7 Distributed Key-Value-Stores

> Distributed Key/Value Stores

Observation

- Many applications do not need a query language
- Instead primary key access only
- Restriction of functionality enables: performance / scalability / maintenance

Approach: Scalable, distributed storage engine

Examples

- Yahoo! PNUTS
- **Amazon Dynamo**
- Google BigTable

- Cassandra (Facebook)
- Voldemort (LinkedIn)

[Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, Ramana Yerneni: PNUTS: Yahoo!'s hosted data serving platform. PVLDB 1(2):1277-1288 (2008)]

[Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, Werner Vogels: Dynamo: amazon's highly available key-value store. SOSP 2007:205-220]

[Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, Robert Gruber: Bigtable: A Distributed Storage System for Structured Data. OSDI 2006:205-218]

Amazon Dynamo

> Motivation

Build a highly decentralized distributed system to provide reliability

Features

- Scalable
- Loosely coupled
- Highly Available

Performance+Efficiency+Reliability=$$$

- Downtime effects
  - Financial Loss
  - Impacts Customer Trust
- Examples
  - Amazon found every 100 ms of latency cost them 1% in sales
  - Google found an extra 0.5 seconds in search page generation time dropped traffic by 20%

> System Assumptions & Requirements

Query Model

- Simple Read/Write operations to a data item (binary objects)
- Uniquely identified by a key

Eventual consistency

- Results in higher availability (CAP theorem)
- No isolation guarantees

Efficiency

- High latency requirements which are in general measured at the 99.9th percentile of the distribution

Scale

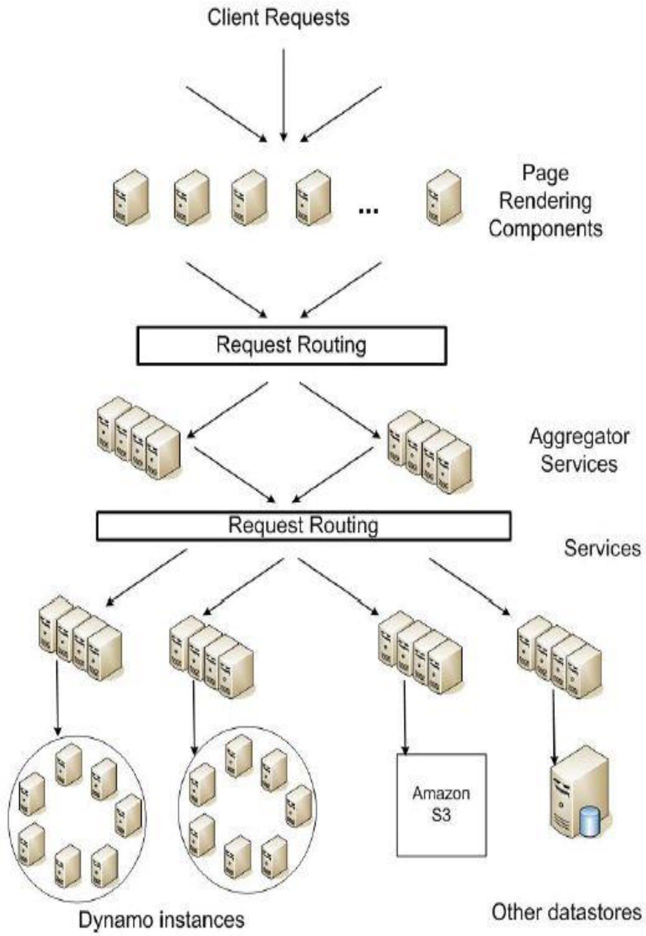
- Dynamo is designed for “hundreds of storage nodes” (but not more), assuming a distinct Dynamo instance for each service.

Others

- Non-hostile environment, No need for security mechanisms like authorization & authentication

> Service Level Agreements (SLA)

- SLA Guarantees
  - Application delivers its functionality in a bounded time
- Clients and services agree on several system-related characteristics as shown
- Each service initializes a distinct instance of Dynamo
- Example
  - Guarantee response within 300 ms for 99.9% of its requests for a 500 request per second instance



> Design Considerations

Sacrifice strong consistency for high data availability

Eventually consistent data using replication algorithms

"Always writeable" data store

- Return from update before all replicas are updated → to satisfy SLA
- Propagate updates to replicas when connectivity is reestablished
- Conflict resolution is executed during read instead of write

Other implementation principles

- Incremental Scalability: Simply add new storage nodes
- Symmetry: All nodes have the same responsibilities (simplifies provisioning and maintenance)
- Decentralization: No centralized control - no master!
- Heterogeneity: Cope with different hardware (generations)

> Techniques used in Dynamo

Problem	Technique	Advantage
Partitioning	Consistent Hashing	Incremental Scalability
High Availability for writes	Vector clocks with reconciliation during reads	Version size is decoupled from update rates
Handling temporary failures	Sloppy Quorum and hinted handoff	Provides high availability and durability guarantee when some of the replicas are not available
Recovering from permanent failures	Anti-entropy using Merkle trees	Synchronizes divergent replicas in the background
Membership and failure detection	Gossip-based membership protocol and failure detection	Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information

> Partitioning Algorithm

Approach

- Dynamically partition the data over the set of storage nodes
- Relies on consistent hashing to distribute load across multiple nodes and hash function is treated as fixed ring

Consistent Hashing

- Each node is identified by an ID uniformly distributed in range [0, 1]
- The ID (hash) of each key is uniformly distributed within the same range [0, 1]
- A page is stored to the closest cache in the ID space
- Departure or arrival of a node only affects its immediate neighbors
- Other nodes remain unaffected

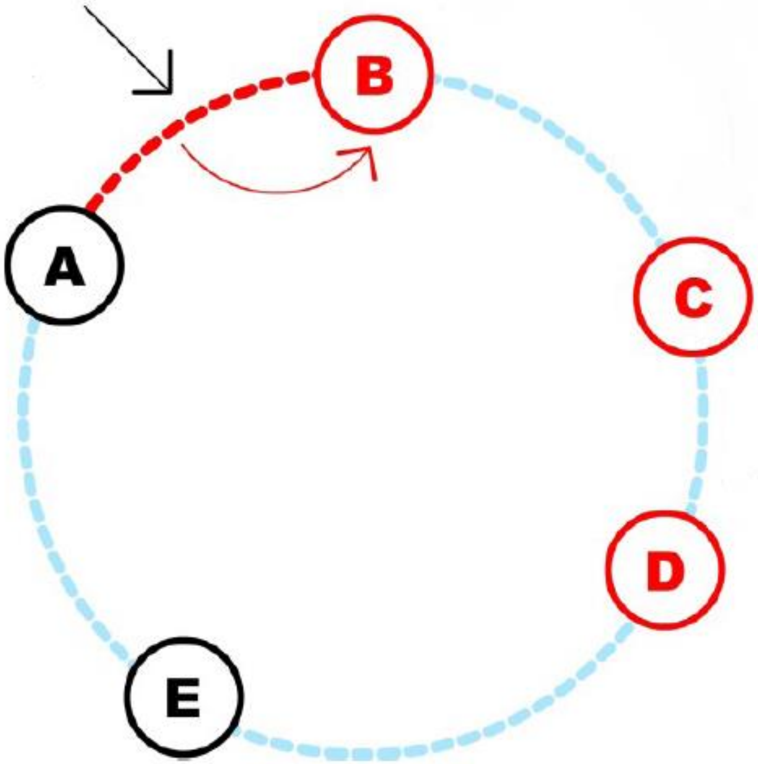
Problems

- Random position assignment of each node on the ring leads to non-uniform data and load distribution (in particular, after failure /during recovery)
- Basic algorithm is oblivious to the heterogeneity in the performance of nodes

> Partitioning algorithm (2)

Example

Key k



> Partitioning algorithm (3)

Virtual Nodes

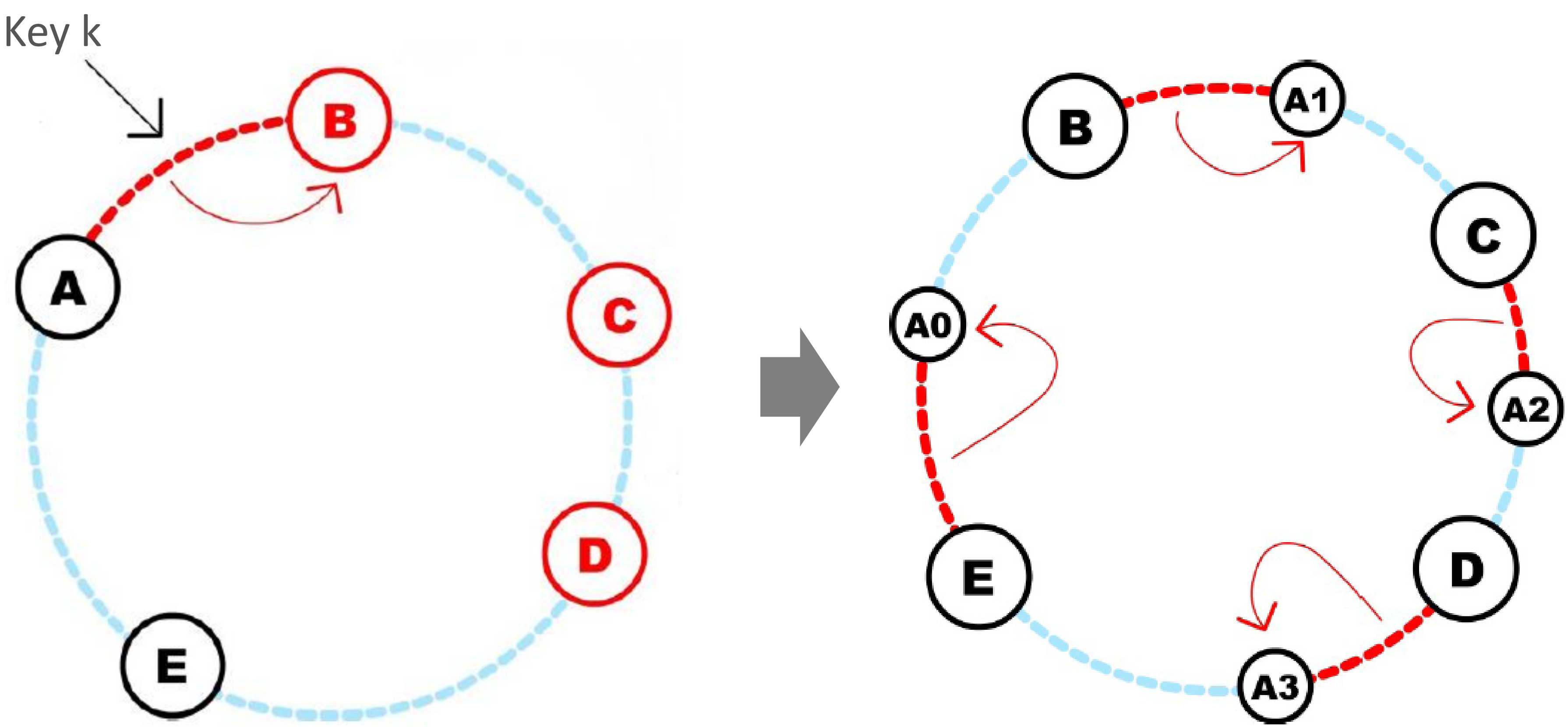
- Instead of mapping a node to a single point in the circle, each node gets assigned to multiple points in the ring
- **Virtual node** looks like a single node in the system, but each node can be responsible for more than one virtual node
- When a new node is added to the system, it is assigned multiple positions in the ring

Advantages

- If a node becomes unavailable the load handled by this node is evenly dispersed across the remaining available nodes
- When a node becomes available again, it accepts a roughly equivalent amount of load from each of the other available nodes
- The number of virtual nodes that a node is responsible can decided based on its capacity, accounting for heterogeneity in the physical infrastructure

> Partitioning algorithm (4)

Example



> Replication

Design Goal

- High availability and durability

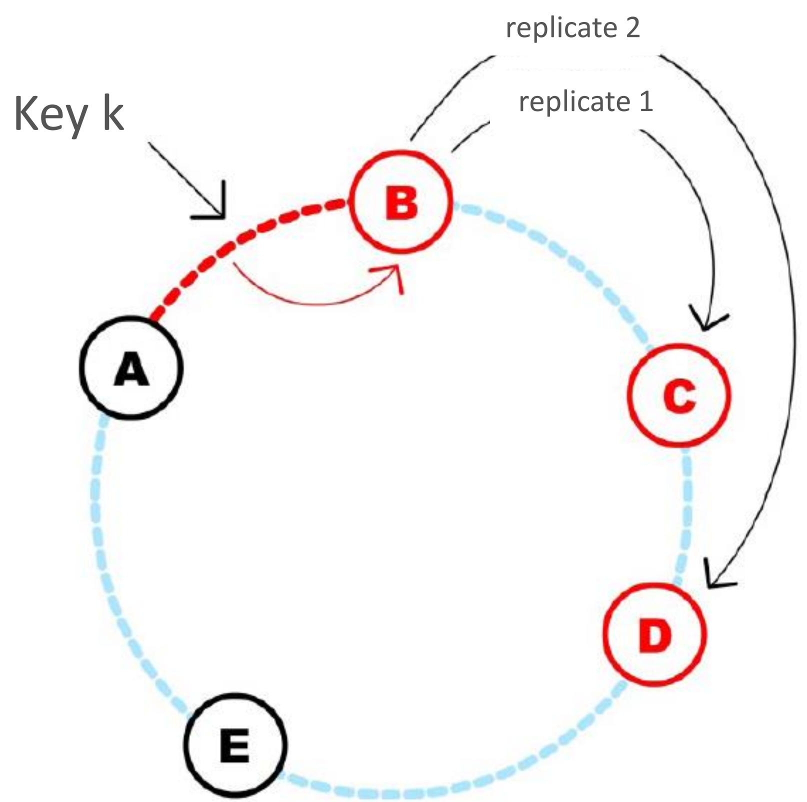
Approach

- Data is replicated among N hosts (N is a parameter configured "per-instance")
- By consistent hashing, each data is assigned to a coordinator node by the consistent hash function
- The coordinator then replicates the data item at the N-1 successor virtual hosts, but leaves those out that run on the same physical host
- The list of virtual hosts responsible for storing a particular key is called the **preference list**
- To account for host failures the preference list contains more than N virtual hosts
- Preference list is constructed in a way such that the physical nodes are spread across multiple data centers to ensure operation in the presence of network partitioning

> Replication (2)

Example (N=3)

- Node B replicates the key k at nodes C and D in addition to storing it locally
- Node D will store the keys that fall in the ranges (A, B], (B, C], and (C, D]



> Data Versioning

System characteristics

- Eventual consistency → asynchronous updates on replicas
- Get() and Put() suspect if the update is done or not
- If there are no failures then there is a bound on the update propagation times
- Under certain failure scenarios (e.g., server outages or network partitions), updates may not arrive at all replicas for an extended period of time

Shopping Cart Example

- “Add to Cart” operation should never be forgotten or rejected (→ “always writeable” data store)
- If the most recent state of the cart is unavailable, and a user makes changes to an older version of the cart, that change is still meaningful and should be preserved
- But at the same time it shouldn’t supersede the currently unavailable state of the cart, which itself may contain changes that should be preserved
- When a customer wants to add an item and the latest version is not available, the item is added to the older version and the divergent versions are reconciled later

> Data Versioning (2)

Approach

- Each modification creates a new version of the data → multiple versions of an object present in the system at the same time
- **Syntactic reconciliation**
  - Most of the time, new versions subsume the previous version(s), and the system itself can determine the authoritative version
- **Semantic reconciliation**
  - Version branching in the presence of failures combined with concurrent updates → conflicting versions of an object
  - Client must perform the reconciliation in order to collapse multiple branches of data evolution back into one

Example

- Merging different versions of a customer's shopping cart

Solution

- Uses **vector clocks** in order to capture causality between different versions of the same object

> Divergent Versions

Problem

- If several versions of the same object cannot be syntactically reconciled based on vector clocks alone → passed to the business logic for semantic reconciliation
- Semantic reconciliation introduces additional load on services, so it is desirable to minimize the need for it

Measured over 24 hour period for shopping cart

- 99.94% of users saw 1 version
- 0.00057% saw 2 versions
- 0.00047% saw 3 versions
- 0.00009% saw 4 versions

Experience shows that the increase in the number of divergent versions is contributed not by failures but due to the increase in number of concurrent writers (bots?)

Excursion: Vector Clocks

Vector Clocks

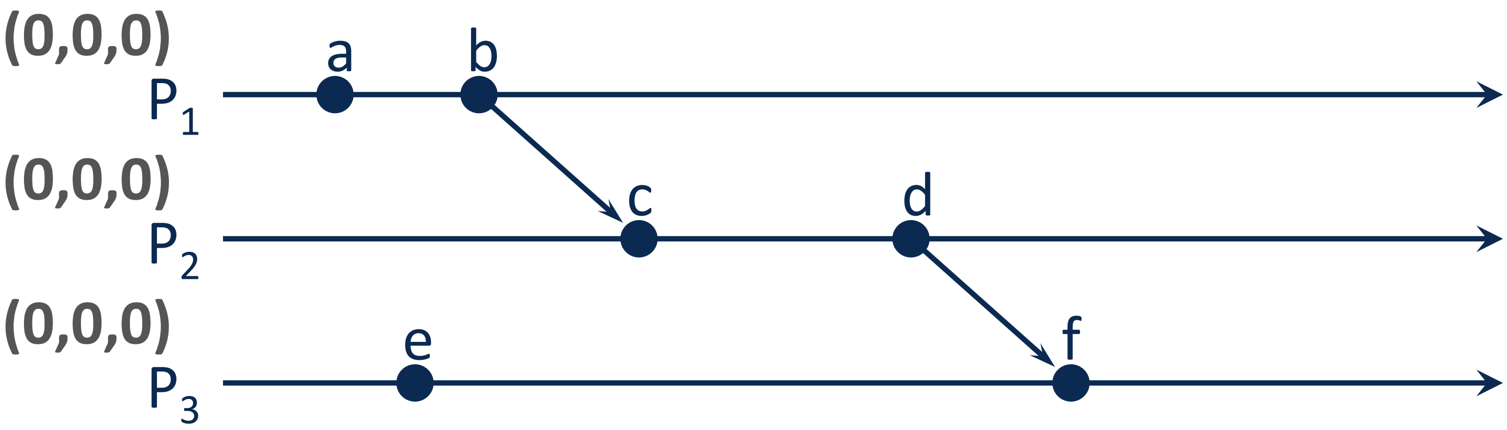
- Algorithm for generating a partial ordering of events in a distributed system and detecting causality violations
- A vector clock of a system of N processes (or nodes) is a vector of N logical clocks, one clock per process
- Rules
  - Vector initialized to 0 at each process
    $V_i [j] = 0$ for $i, j = 1, \ldots, N$
  - Process increments its element of the vector in local vector before timestamping event
    $V_i [i] = V_i [i] + 1$
  - Message is sent from process P_i with V_i attached to it
  - When P_j receives message, compares vectors element by element and sets local vector to the higher of two values
    $V_j [i] = \max(V_i [i], V_j [i])$ for $i=1, \ldots, N$

> Vector Clocks (2)

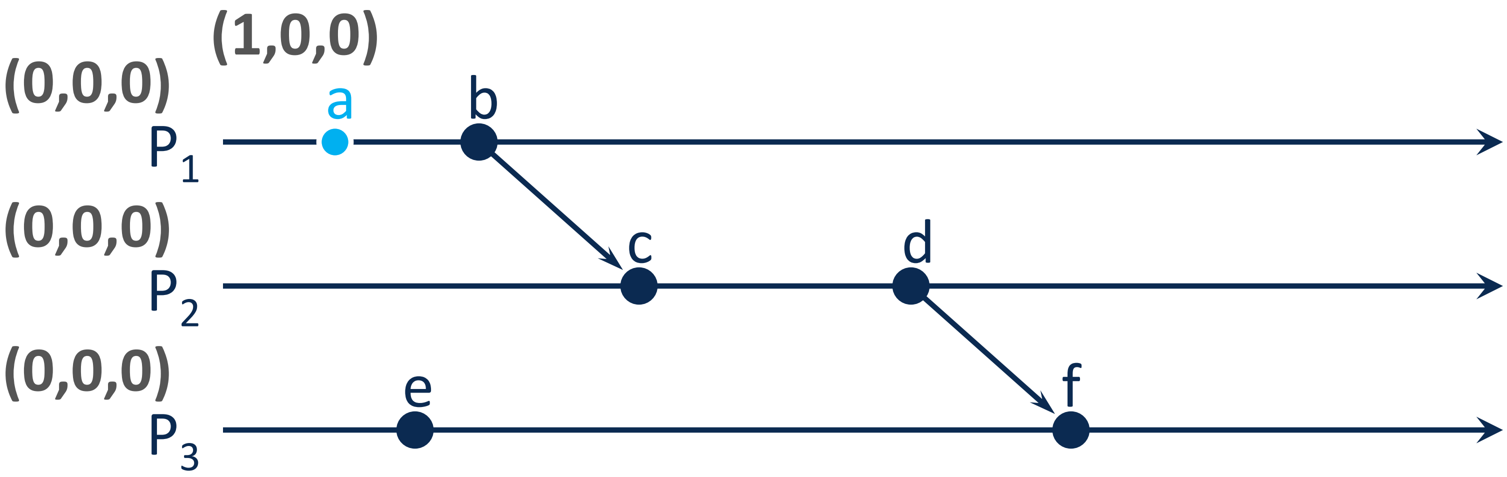
Comparing vector timestamps

- Define
  - V = V' iff V [i] = V'[i] for i = 1 … N
  - V $\leq$ V' iff V [i] $\leq$ V'[i] for i = 1 … N
- For any two events e, e'
  - if e $\rightarrow$ e' then V(e) < V(e')
  - if **V(e) < V(e') then e $\rightarrow$ e'**
- Events are concurrent when vector clocks are not comparable
  - V(e) $\leq$ V(e') nor V(e') $\leq$ V(e)
  - e.g., (2,1,0) | | (0,0,1)

> Vector Clocks (3) – Example

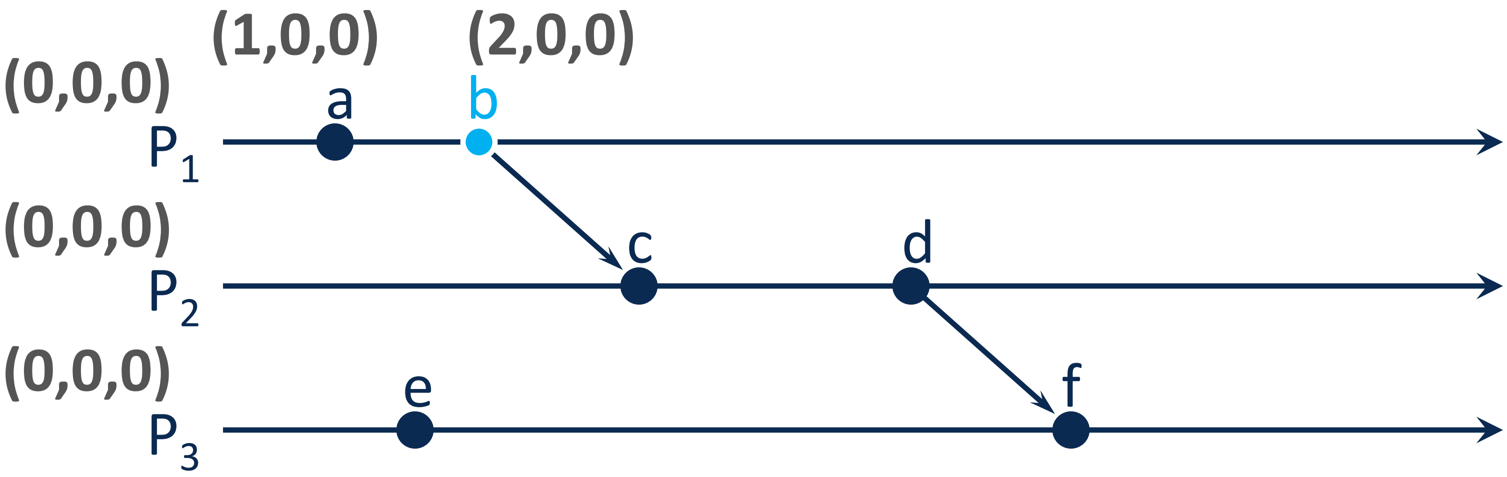


> Vector Clocks (4) – Example



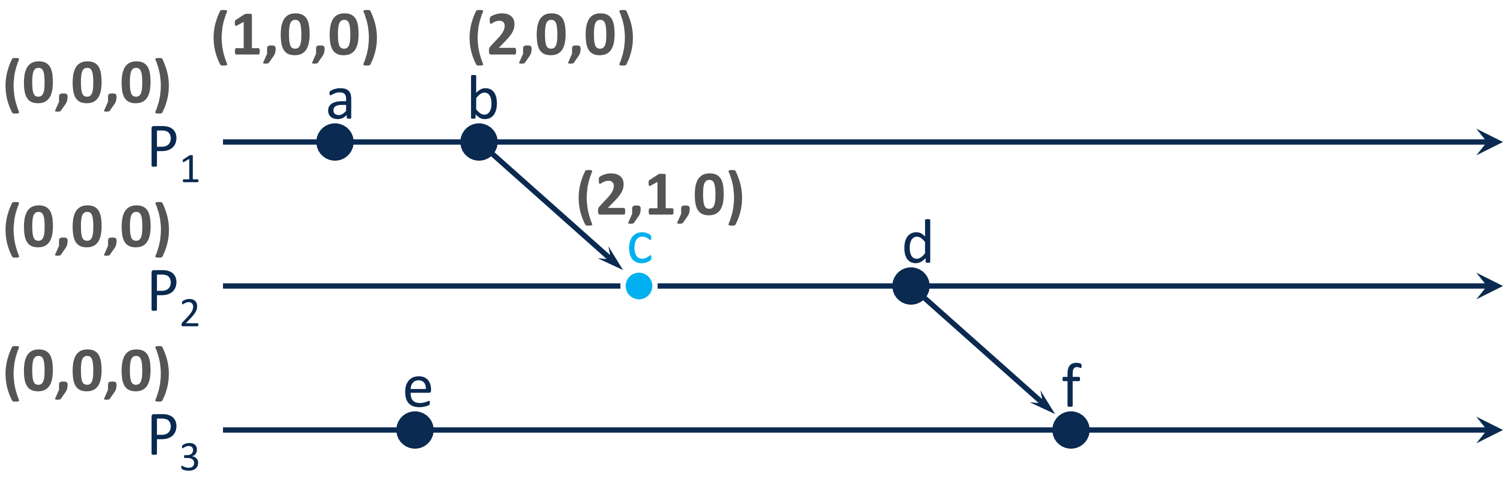
Event	timestamp
a	(1,0,0)

> Vector Clocks (5) – Example



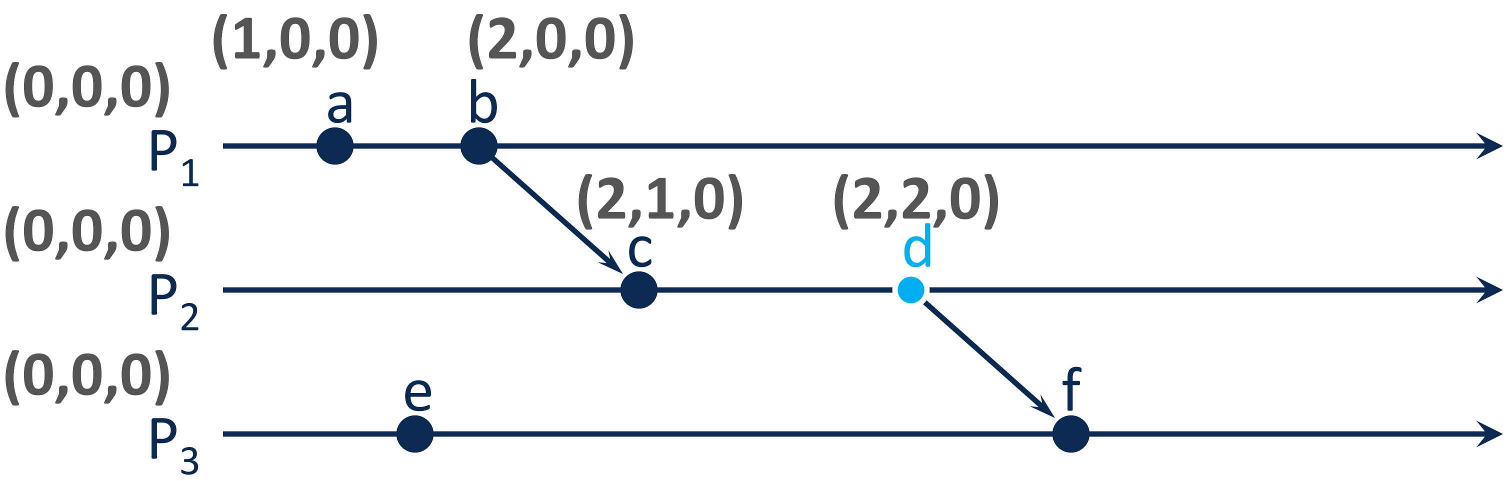
Event	timestamp
a	(1,0,0)
b	(2,0,0)

> Vector Clocks (6) – Example



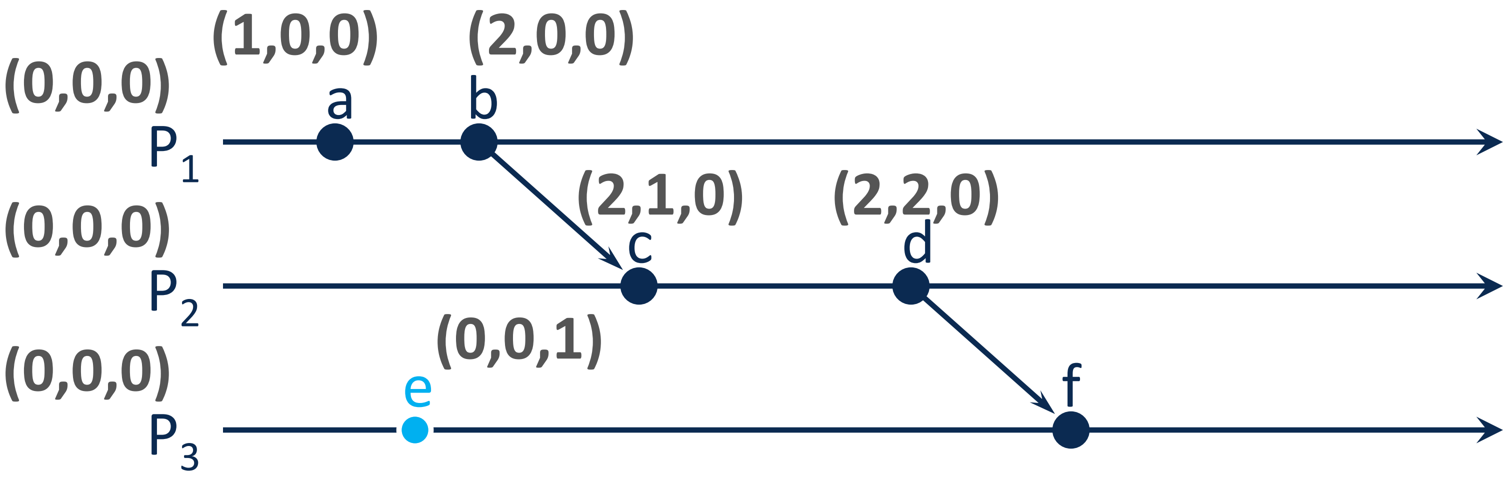
Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)

> Vector Clocks (7) – Example



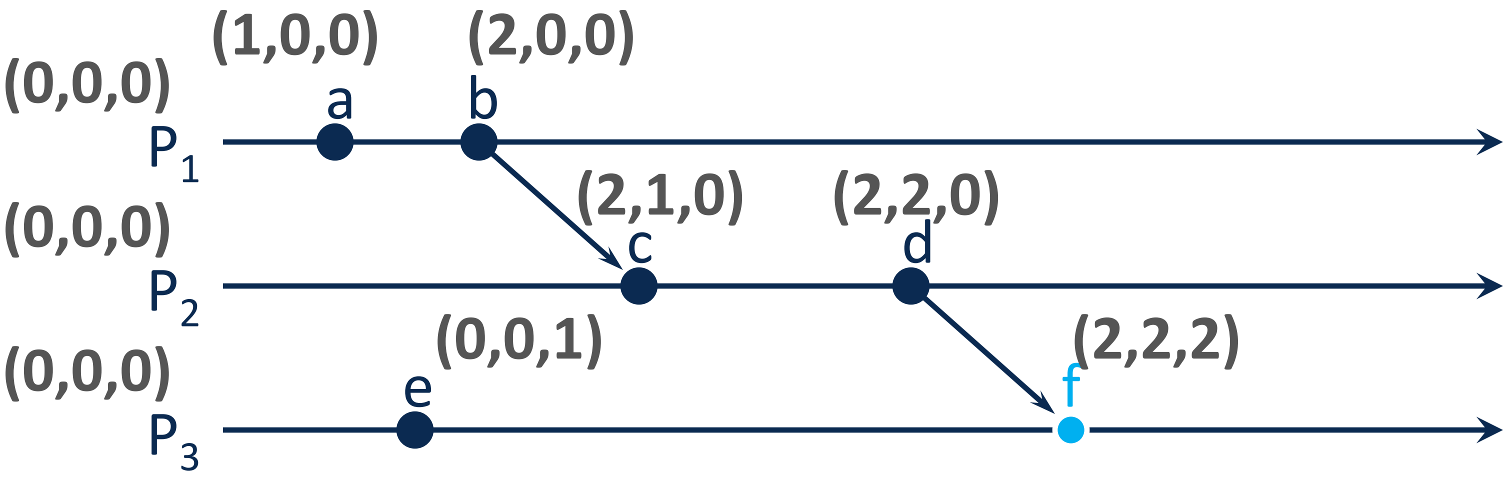
Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)
d	(2,2,0)

> Vector Clocks (8) – Example



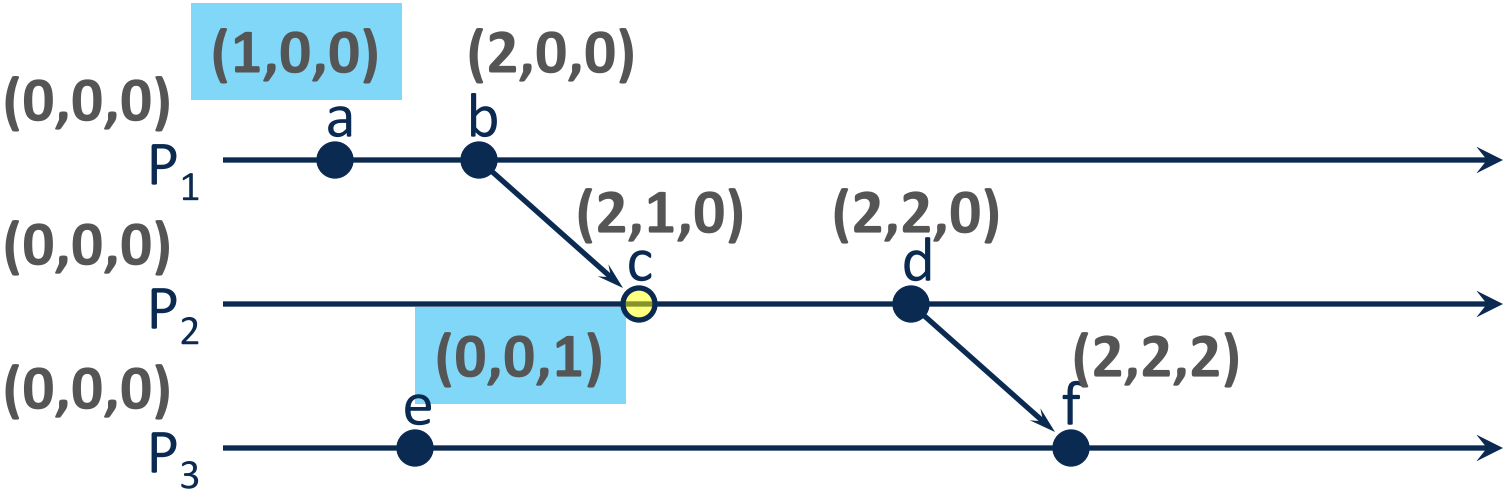
Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)
d	(2,2,0)
e	(0,0,1)

> Vector Clocks (9) – Example



Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)
d	(2,2,0)
e	(0,0,1)
f	(2,2,2)

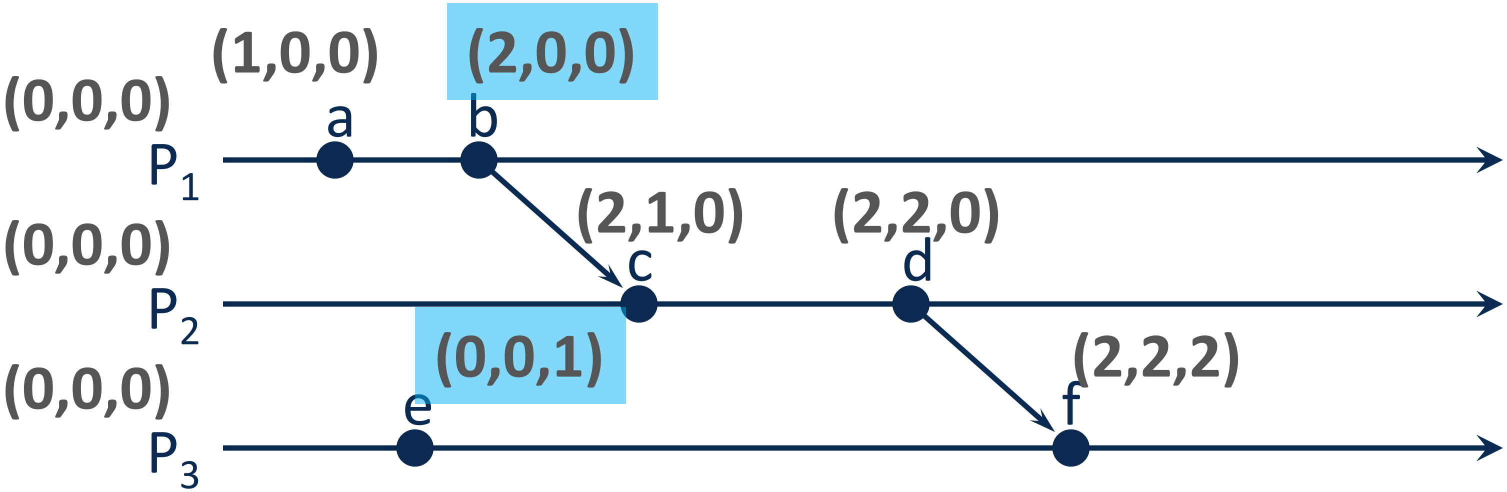
> Vector Clocks (10) – Example



Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)
d	(2,2,0)
e	(0,0,1)
f	(2,2,2)

concurrent events

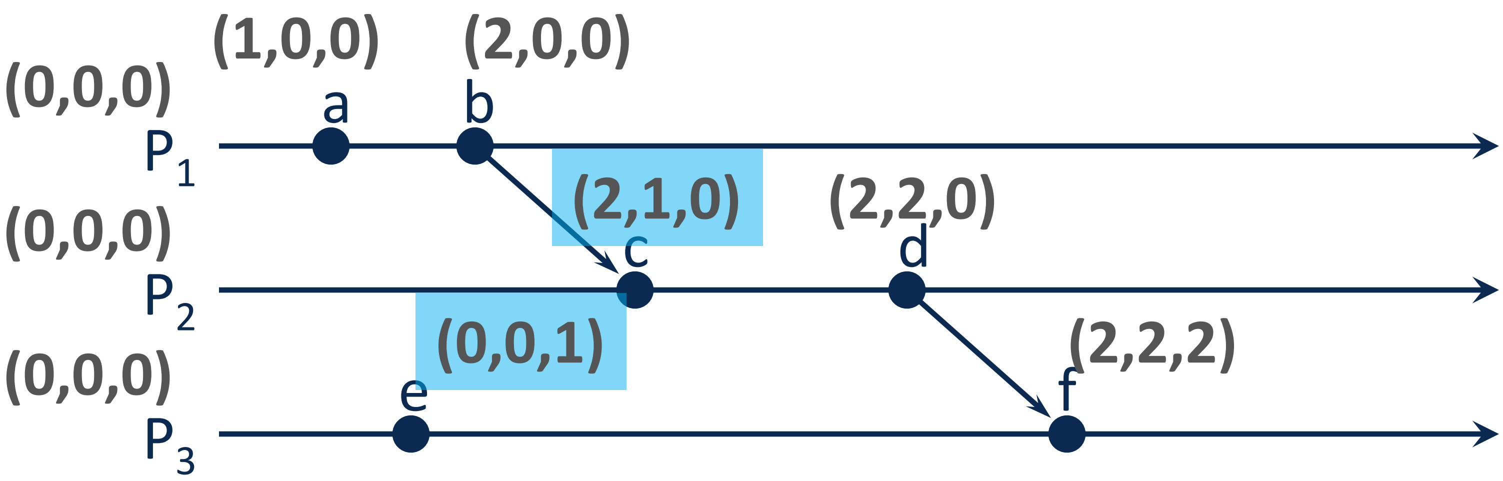
> Vector Clocks (11) – Example



Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)
d	(2,2,0)
e	(0,0,1)
f	(2,2,2)

concurrent events

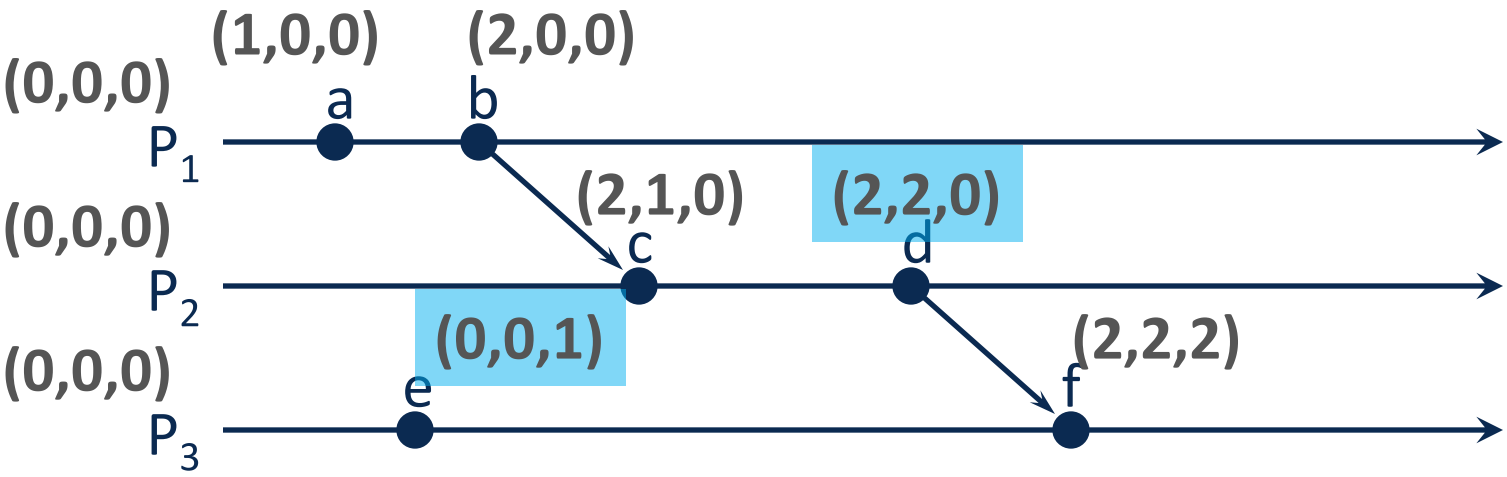
> Vector Clocks (12) – Example



Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)
d	(2,2,0)
e	(0,0,1)
f	(2,2,2)

concurrent events

> Vector Clocks (13) – Example



Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)
d	(2,2,0)
e	(0,0,1)
f	(2,2,2)

concurrent events

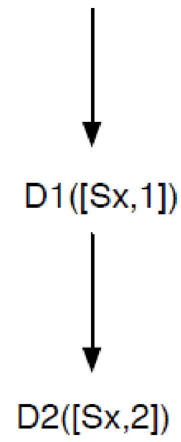
Vector Clocks in Dynamo

- Capture causality between different versions of the same object
- List of (node, counter) pairs → One vector clock is associated with every version of every object
- Determine whether **two versions** of an object are on **parallel branches** or have a **causal ordering**, by comparing their vector clocks
  - Counters on the first object's clock are less-than-or-equal to all of the nodes in the second clock, then the first is an ancestor of the second and can be forgotten
  - Otherwise, the two changes are considered to be in conflict and require **reconciliation**
- A read operation now might collect different versions
  - Through syntactic reconciliation, Dynamo tries to determine a causal ordering
  - If this is not possible, it returns a set of different versions (leaves of a versioning tree)
  - An update of the requested object (determined by the context) is considered to be a reconciliation of the different version collapsed into a single new version

> Vector Clocks in Dynamo (2)

Example

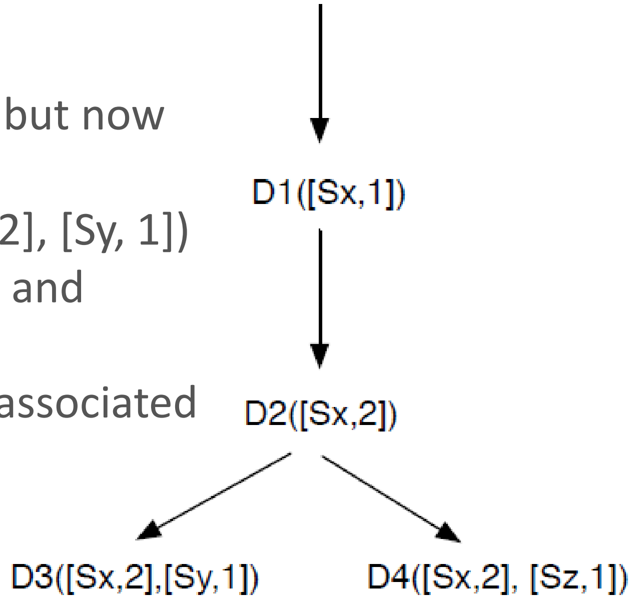
- Client **C1** writes a new object (D1)
- Node **Sx** coordinates the write for the given key and generates a vector clock
- C1 updates (overwrites) the object (to D2), and again node Sx coordinates the write
- The associated clock is set to ([Sx, 2])
- Note that there still might be replicas of D1 on other systems
- If a different node now handled a read request for the same object, then Dynamo could use syntactic reconciliation to determine that D2 descends from D1 (if both copies are within the read set)



> Vector Clocks in Dynamo (3)

Example

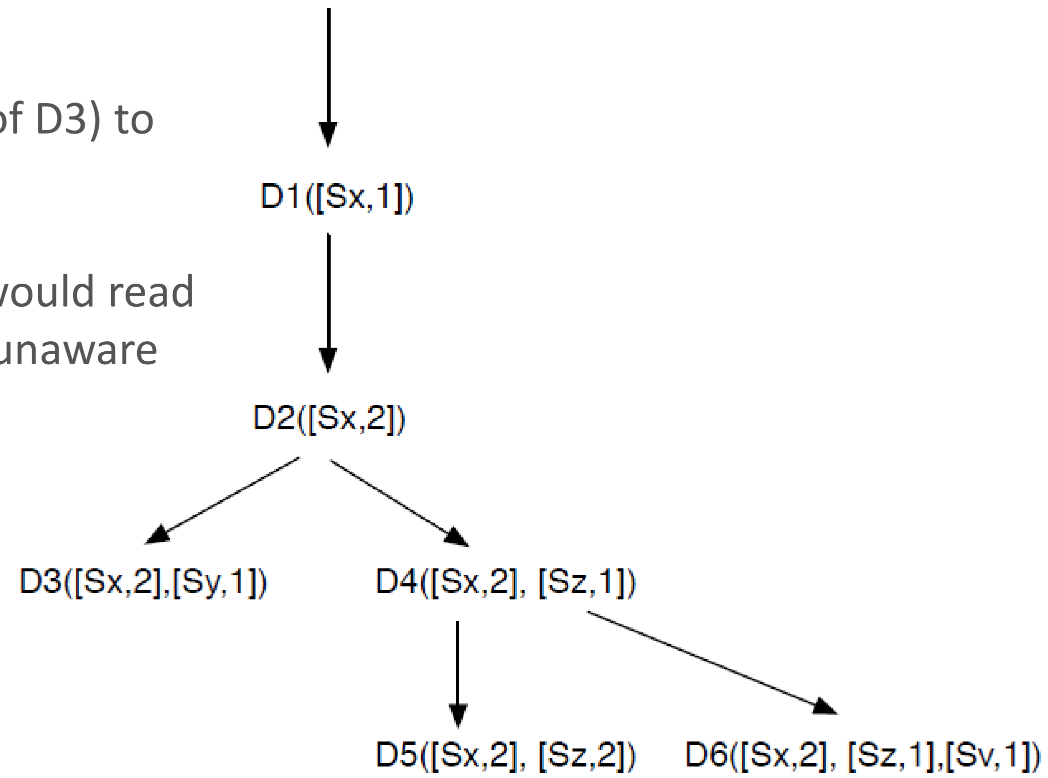
- **C1** updates the object again (to D3), but now on a different node **Sy**
- Sy now sets the vector clock to ([Sx, 2], [Sy, 1])
- At the same time, client **C2** reads D2 and updates it (to D4)
- **Sz** handles the request and sets the associated vector clock to ([Sx, 2], [Sz, 1])
- At that point, D3 and D4 are from different branches and syntactic reconciliation would not suffice



> Vector Clocks in Dynamo (4)

Example

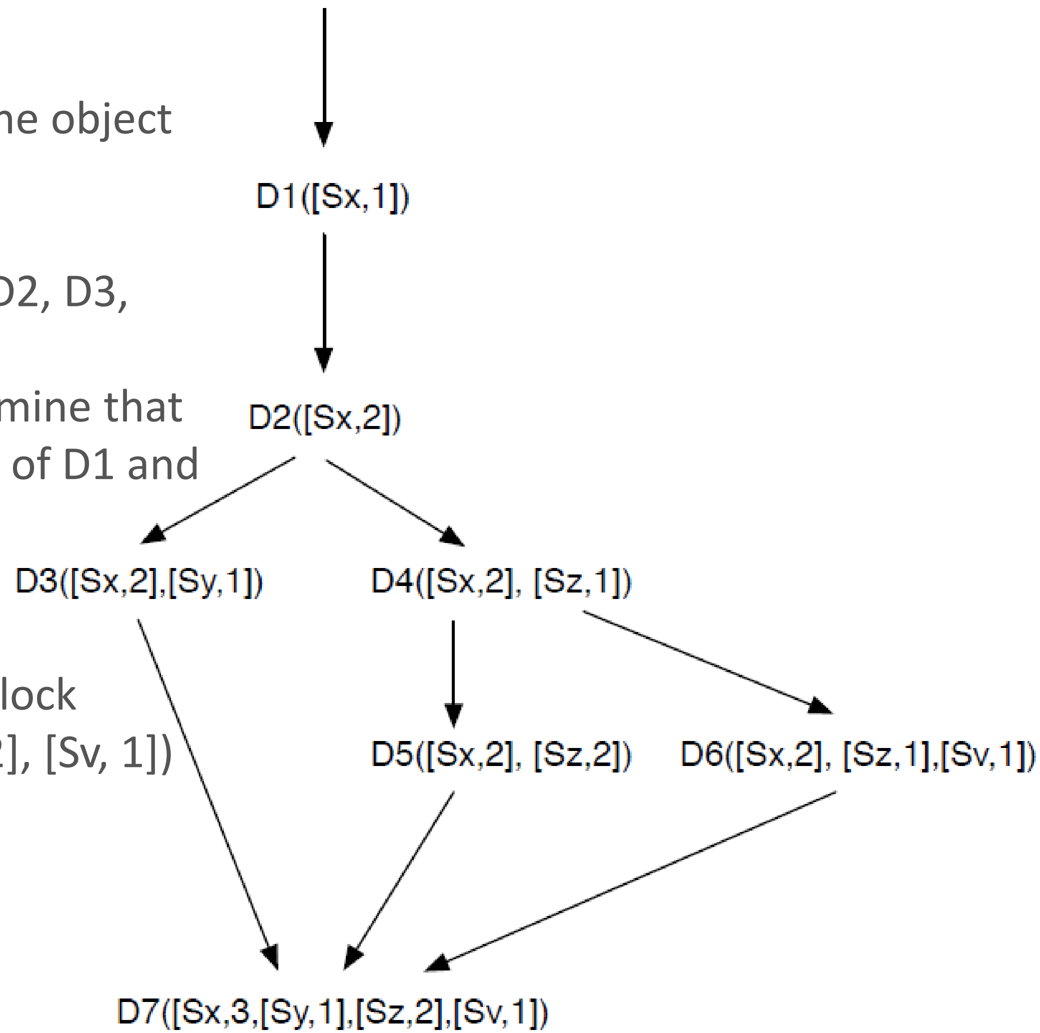
- **C2** updates D4 again (still unaware of D3) to D5, on the same node **Sz**
- Vector clock is set to ([Sx, 2], [Sz, 2])
- At the same time, a third client **C3** would read D4 because **Sv** handling the read is unaware of D3 and D5
- C3 would also update D4 to D6, and Sv would set the clock to ([Sx, 2], [Sz, 1], [Sv, 1])



> Vector Clocks in Dynamo (5)

Example

- After a while, **C1** is back and reads the object again
- Node **Sx** handles the request
- Dynamo might find all versions D1, D2, D3, D4, D5 and D6
- Syntactic reconciliation would determine that D3, D5 and D6 are causal successors of D1 and D2 and D4 (only for D5 and D6)
- So Sx returns D3, D5 and D6
- An update then would reconcile the different branches, and the vector clock would look like ([Sx, 3], [Sy, 1], [Sz, 2], [Sv, 1])



> Vector Clocks in Dynamo (6)

Size of the vector clocks

- Vector clocks may grow if many servers coordinate the writes to an object
- In practice, this is not likely because the writes are handled by one of the top N nodes in the preference list
- In case of network partitions or multiple server failures, write requests may be handled by nodes that are not in the top N nodes in the preference list causing the size of vector clock to grow

→ Limit the size of vector clock

- Truncation scheme: Along with each (node, counter) pair, Dynamo stores a timestamp that indicates the last time the node updated the data item
- When the number of (node, counter) pairs in the vector clock reaches a threshold, the oldest pair is removed from the clock
- Can lead to inefficiencies in reconciliation as the descendant relationships cannot be derived accurately
- "This problem has not surfaced in production and therefore this issue has not been thoroughly investigated…"

> Execution of get() and put() operations

Get and Put functions

- Reads and writes are handled by a coordinator
- Coordinator is the top alive node in the preference list

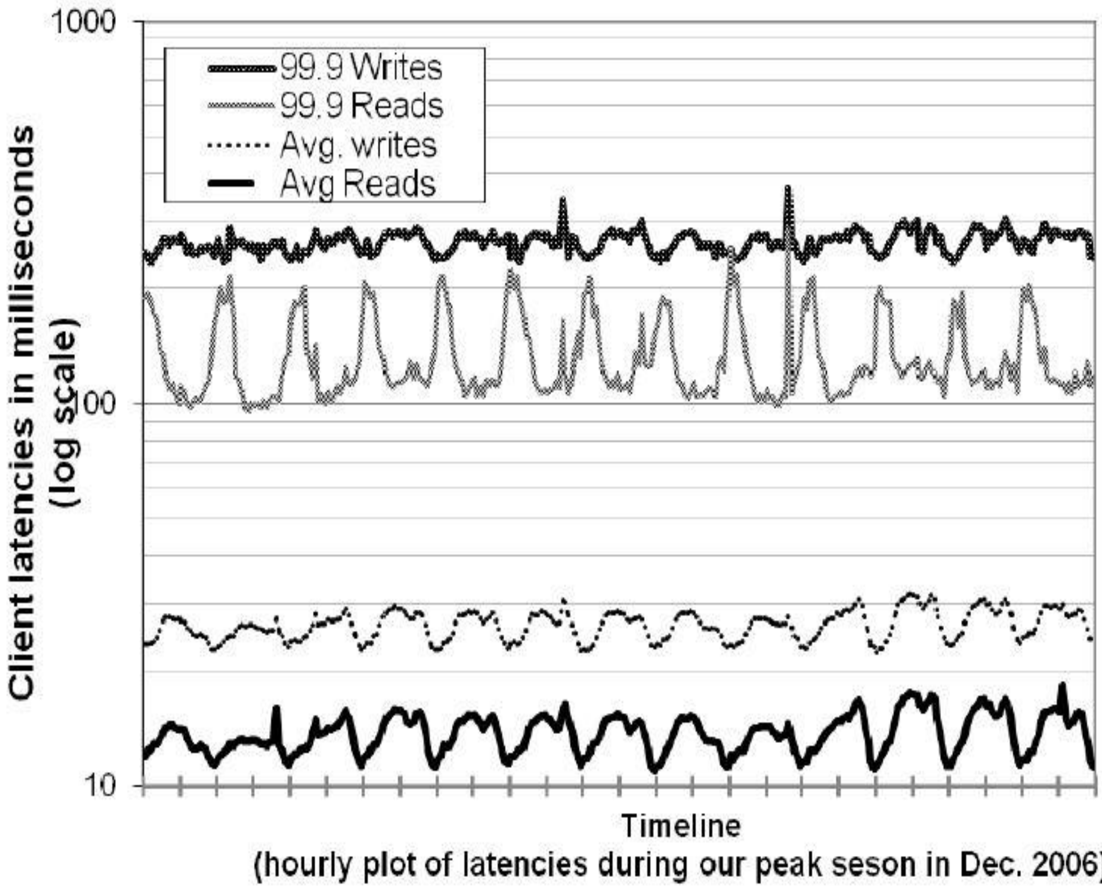
- Route the requests through generic **load balancer**
  - Selects a node based on the load information

or

- Use partition-aware **client library**
  - Knows the partitions and routes request directly to coordinator
  - Achieves lower latency
- Use of a consistency protocol similar to quorum systems to maintain the consistency of data

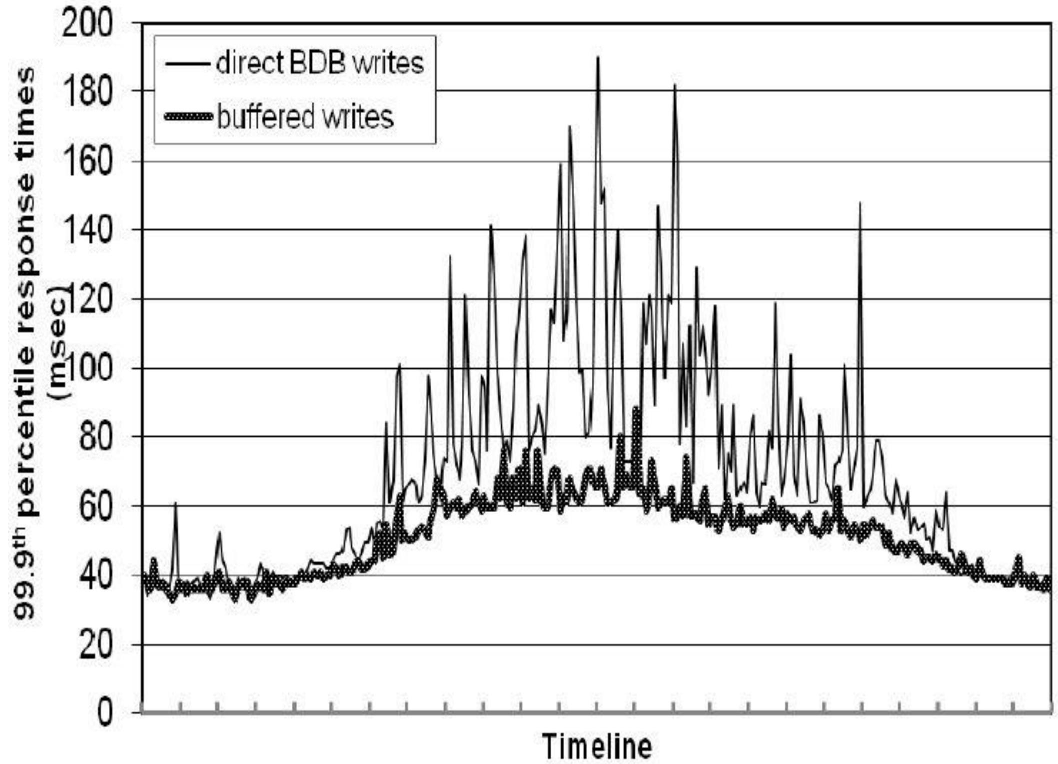
> Evaluation

- Average and 99.9 percentiles of latencies for read and write requests during peak request season of December 2006
- Intervals between consecutive ticks in the x-axis correspond to 12 hours
- Latencies follow a day-based pattern similar to the request rate
- 99.9 percentile latencies are an order of magnitude higher than averages



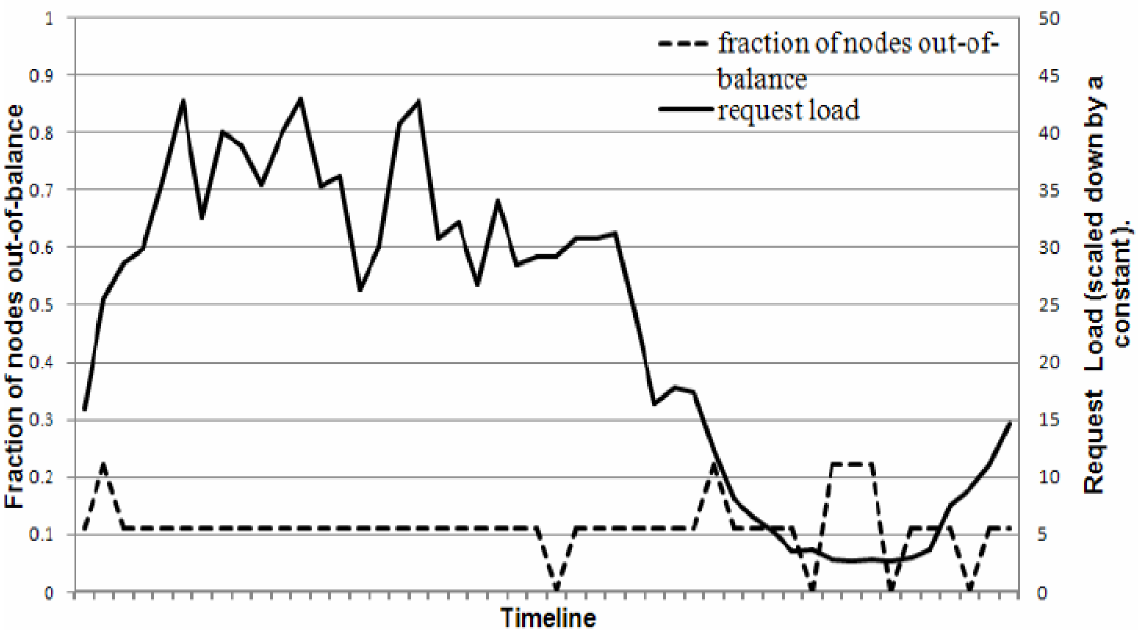
> Evaluation (2)

- Optimization: Trade-off durability guarantees for
- Object buffer in main memory
- Each write operation is stored in the buffer and gets periodically written to storage by a writer thread
- Comparison of performance of 99.9th percentile latencies for buffered vs. non-buffered writes over a period of 24 hours
- Intervals between consecutive ticks in the x-axis correspond to one hour



Ensuring Uniform Load distribution

- Requests received by each node was measured for a period of 24 hours - broken down into intervals of 30 minutes
- Node is considered to be “in-balance”, if the node’s request load deviates from the average load by a value a less than a certain threshold (here 15%)
- Otherwise the node was deemed “out-of-balance”
- Out-of-balance ratio decreases with increasing load
- During low loads the out-of-balance ratio is as high as 20% and during high loads it is close to 10%



> Conclusion

Basic idea

- Dynamo allows Amazon's customers to have a consistent experience even in face of server and network errors
- Gives a scalable solution with millions of data points to be queried quickly and efficiently
- Offloads complexity to the application to provide a simple, flexible, and fast server-side implementation