Dynamo: Key-Value Cloud Storage

Brad Karp
UCL Computer Science

CS M038 / GZ06
8th February 2013

Context: P2P vs. Data Center (key, value) Storage

- Chord and DHash intended for wide-area peer-to-peer systems
  - Individual nodes at Internet's edge
  - Central challenges: low-latency key lookup with small forwarding state per node
  - Consistent hashing to map keys to nodes
  - Replication at successors for availability under failure
- Are these techniques useful in a more traditional data-center scenario?

Amazon’s Workload (in 2007)

- Peak load: tens of millions of customers
- Tens of thousands of servers in globally distributed data centers
- Dynamo: (key, value) storage back end for Amazon’s web-based store
  - put(), get(); values “usually less than 1 MB”
- Requirements:
  - Low latency of requests: focus on 99.9% SLA
  - Highly available despite failures (measured in success of users’ operations)
  - Scalable as workload grows to more servers

Amazon’s Workload (cont’d)

- “Shopping cart service must always be able to write to and read from its data store.”
  - despite disks failing, network routes flapping, “data centers destroyed by tornadoes”
- Services that use (key, value) stores:
  - Best-seller lists
  - Shopping carts
  - Customer preferences
  - Session management
  - Sales rank
  - Product catalog

Techniques (mostly not new)

- Place replicated data on nodes according to consistent hashing
- Maintain consistency of replicated data using version vectors (“vector clocks” in paper)
- Eventual consistency for replicated data: prioritize success and low latency of writes and reads over consistency (unlike DBs)
- Efficiently sync replicas using Merkle trees

Techniques (mostly not new)

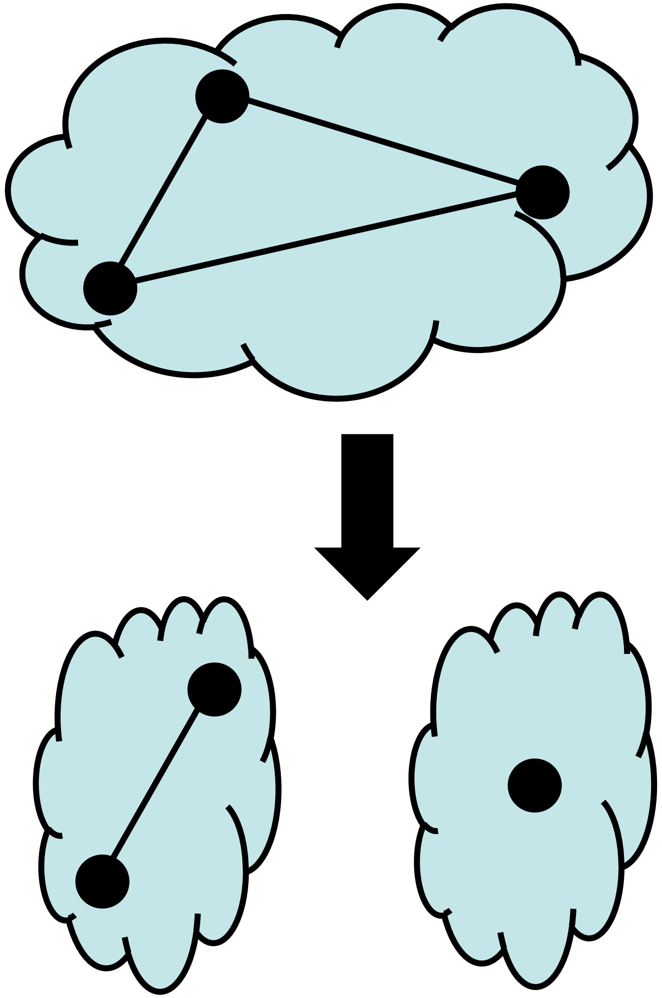
- Place replicated data on nodes according to consistent hashing
- Maintain consistency of replicated data

Key trade-offs:
response time vs. **consistency** vs. **durability**

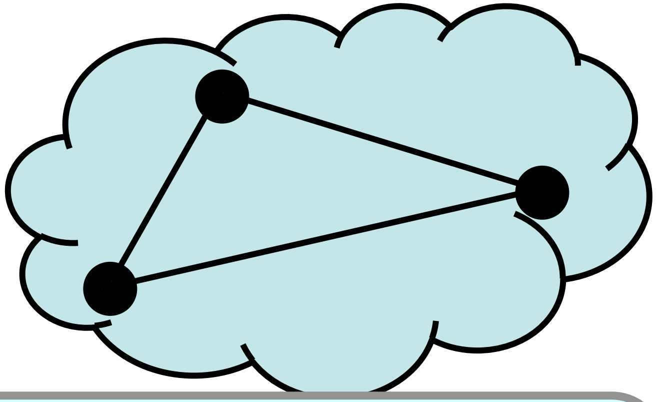
- Eventual consistency for replicated data: prioritize success and low latency of writes and reads over consistency (unlike DBs)
- Efficiently sync replicas using Merkle trees

Partitions Force Choice Between Availability and Consistency

- Suppose 3 replicas are partitioned into 2 and 1
- If one replica fixed as master, no client in other partition can write
- In Paxos-based primary-backup, no client in minority partition can write

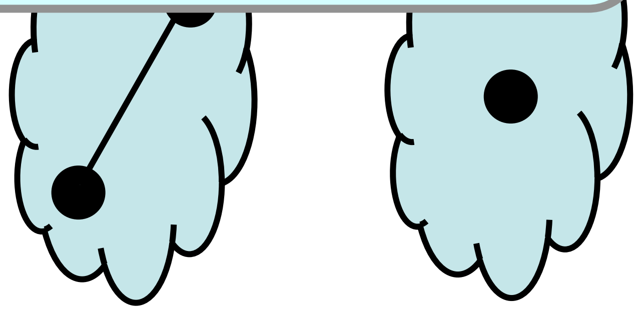


Partitions Force Choice Between Availability and Consistency



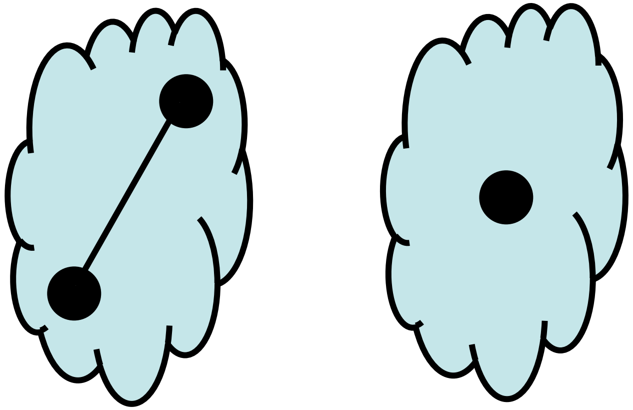
- Suppose 3 replicas are partitioned into 2 and 1
- If one replica fixed

Traditional distributed databases emphasize **consistency** over availability when there are partitions



- In Paxos-based primary-backup, no client in minority partition can write

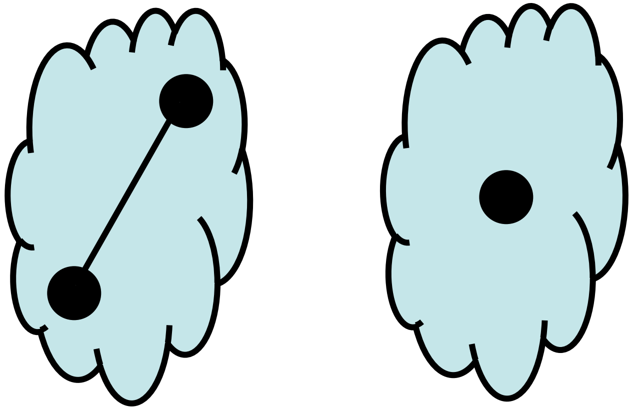
Alternative: Eventual Consistency



- Tell client write complete when only some replicas have stored it
- Propagate to other replicas in background
- Allows writes in both partitions…
- …but risks:
  - returning stale data
  - write conflicts when partition heals

Alternative: Eventual Consistency

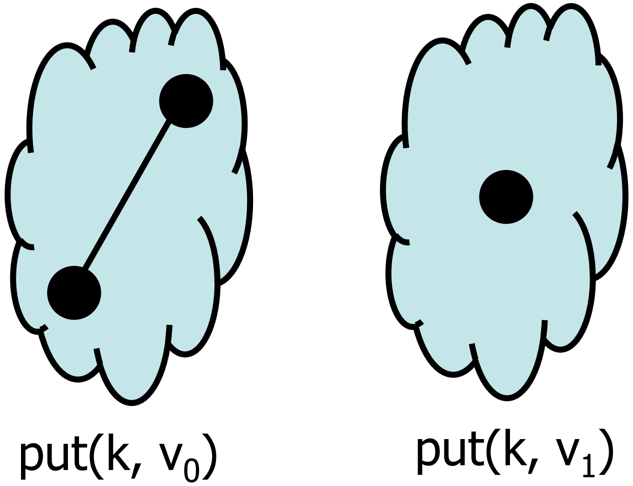
- Tell client write complete when only some replicas have stored it
- Propagate to other replicas in background
- Allows writes in both partitions…
- …but risks:
  - returning stale data
  - write conflicts when partition heals



put(k, v_0)

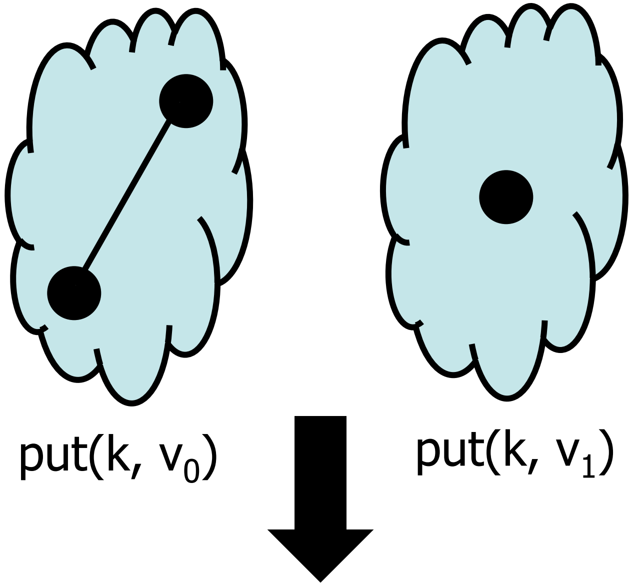
Alternative: Eventual Consistency

- Tell client write complete when only some replicas have stored it
- Propagate to other replicas in background
- Allows writes in both partitions…
- …but risks:
  - returning stale data
  - write conflicts when partition heals



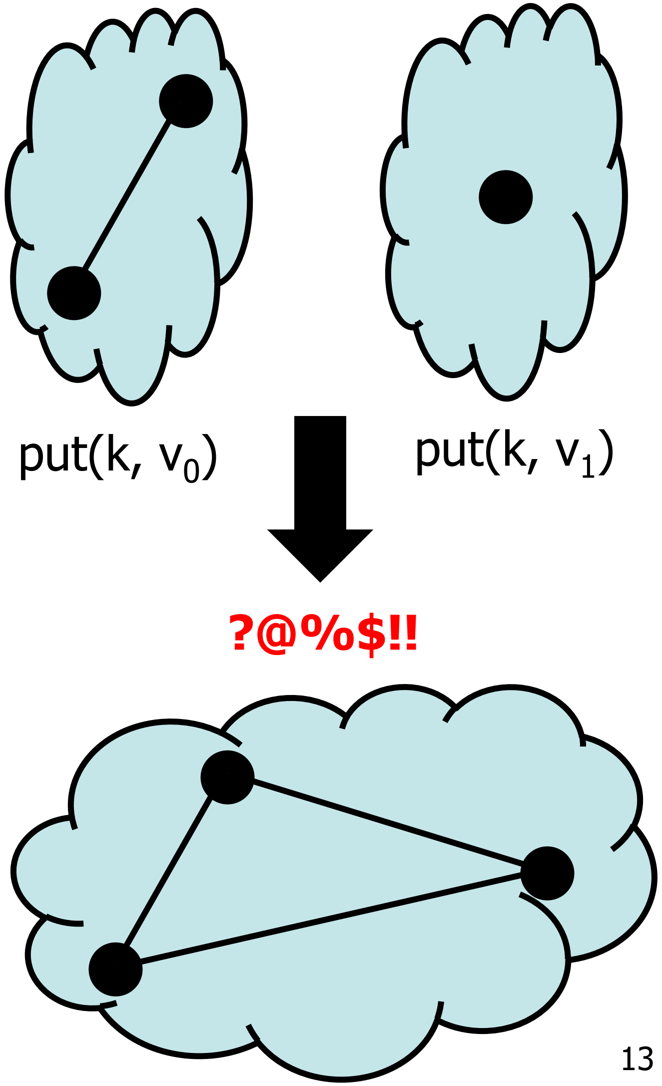
Alternative: Eventual Consistency

- Tell client write complete when only some replicas have stored it
- Propagate to other replicas in background
- Allows writes in both partitions…
- …but risks:
  - returning stale data
  - write conflicts when partition heals



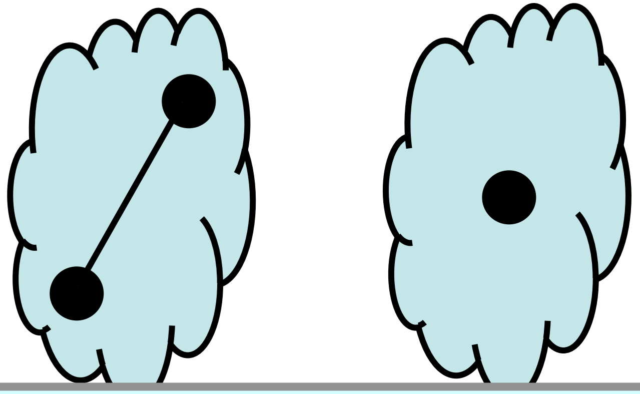
Alternative: Eventual Consistency

- Tell client write complete when only some replicas have stored it
- Propagate to other replicas in background
- Allows writes in both partitions…
- …but risks:
  - returning stale data
  - write conflicts when partition heals



Alternative: Eventual Consistency

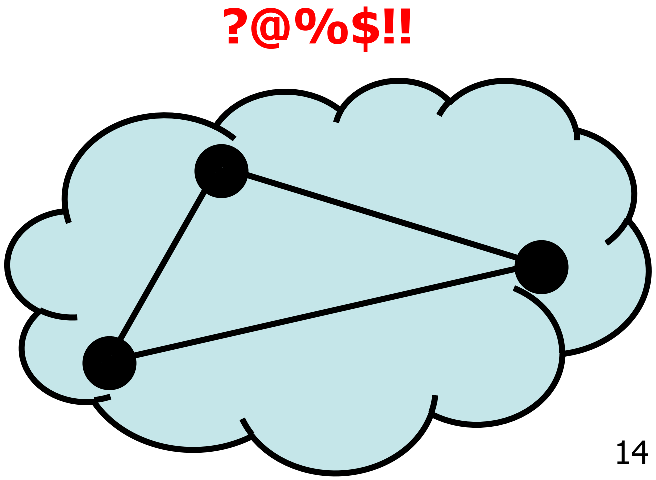
- Tell client write complete when only some replicas have stored it



- Propagate to other

Dynamo emphasizes **availability** over consistency when there are partitions

partitions…



- …but risks:
  - returning stale data
  - write conflicts when partition heals

Dynamo’s Interface

- Keys and values opaque to Dynamo
- Dynamo hashes keys into 128-bit identifiers with MD-5
- 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 value versions this new value version derived from

Consistent Hashing in Dynamo

- Much like in Chord
- (k, v) pair stored at k’s successors (named “preference list” in paper)
- Each physical node acts as multiple virtual nodes, each with own place on ring
  - When physical node fails, many other physical nodes handle its load
  - When physical node added, takes load from many other physical nodes
  - Physical nodes of heterogeneous capacity can host different numbers of virtual nodes

Gossip and “Lookup”

- To add node, administrator explicitly informs existing node of new node’s address
- Once per second, each node contacts random other node; they exchange their lists of known nodes (including virtual node IDs)
- Each node learns which other nodes handle all key ranges
- Result: all nodes can send directly to any key’s successor

Data Replication

- Successor node of key k named “coordinator”
- Nodes send put(k,…) to k’s coordinator
- k’s coordinator replicates to preference list
- Goal: each (k, v) pair replicated at N nodes
- Preference list longer than N to allow for failed nodes

Consistency Model: “Sloppy Quorums”

- Goal: want it to be likely that get()s see most recent put()s
- Goal: don’t want to block get()s or put()s from completing during partitions
- Dynamo tries to store all values put() under k on first N live nodes of coordinator’s preference list
- Coordinator replies “success” for put() when only W replicas have completed write
- Coordinator replies “success” for get() when only R replicas have completed read
- R < N and W < N to make get(), put() fast

Sloppy Quorums and put()s

- Suppose coordinator doesn’t receive W replies when replicating a put()
- Could return failure, but remember goal of high availability for writes…
- Hinted handoff: coordinator tries next successors on ring (beyond preference list) if necessary
  - Indicates to recipient correct replica node
  - Recipient will periodically try to forward to correct replica node

Wide-Area Replication

- Last paragraph in Section 4.6 states that preference lists always contain nodes from more than one data center
- Consequence: data likely to survive failure of entire data center
- Synchronously waiting for writes to remote data center would incur unacceptably high latency
- Compromise: $W < N$, eventual consistency

Sloppy Quorums and get()s

- Suppose coordinator doesn’t receive R replies when processing a get()
- p. 211: “R is the minimum number of nodes that must participate in a successful read operation.” (Sounds like these get()s fail.)
- Why not return whatever data was found, though? As we will see, consistency not guaranteed anyway…

Sloppy Quorums and Freshness

- Common case given in paper:
  N = 3, R = 2, W = 2
- With these values, do sloppy quorums guarantee a get() sees all prior put()s?
- If no failures, yes:
  - Two writers saw each put()
  - Two readers responded to each get()
  - Write and read quorums must overlap!
- If failures, no:
  - Two nodes in preference list go down; put() replicated outside preference list
  - Two nodes in preference list come back up; get() occurs before they receive prior put()

Sloppy Quorums and Freshness

- Common case given in paper:
  N = 3, R = 2, W = 2
- With these values, do sloppy quorums guarantee a get() sees all prior put()s?
- If no failures, yes:
  - Two writers saw each put()

Sloppy quorums increase probability get()s observe recent put()s, but **no guarantee**

- If failures, no:
  - Two nodes in preference list go down; put() replicated outside preference list
  - Two nodes in preference list come back up; get() occurs before they receive prior put()

Conflicts

- Suppose N = 3, W = 2, R = 2, and nodes are named A, B, C
- First put(k, …) complete on A, B
- Second put(k, …) complete on B, C
- Now get(k) arrives, completes first at A, C
- Conflicting results from A and C: each has seen different put(k, …)
- Dynamo returns both results
- What does client do now?

Conflicts vs. Applications

- Shopping cart:
  - Could take union of two results
  - What if second put() was result of user deleting item from cart stored in first put()?
  - Result: “resurrection” of deleted item
- Can we do better? Can Dynamo resolve cases when multiple values are found?
  - Sometimes. If it can’t, application must do so.

Version Vectors ("Vector Clocks")

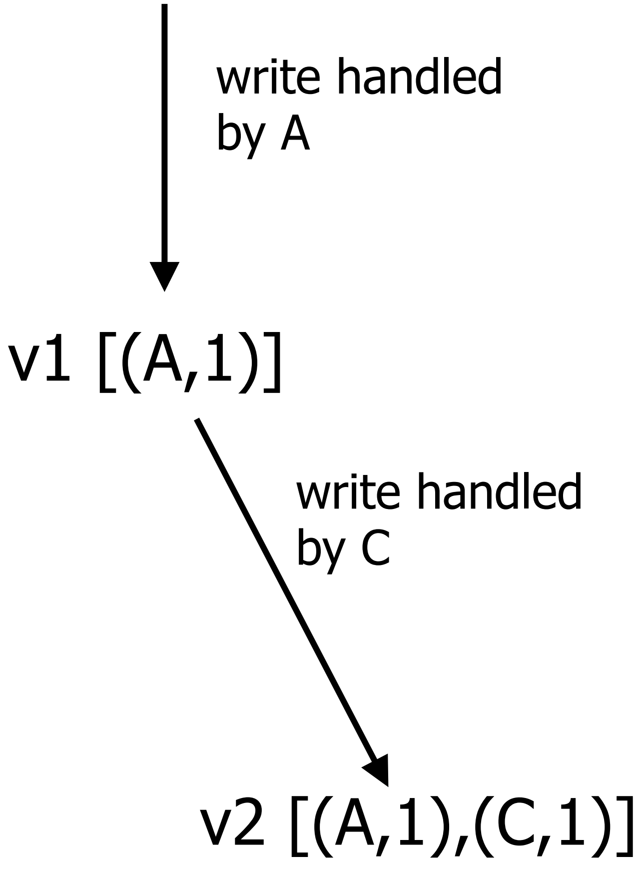
- Version vector: list of (node, counter) pairs, e.g., [(A, 1), (B, 3), …]
- Dynamo stores version vector with each stored (k, v) pair
- Idea: track "ancestor/descendant" relationship between different versions of data stored under the same key k

Version Vectors (cont'd)

- Rule: given two versions, each with a VV, if each counter of the first version is less than or equal to that of the second, then the first is an ancestor of the second and can be forgotten by Dynamo
- Each time a put() occurs, Dynamo increments the counter in the VV for the coordinator node
- Each time a get() occurs, Dynamo returns the VV for the value(s) returned (in the "context")
- When get()ting a value, modifying it, and put()ting it again under the same key, user must supply the context Dynamo provided in the result of the get()

Version Vectors (Auto-Resolving Example)

- put() handled by A is version v1, stamped with VV [(A,1)]
- put() updating same k handled by C is version v2, stamped with VV [(A,1),(C,1)]
- get(k) retrieved from A and C returns only v2



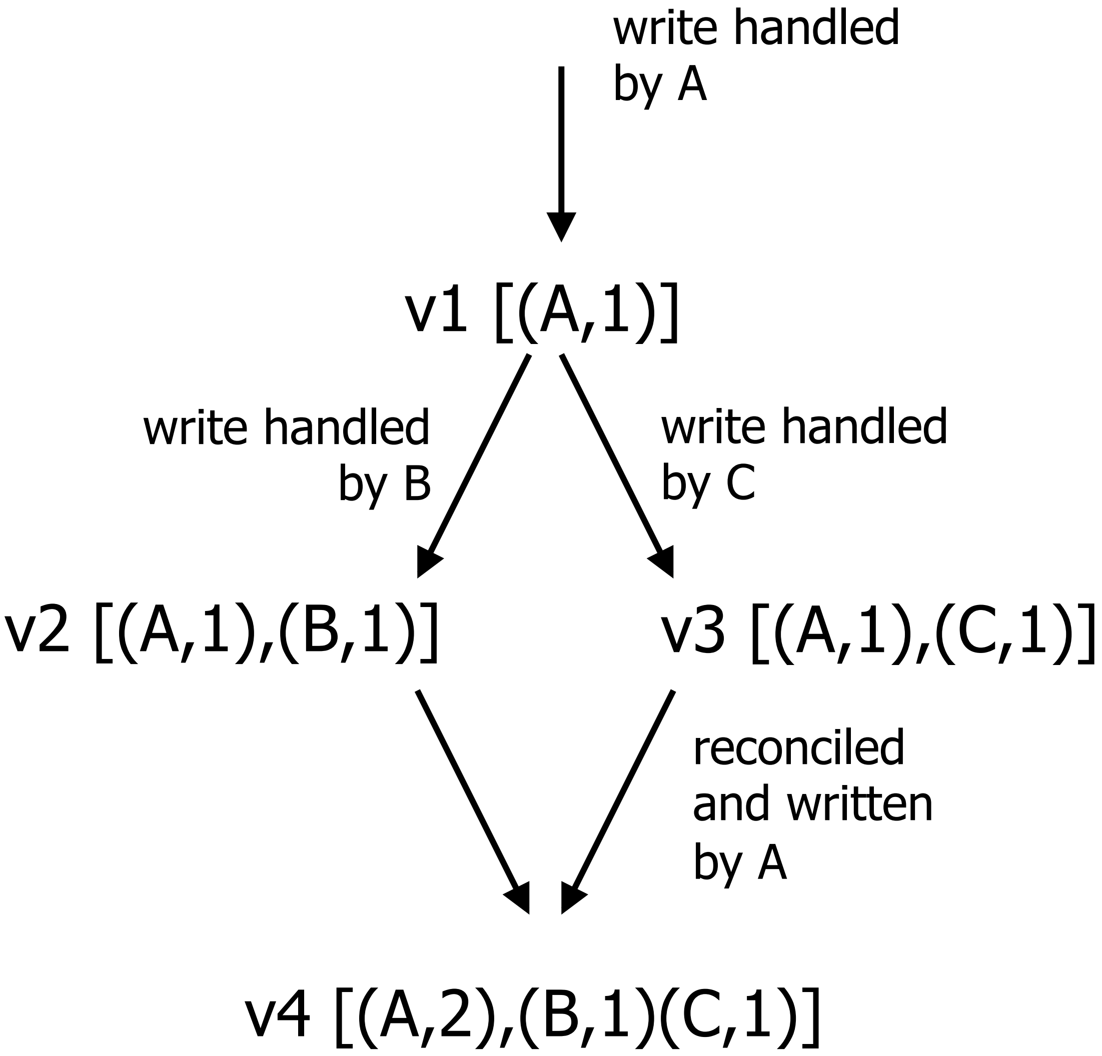
write handled by A

v1 [(A,1)]

write handled by C

v2 [(A,1),(C,1)]

Version Vectors (Application-Resolving Example)



write handled by A

v1 [(A,1)]

write handled by B

write handled by C

v2 [(A,1),(B,1)]

v3 [(A,1),(C,1)]

reconciled and written by A

v4 [(A,2),(B,1)(C,1)]

Limitations of Application-Based Reconciliation

- Suppose two clients wish to increment the same counter concurrently (stored under the same key k)
- Each will independently read the same value
- Each will locally modify it
- Each will write back
- A subsequent reader will see two instances of the same value…no sensible way to identify two increments occurred!

Trimming Version Vectors

- Many nodes may process a series of put()s to same key; VVs may get long
- Must they grow forever?
- Dynamo stores time of modification with each entry in VV
- When VV longer than 10 hosts long, VV drops the timestamp of host that least recently processed that key
- Avoids passing huge VVs around
- Conservative: might force client to do reconciliation, as loses state about a value’s ancestry

Maintaining Replication: Node-to-Node Reconciliation

- A node holding data received via “hinted handoff” may crash before it can pass data to unavailable node in preference list
- Need another way to ensure each (k, v) pair replicated N times
- Nodes nearby on ring periodically compare the (k, v) pairs they hold, and copy any they are missing that are held by the other

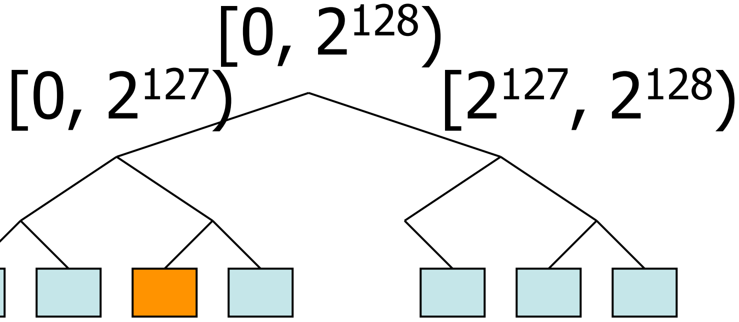
Efficient Reconciliation: Merkle Trees

- Idea: hierarchically summarize the (k, v) pairs a node holds by ranges of keys
- Leaf node: hash of one (k, v) pair
- Internal node: hash of concatenation of children
- Compare roots; if match, done
- If don’t match, compare children; recur…

Merkle Tree Reconciliation: Example

- B is missing orange key; A is missing green one
- Compare nodes from root downwards, pruning when hashes match

A' s values:

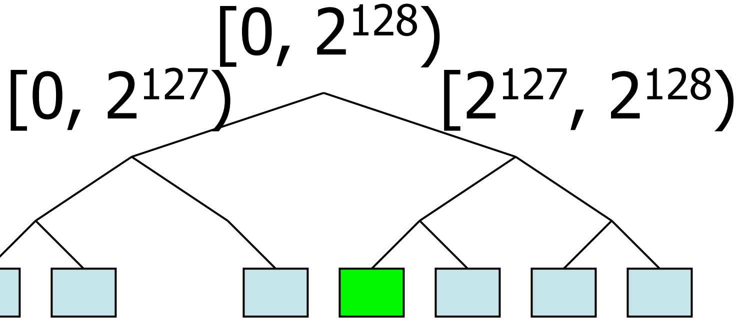


$[0, 2^{128})$

$[0, 2^{127})$

$[2^{127}, 2^{128})$

B' s values:



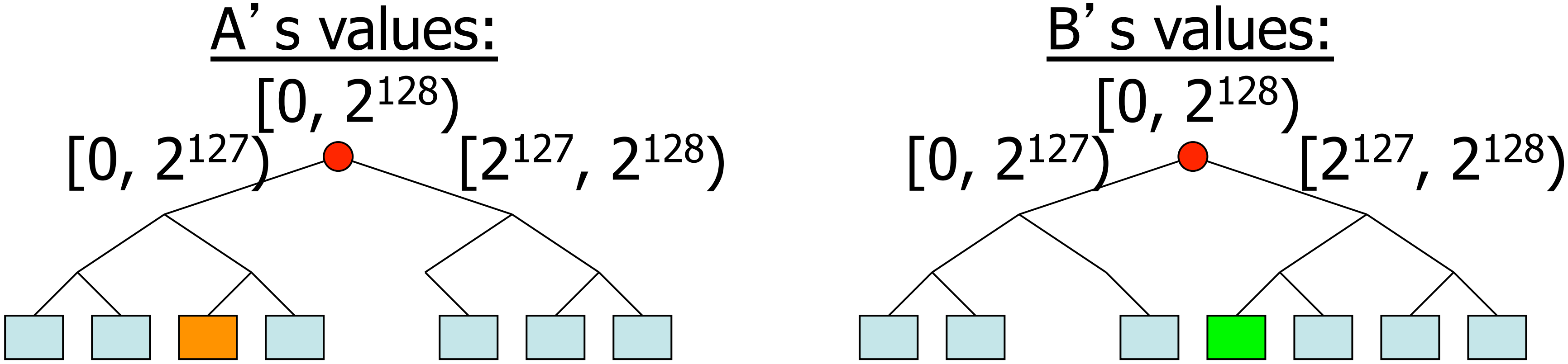
$[0, 2^{128})$

$[0, 2^{127})$

$[2^{127}, 2^{128})$

Merkle Tree Reconciliation: Example

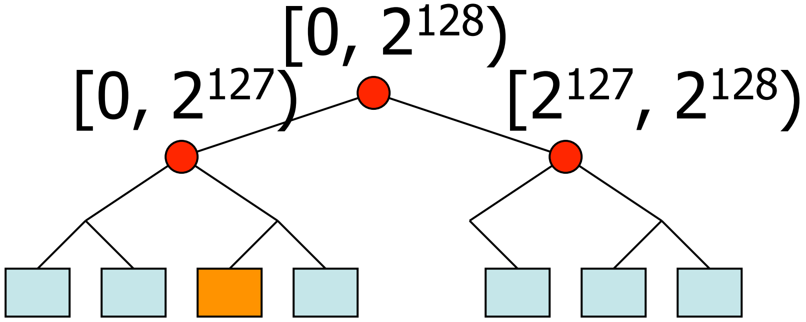
- B is missing orange key; A is missing green one
- Compare nodes from root downwards, pruning when hashes match



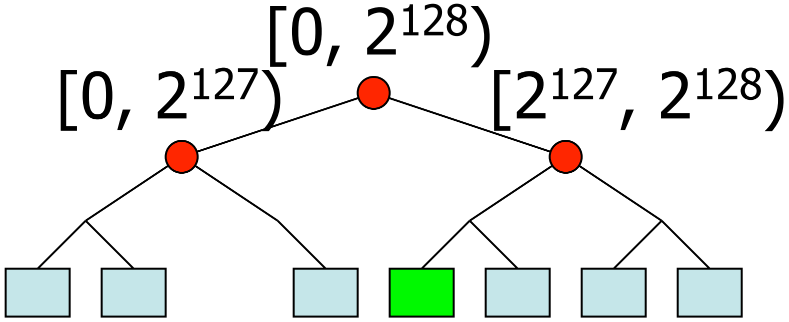
Merkle Tree Reconciliation: Example

- B is missing orange key; A is missing green one
- Compare nodes from root downwards, pruning when hashes match

A’s values:



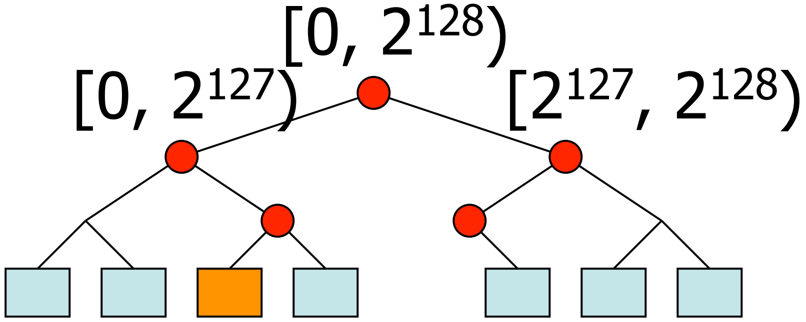
B’s values:



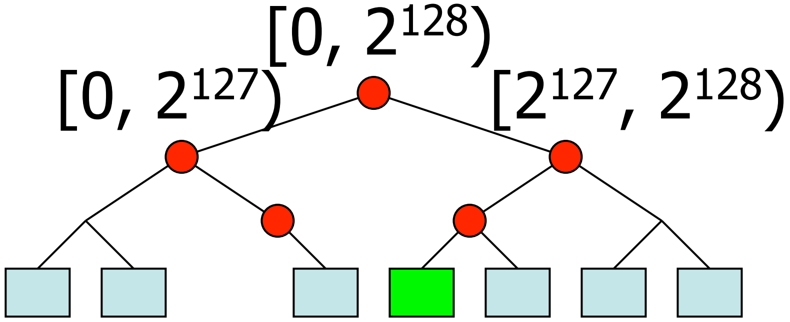
Merkle Tree Reconciliation: Example

- B is missing orange key; A is missing green one
- Compare nodes from root downwards, pruning when hashes match

A’s values:



B’s values:



How General Is Dynamo's Consistency Model?

- Doesn't support, e.g., concurrent increments…or many other operations
- Amazon's stated uses:
  - Shopping cart (merge carts, resurrected deleted items)
  - Session information for users
  - Product list (but probably nearly read-only, so R = 1, W = N would work well)

How Useful Is It to Vary N, R, W?

N	R	W	Behavior
3	2	2	Parameters from paper: good durability, good R/W latency
3	3	1	Slow reads, weak durability, fast writes
3	1	3	Slow writes, strong durability, fast reads
3	3	3	More likely that reads see all prior writes?
3	1	1	Doesn’t make sense. (Why not?)

Dynamo: Summary

- Consistent hashing broadly useful for replication—not only in P2P systems
- Extreme emphasis on availability and low latency, unusually, at the cost of some inconsistency
- Eventual consistency lets writes and reads return quickly, even when partitions and failures
- Version vectors allow some conflicts to be resolved automatically; others left to application
- Seems to meet Amazon's specific applications' consistency needs…
- …but definitely not right for all applications