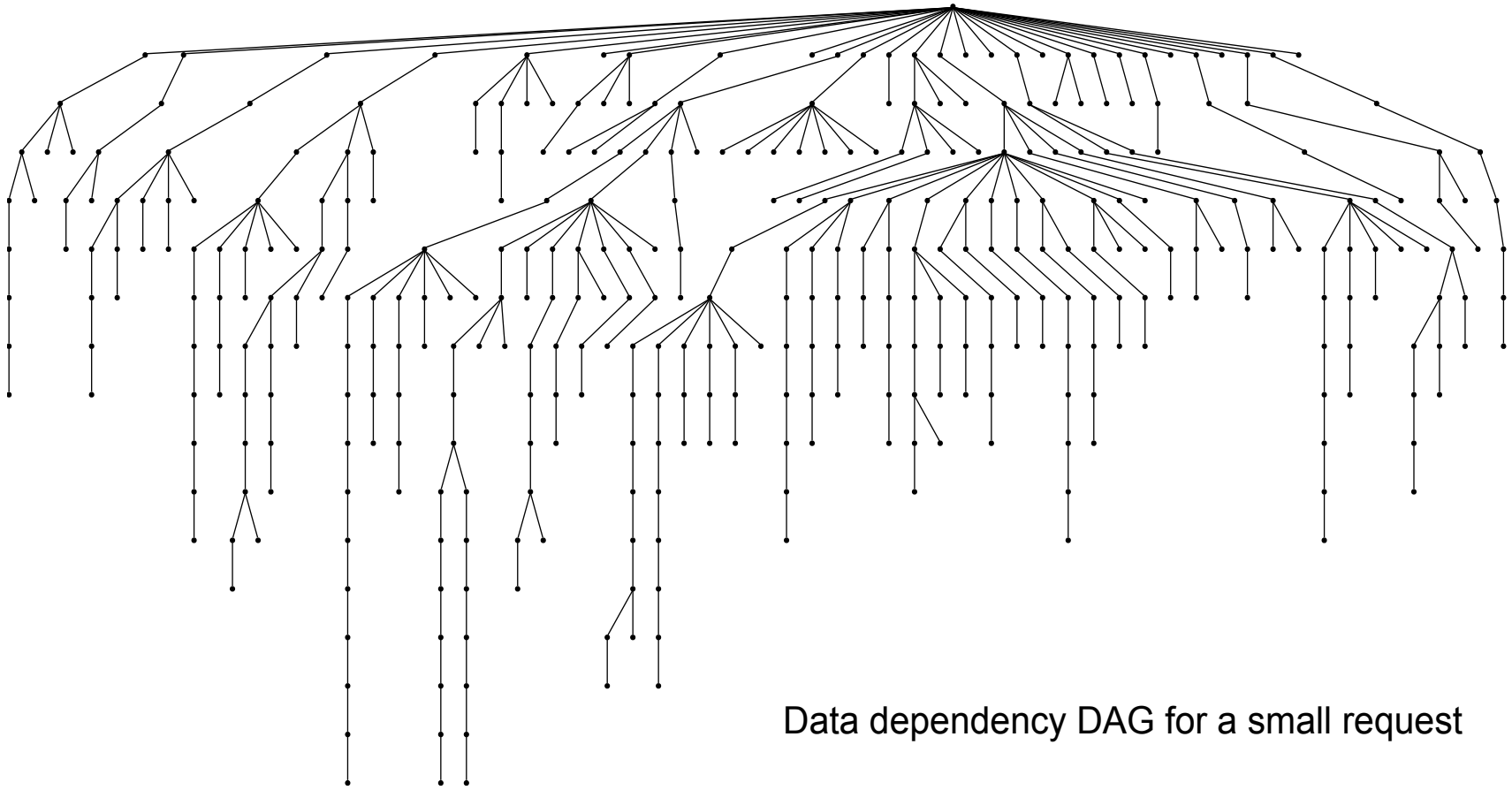
- Need a system that can handle billions of requests per second
  - Serve requests with low latency
  - Aggregate content on the fly from multiple sources
    - On average 521 distinct data items fetched for loading popular pages
    - 95th percentile: 1740 data item being fetched for a page
  - Access 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

Solution → Memcache

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

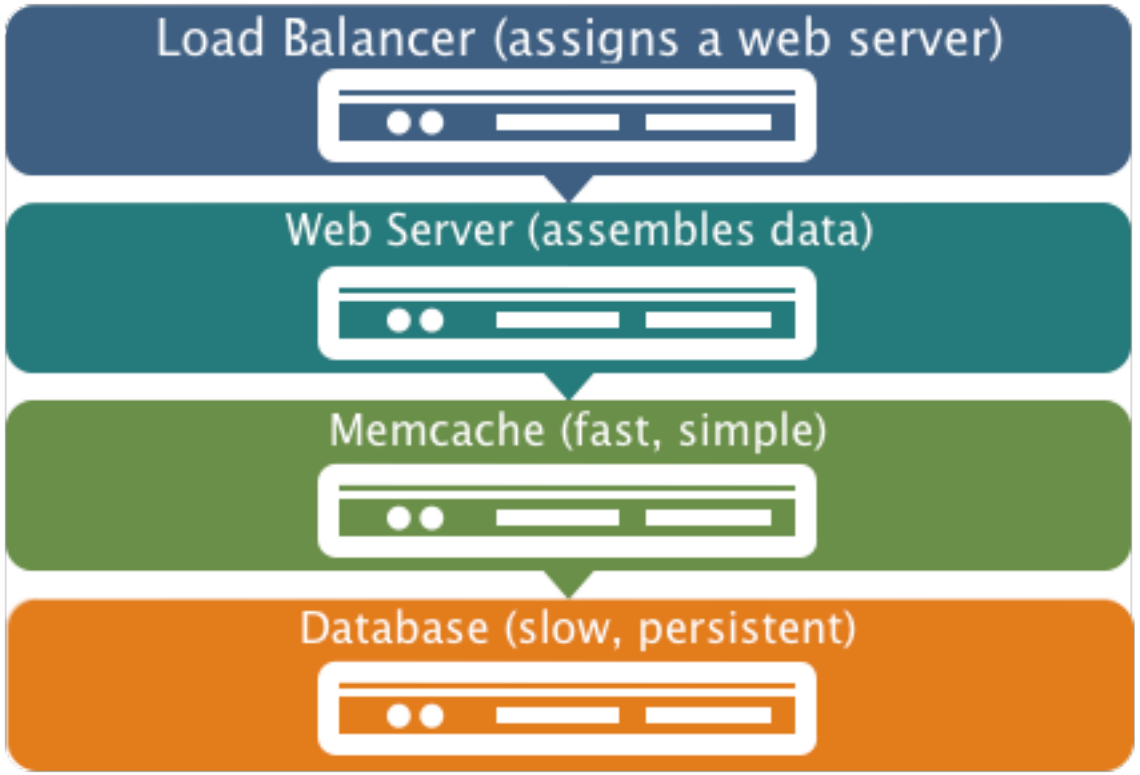
Why used Memcache?

High fanout and multiple rounds of data fetching



Data dependency DAG for a small request

Service & Storage Architecture

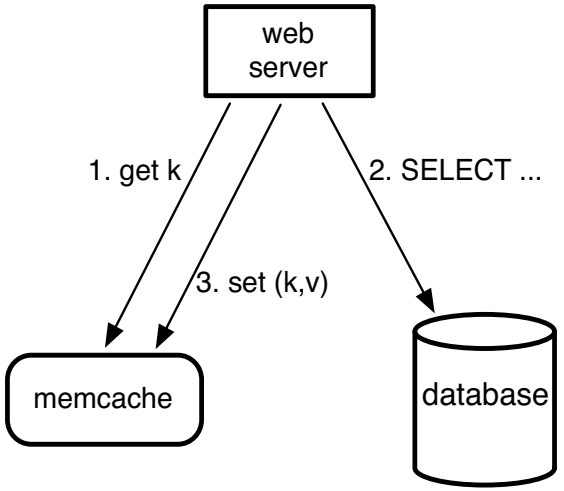


How Facebook uses Memcache?

How Facebook uses Memcache?

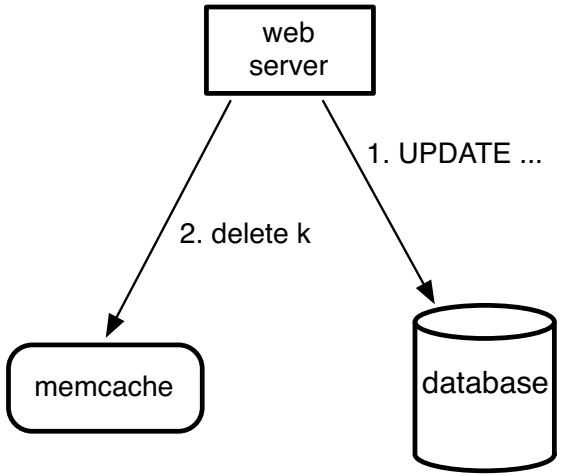
read:

```
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: Storage Architecture

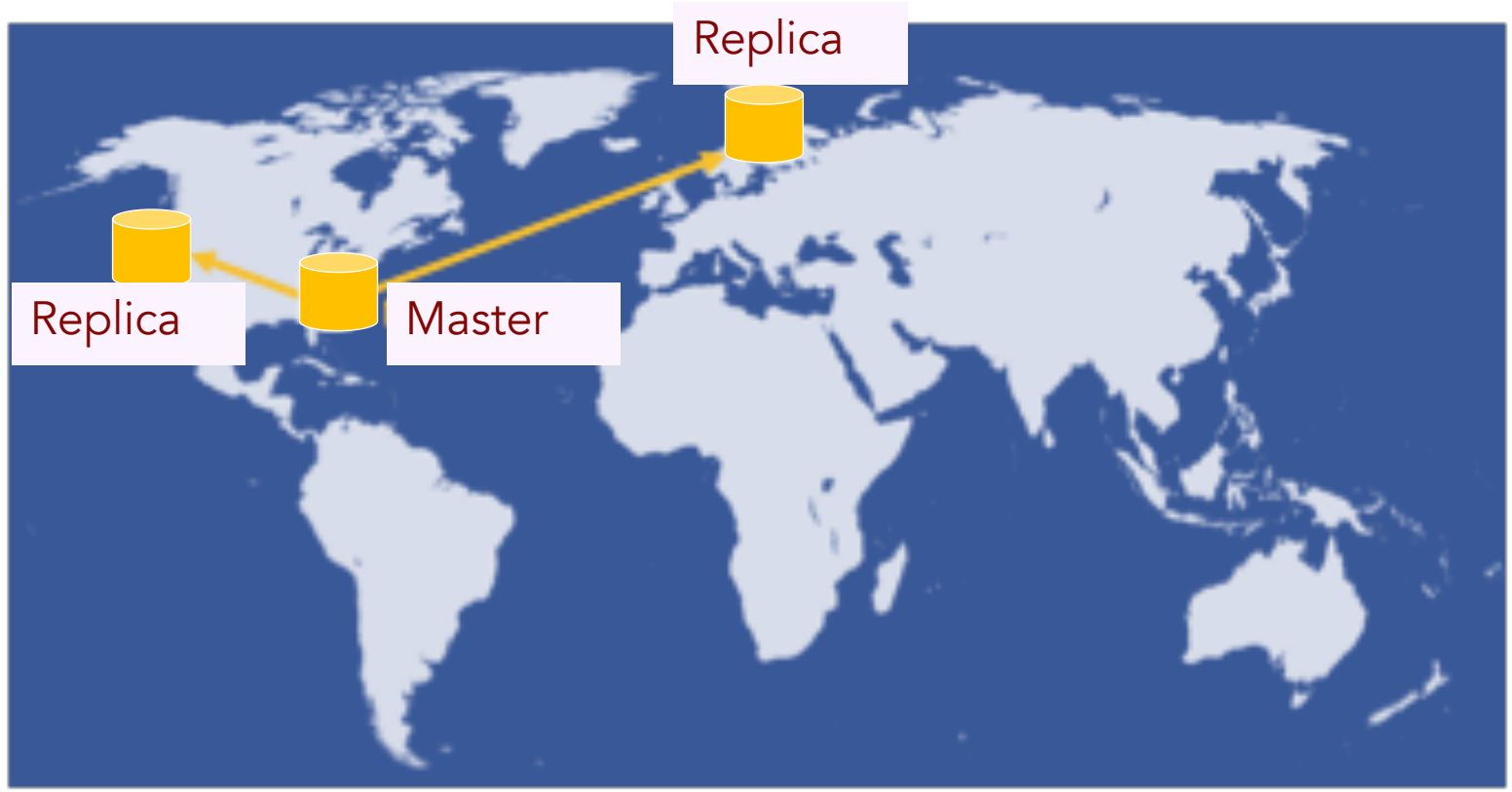
- Why all updates are directed to the DB?
  - Wouldn't this makes the latency of writes slow?

Access Latencies

nanosecond events	microsecond events	millisecond events
register file: 1ns–5ns	datacenter networking: O(1μs)	disk: O(10ms)
cache accesses: 4ns–30ns	new NVM memories: O(1μs)	low-end flash: O(1ms)
memory access: 100ns	high-end flash: O(10μs)	wide-area networking: O(10ms)
	GPU/accelerator: O(10μs)	

"Attack of the Killer Microseconds", Communications of the ACM

Overall Architecture



Geograhically Distributed Storage and Services

38

Discussion: Storage Architecture

- Why all updates are directed to the DB?
  - Wouldn't this make the latency of writes slow?
- Updates stored in DB for persistence/consistency
  - DB should be the authoritative copy of the data
  - But that does make writes slow
    - And writes propagate to the master region, delays over WAN can further increase delays
  - Tolerates slow writes because of read-heavy workload
    - And OK with stale reads for a short time

Why invalidate the cache?

- Why not update the cache directly?
  - Can save time in updating memcache

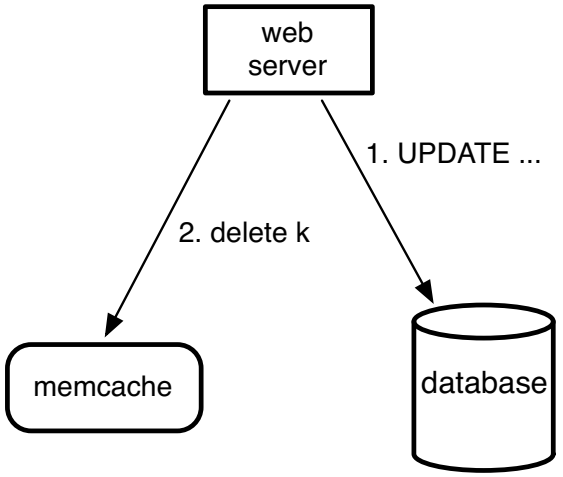
Why invalidate the cache?

write:

v = new value

send k,v to DB //write directly to DB

delete(k) //invalidate from cache



Why invalidate the cache?

Consider this example

SY= Web Server Y

Why invalidate the cache?

Consider this example

- S1: X=1 → DB
- S2: X=2 → DB
- S2: put(X,2)
- S1: put(X,1)

Web servers will continue to read stale data until another update to the key

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

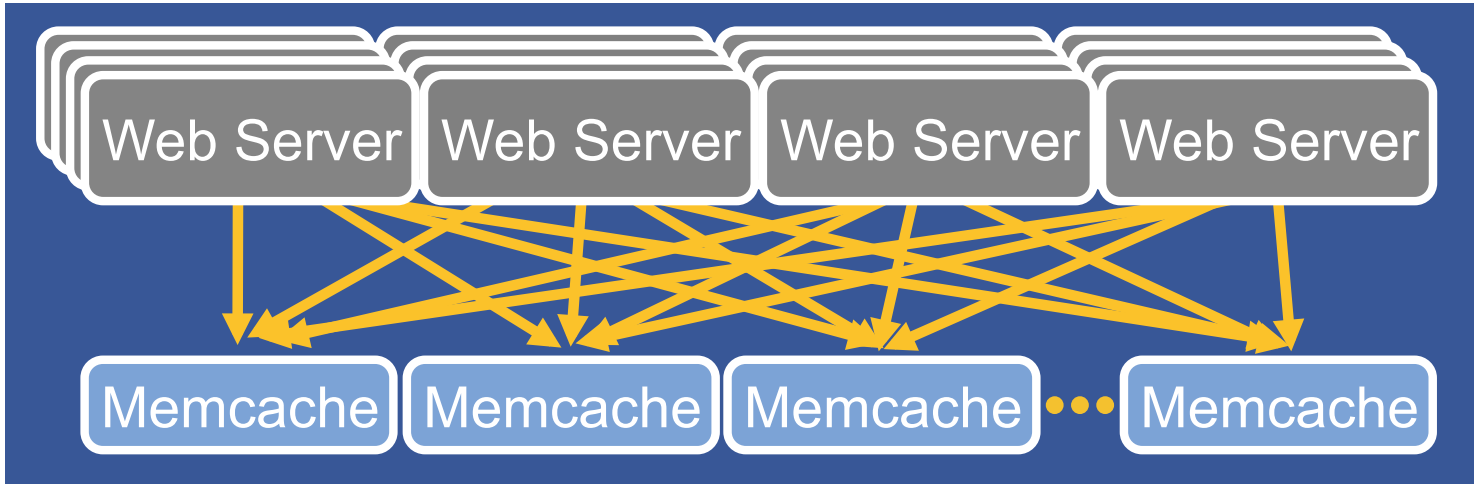
- How do they try to solve this problem?
  - Use replication

Do they replicate all objects?

All-to-All Communication

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

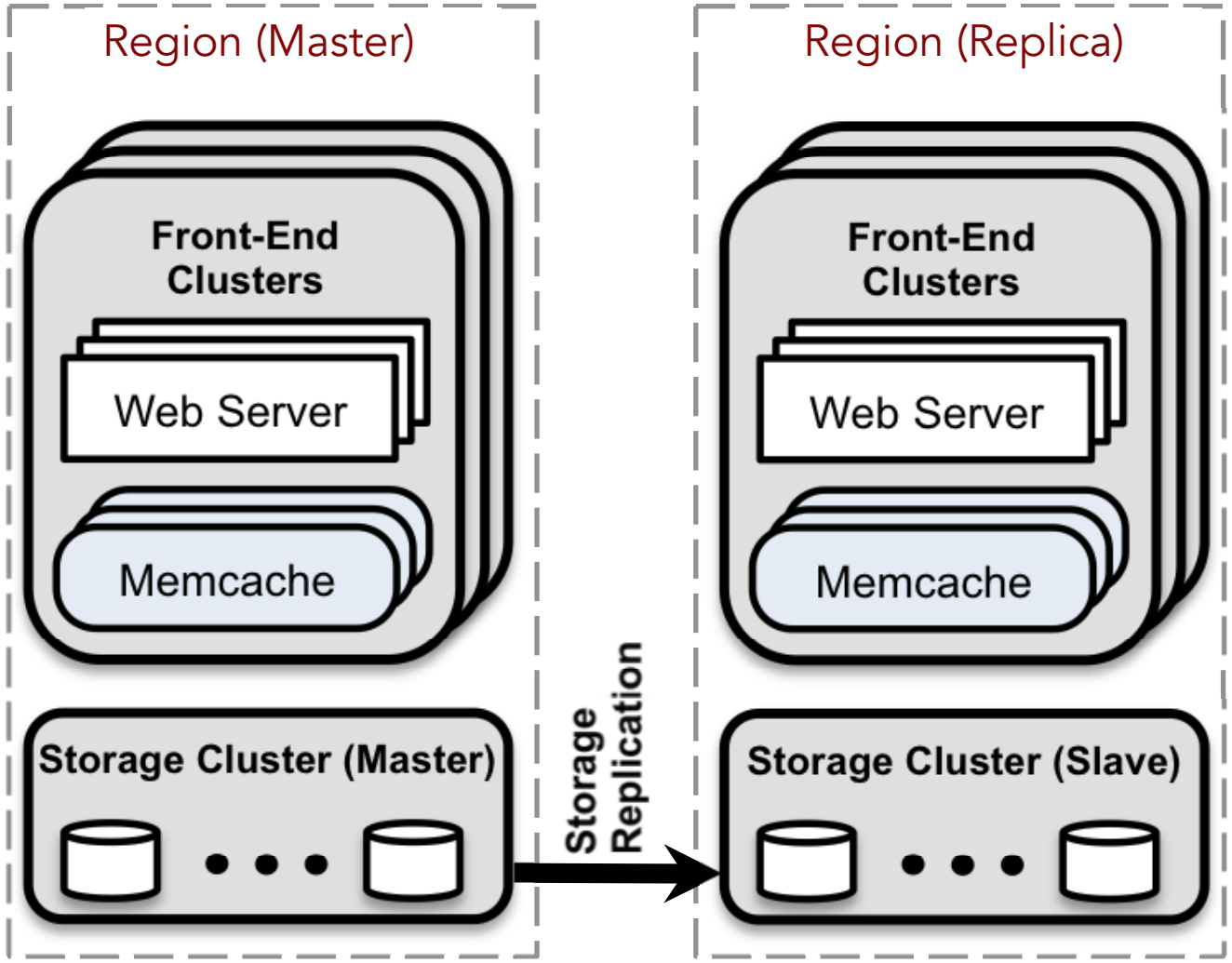
Partition forces each web server to talk to many memcache servers



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

Next Lecture

- Wrap-up discussion on Memcache
- Google Spanner