Key-Value Stores

Iztok Savnik, FAMNIT

November, 2022.

Literature

Paper:

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapa, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels, *Amazon's Highly-Available Key-Value Store*, SOSP, 2007.

Presentatiton:

Jeff Avery, CS 848: Modern Database Systems, 2015.

Outline

- Motivation
- Requirements
- Design
- Architecture
- Implementation

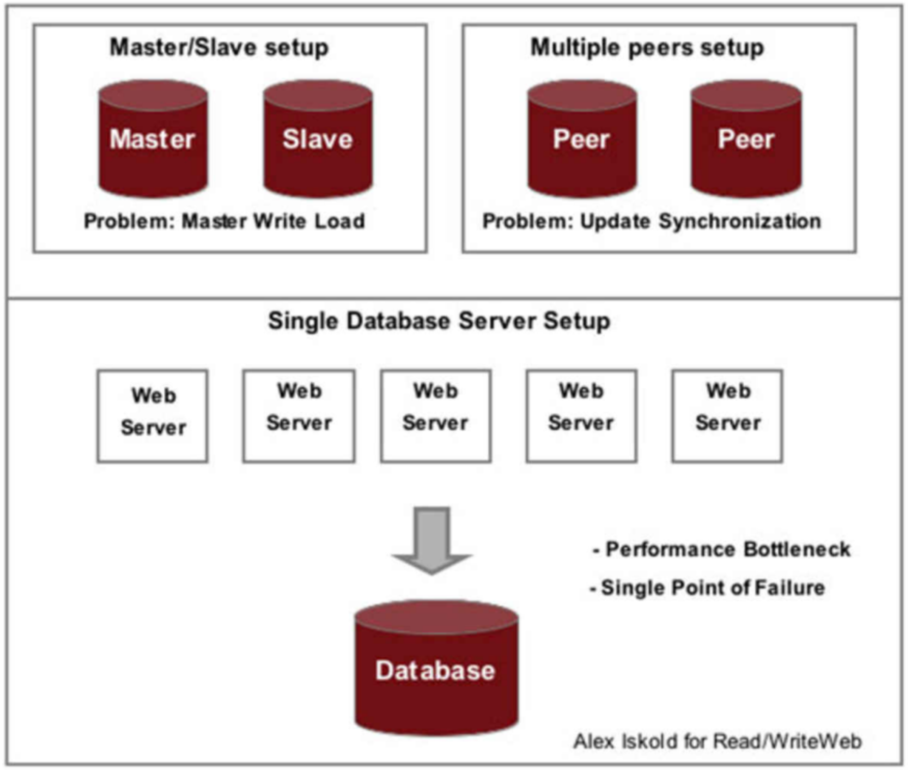
Motivation: KV stores

- “Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e--commerce operations in the world; even the slightest outage has significant financial consequences and impacts customer trust. “
- “Customers should be able to view and add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornados.“
- “There are always a small but significant number of server and network components that are failing at any given time. As such Amazon’s software systems need to be constructed in a manner that treats failure handling as the normal case without impacting availability or performance.”

- DeCandia (2007)

Motivation: RDBMS Replication

“It is difficult to create redundancy and parallelism with relational databases, so they become a single point of failure. In particular, replication is not trivial.”


Master/Slave setup
Master
Slave
Problem: Master Write Load
Multiple peers setup
Peer
Peer
Problem: Update Synchronization
Single Database Server Setup
Web Server
Web Server
Web Server
Web Server
Web Server
- Performance Bottleneck
- Single Point of Failure
Database
Alex Iskold for Read/WriteWeb

DynamoDB Approach

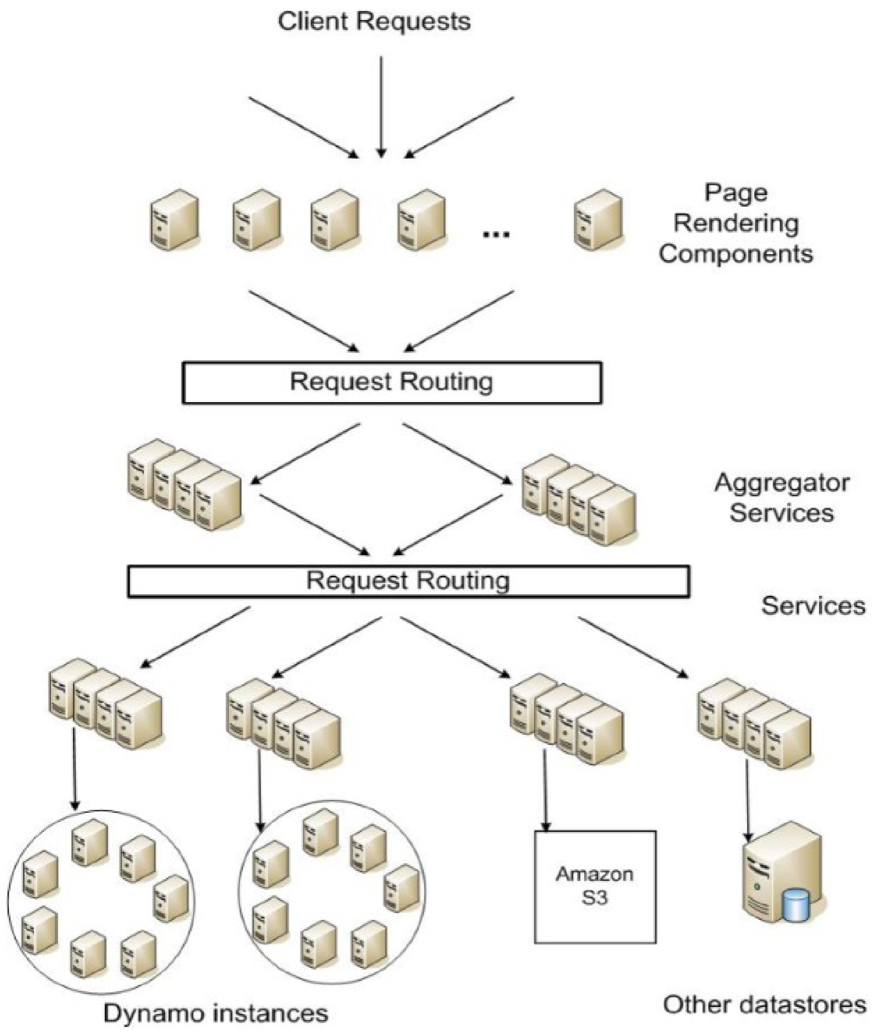
- Challenge
  - Designing a highly-available system that can scale to millions of users, while meeting service-level SLA.
- Problem
  - Traditional systems perform synchronous replica coordination in order to provide strongly consistent data, at the cost of availability.
  - Network and hardware failures mean that strong consistency and high data availability cannot be achieved simultaneously.
- Solution
  - Sacrifice strong consistency for high availability.
  - Give users the “ability to trade-off cost, consistency, durability and performance, while maintaining high-availability.”

Requirements: System

- ACID
  - ACID (Atomicity, Consistency, Isolation, Durability) is a set of properties that guarantee that database transactions are processed reliably.
  - Dynamo targets applications that operate with weaker consistency (the “C” in ACID) if this results in high availability.
- Query Model
  - Simple read and write operations to a data item that is uniquely identified by a key.
  - No operations span multiple data items and there is no need for relational schema.
- Efficiency
  - The system needs to function on a commodity hardware infrastructure.
  - Services must be able to configure Dynamo to consistently achieve latency and throughput requirements.
- Facing
  - Dynamo is used only by Amazon’s internal services.

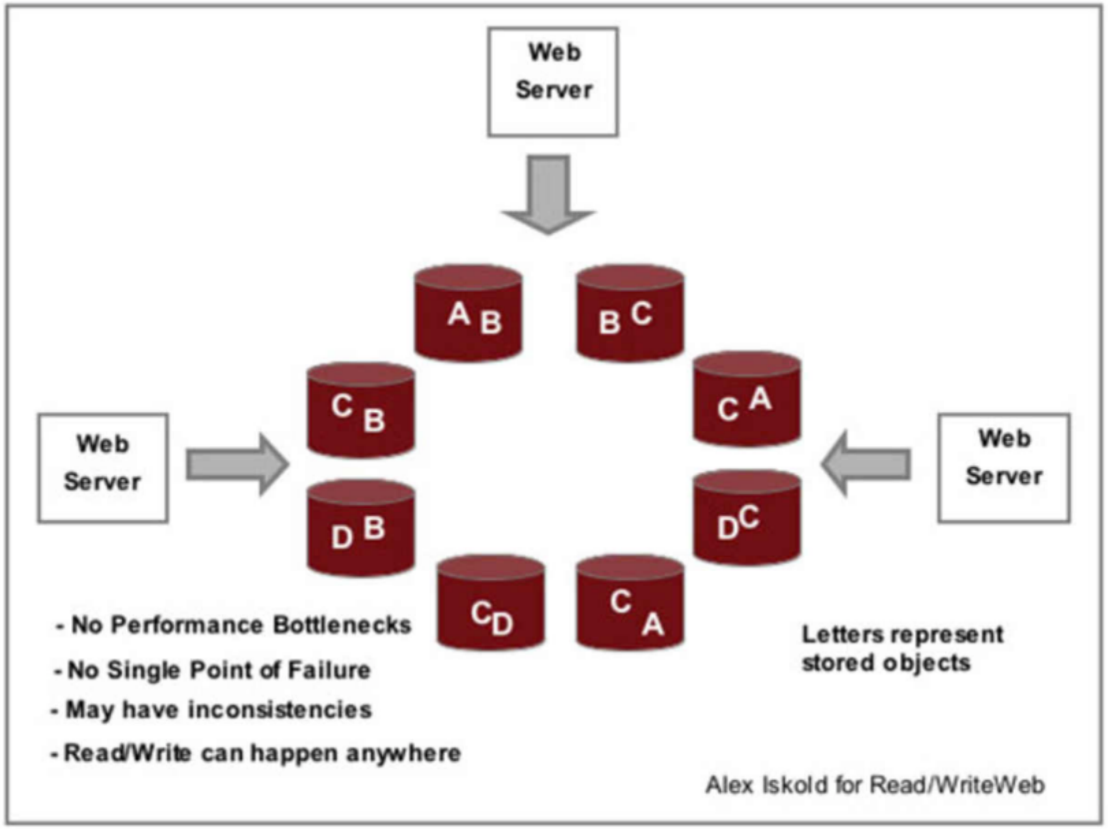
Requirements: SLA

- Service Level Agreements
- Contract where a client and a service agree on system-related characteristics.
  - Promises bounded time for a response.
- Every dependency in the platform needs to deliver with even higher bounds.
- At Amazon, SLAs are expressed and measured at the 99.9th percentile of the distribution (i.e. the edge-cases can represent critical customers).



Design: Distributed storage

- Remove the database as the bottleneck.
- Distributed storage nodes share the burden.
- Requests are routed to the storage node holding the data.


Web Server
Web Server
Web Server
A B
B C
C B
C A
D B
D C
C D
C A
- No Performance Bottlenecks
- No Single Point of Failure
- May have inconsistencies
- Read/Write can happen anywhere
Letters represent stored objects
Alex Iskold for Read/WriteWeb

Design: Optimistic Replication

- Optimistic replication allows changes to propagate asynchronously. Availability is increased, but the risk is that you have multiple, conflicting versions of data in the system.
- Conflicts aren’t prevented, but resolved.
  - Notion of an “eventually consistent data store” and delaying reconciliation.
  - When to resolve: resolving conflicts during reads, not writes (e.g. shopping cart example, cannot reject writes).
  - Who to resolve: tradeoff between system and application level resolution.

Design: Key Principles

- Incremental scalability: Dynamo should be able to scale out one storage host (henceforth, referred to as “node”) at a time
- Symmetry: Every node in Dynamo should have the same set of responsibilities as its peers; there should be no distinguished node or nodes that take special roles or extra set of responsibilities.
- Decentralization: An extension of symmetry, the design should favor decentralized peer-to-peer techniques
- Heterogeneity: The system needs to be able to exploit heterogeneity (i.e. work allocated is proportional to the characteristics of the hardware).

Architecture: Dynamo Techniques

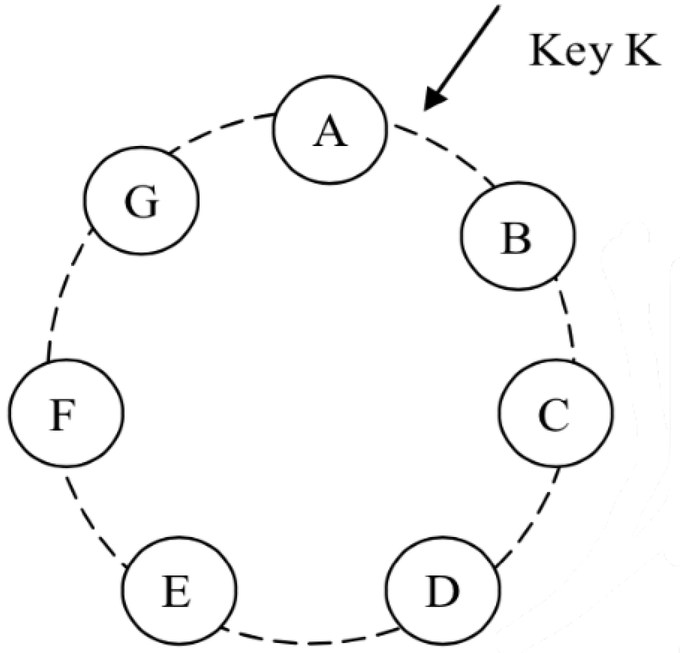
Problem	Technique
Partitioning and replication	Consistent hashing (notions of “eventually consistent” and delayed reconciliation).
Consistency	Object versioning. Quorum--like techniques used to maintain consistency.
Recovering from failures	Merkle trees used to quickly detect inconsistencies.
Membership	Gossip--based membership protocol, also used for failure detection.
Decentralized	Minimal need for manual administration (i.e. no manual partitioning required)

Architecture: System Interface

- Focus on simple query model.
- Key-value storage of objects:
  - get() : for a given key, returns a single object or list of objects with their context (metadata including version).
  - put() : writes replicas (versions) to disk.

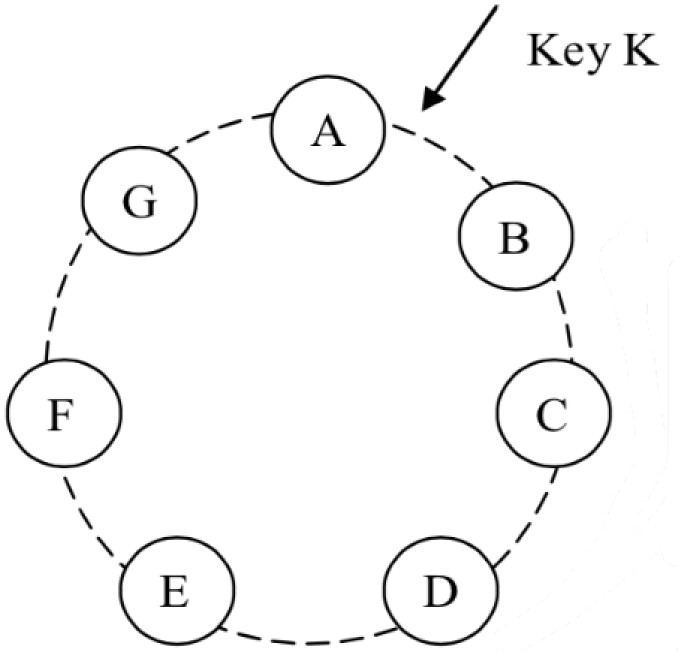
Architecture: Partitioning

- Scale incrementally by dynamic partitioning across all available nodes.
- Consistent hashing: the output range of the hash function returns is a bounded, circular region.
  - Newly added nodes are randomly assigned a key/positon.
  - Nodes are responsible for the values ranging from the previous node to themselves.



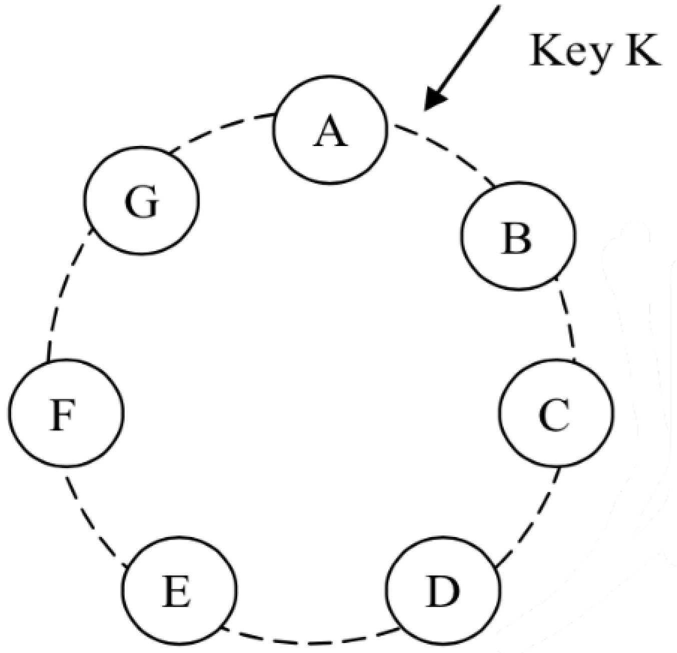
Partitioning: Virtual Nodes

- The Problem with Consistent Hashing:
  - Random positioning leads to non-uniform load distribution
  - Oblivious to the heterogeneity in the performance of nodes
- Solution:
  - Each node gets assigned to multiple positions in the ring (positions called **tokens**)
  - These nodes are also called "**virtual nodes**"



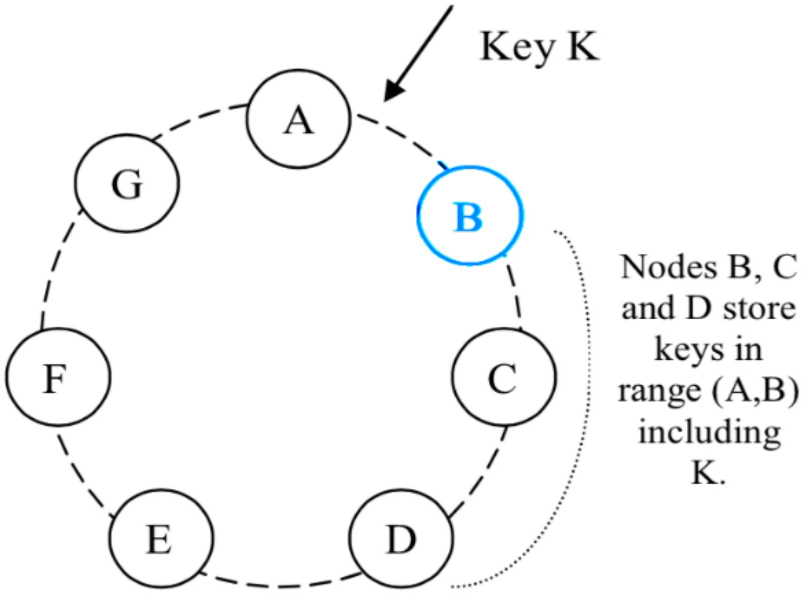
Partitioning: Virtual Nodes

- Solution: (cont.)
  - This allows for failover when a node is (un)available, or load rebalancing in extreme cases.
    - node unavailable: the load is evenly dispersed across the remaining nodes.
    - node available again (or new): it accepts a roughly equivalent amount of load from each of the other available nodes
    - # of virtual nodes assigned based on capacity and physical node infrastructure
  - Virtual nodes are easier to reallocate!



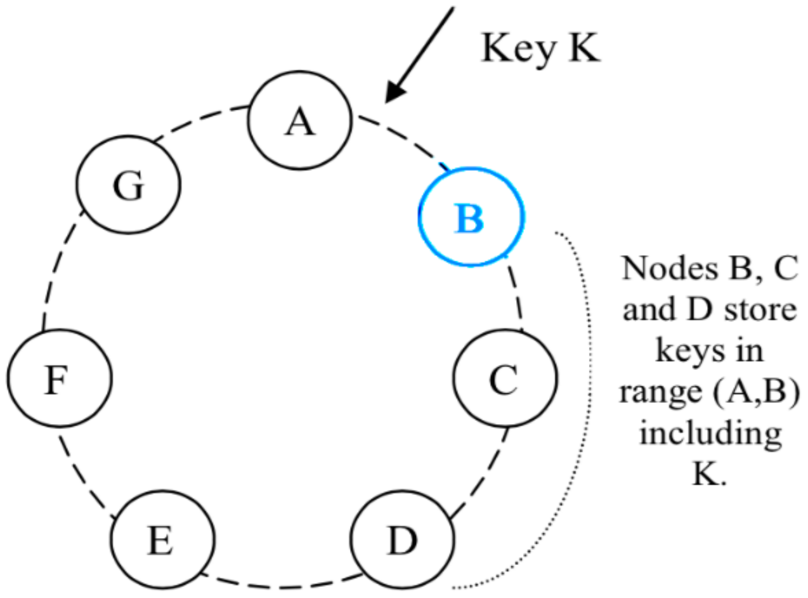
Partitioning: Replication

- To achieve high--reliability, Dynamo replicates data across nodes.
  - Each data item is replicated at N hosts, where N is a parameter configured “per-instance”!
  - Each key is assigned a “**coordinator**” node that is responsible for handling replication.
  - The coordinator, in turn, handles replication for all items that fall within its range.
  - The coordinator replicates these keys at the N-1 clockwise successor nodes in the ring.



Partitioning: Replication

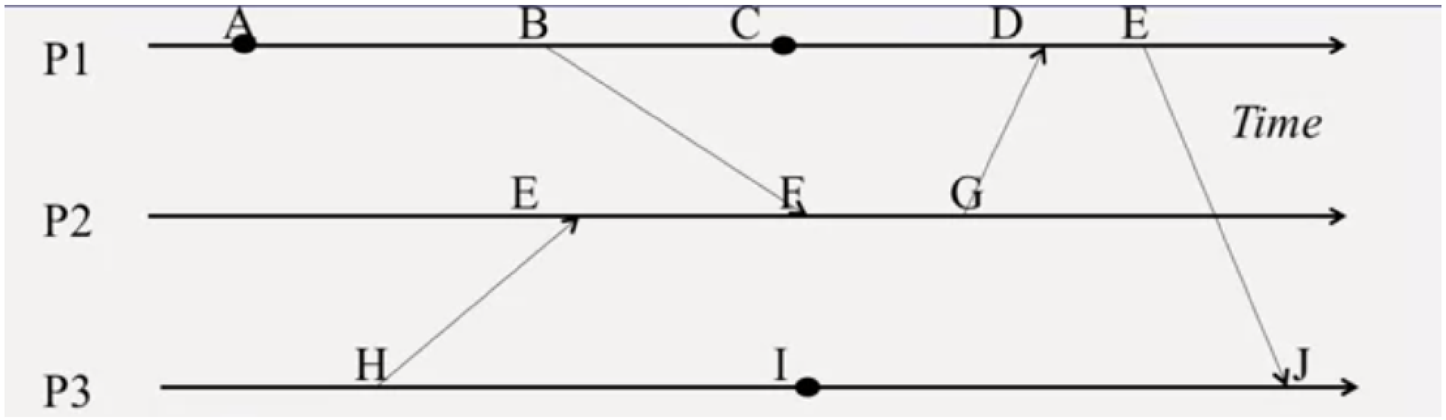
- This list of key-owners is called the **preference list** (and is circulated around the system).
- Every node can determine which nodes should be in this list for any particular key
- Since virtual nodes used: N succ. positions may be owned by less than N distinct physical nodes
- The preference list for a key skipps positions so that list contains only distinct physical nodes

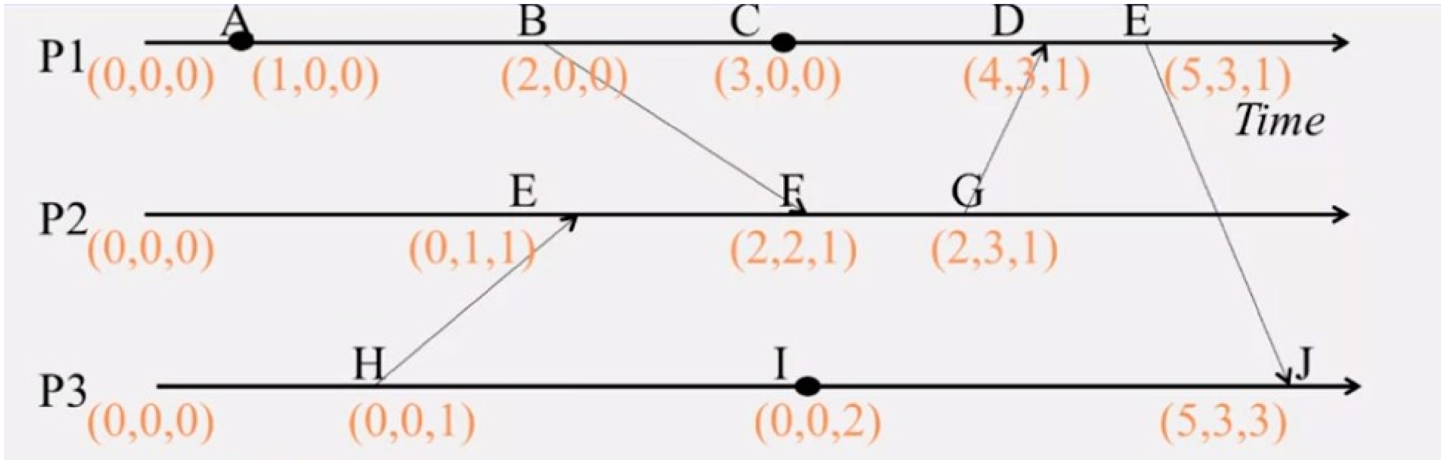


Architecture: Consistency

- Dynamo guarantees “**eventual consistency**”. Updates are propagated asynchronously, so there’s no guarantee that replicas are always in-sync.
- Multiple versions of a particular key-value pair may exist at any given time (e.g. one node returns a value before a replica has propagated an update).
- To handle this use **data versioning**:
  - Each modification results in a new version being created.
  - Most of the time, new versions replace old versions.
  - When versions cannot be reconciled automatically, <u>vector clocks</u> are used to order the versions and atempt to reconcile version histories.

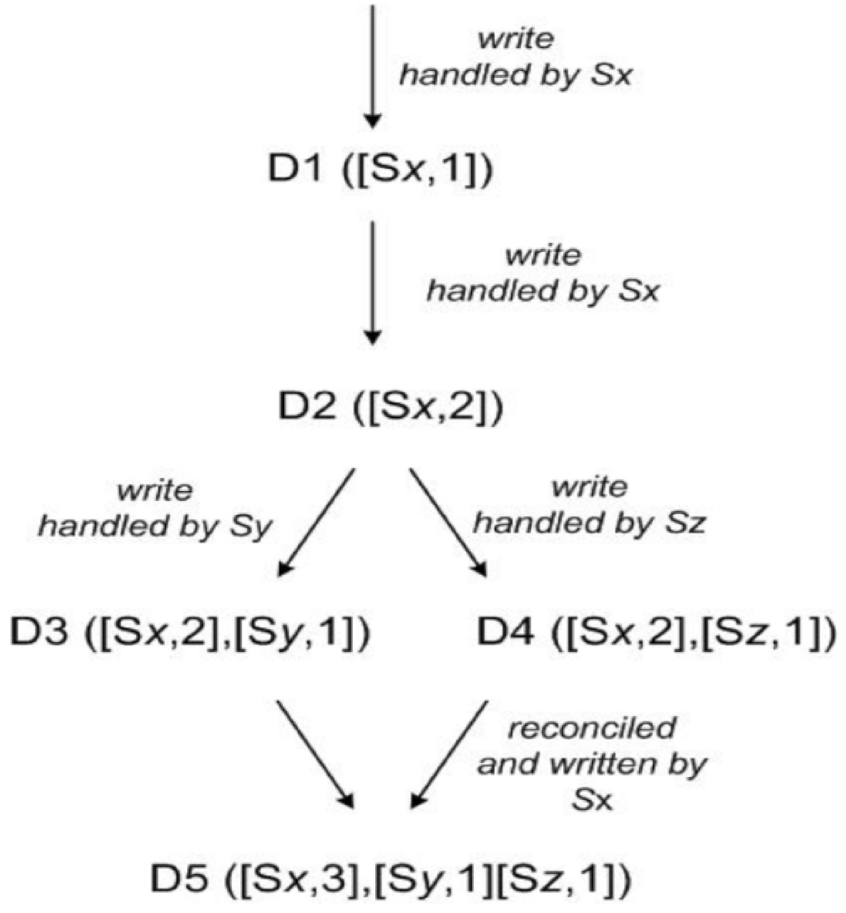
Consistency: Vector Clocks





Consistency: Vector Clocks

- A vector clock is effectively a list of (node, counter) pairs.
- One vector clock is associated with every version of every object.
- Having two versions of object we can determine:
  - Parallel branches or causal ordering
  - Counters of Vc1 are ≤ than the correlated counters of Vc2 then Vc1 is an ancestor of Vc2
  - Otherwise, parallel updates: object versions in conflict and require reconciliation.



Consistency: Vector Clocks

- Upon **update** request:
  - Must specify a version (from the context that it obtained during the initial get() operation).
  - Any updates to data will result in the node creating a new version (with a new vector clock timestamp).
- Upon **read** request:
  - May **reconcile** and collapse versions
  - If multiple branches that cannot be (syntactically) reconciled, it will return all objects at leaves with version info in the contex

Architecture: get() and put()

- First base case.
- Normal operations look like this:
  - get() and put() operations from the application need to be sent to the appropriate node.
  - Two strategies to locate this node:
    - Allow a load-balancer to select the appropriate node
    - Use a partition-aware library that knows which nodes are responsible for a given key. This pushes responsibility for node selection onto the application.
  - A node handling a read or write operation is known as the **coordinator**. Typically, this is the first among the top N nodes in the preference list.

Handling Failures: Data Consistency

- What happens when storage nodes disagree?
  - Dynamo uses a quorum-based consistency protocol. This means that a number of storage nodes must “agree” on result.
- For a series of N nodes, there are two configurable values for a given request:
  - R : the minimum number of nodes that must participate in a **read** request
  - W : the minimum number of nodes that must participate in a **write** request
- Use:
  - $R + W > N$: Quorum
  - $R + W \leq N$: Not-Quorum (but better latency!)

Handling Failures: Quorum Use

- Examples
  - put(): The coordinator node will
    - Create a new version.
    - Send the data to N healthy nodes.
    - If W-1 respond, treat the write as successful.
  - get(): The coordinator node will
    - Request versions of that data from N highest-ranked reachable nodes in the preference list.
    - Waits for R responses before returning the result to the client. If it receives multiple versions, it will reconcile and write-back a new version (superseding the previous ones).

Handling Failures: Hinted Handoff

Dynamo uses a “sloppy quorum”, where the first N available, healthy nodes are used to determine quorum. This increases durability for writes.

- **Hinted handoff** occurs when a node is unavailable, and the next replica handles the request. This node tracks the node that was unavailable, and when it comes back online, delivers the replica.
- Value of W determines durability:
  - W=1: allows any writes as long as a single node is available to process it.
  - W>1: in practice, we usually aim slightly higher than 1.

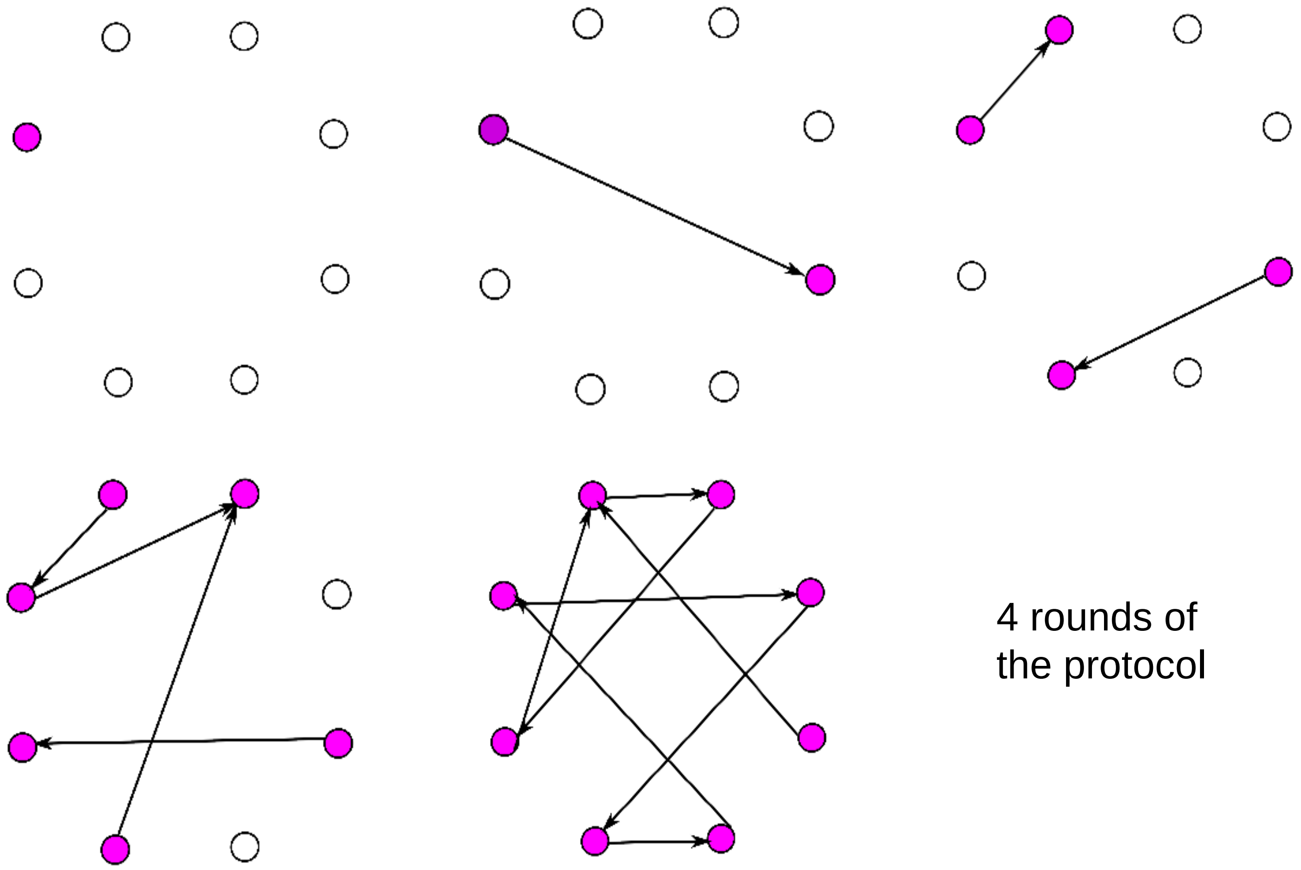
Handling Failures: Replica Sync

- To detect inconsistencies between replicas faster, and decrease timme required, Dynamo uses **Merkle trees**.
  - A Merkle tree is a hash tree where leaves are hashes of the individual leaves. Parent nodes higher in the tree are hashes of their children.
  - Each branch can be checked for membership without traversing the entire tree. Less data needs to be passed around when checking status.
- Each node maintains a Merkle tree for each key range it hosts. To compare values with another node, they just exchange root nodes for that tree.

Ring Membership & Failure Detection

- Node outage (or manual error) rarely signifies a permanent departure
  - Should not result in rebalancing of the partition assignment or repair of the unreachable replicas
  - Therefore, use explicit mechanism to initiate add/remove of nodes from ring (to avoid trashing)
- **Gossip-based protocol** propagates changes automatically:
  - Maintains an eventually consistent view of membership
  - Each node contacts another random node and they exchange membership information.

Ring Membership: Gosip Protocol



4 rounds of
the protocol

Ring Membership & Failure Detection

- Adding new nodes
  - New nodes are assigned a set of tokens (virt. nodes) and gossip
  - Map nodes to their respective token sets and make mapping persistent
  - Partitioning and placement information also propagates via the gossip-based protocol. Therefore, each node is aware of token ranges handled by its peers
  - Seed nodes prevent new nodes from becoming isolated (logical partitions)
- Node-mappings are also propagated though gossip!
- Unresponsive nodes are also flagged and gossiped-about.

Implementation: Storage Node

Built in Java.

Each storage node has three main components:

- Request coordination
  - Multi-stage messaging pipeline, built using Java NIO channels.
  - Each client requests results in the creation of a single state machine to handle that request.
- Membership and failure detection
- Local persistence engine
  - Pluggable, supports many different engines (incl. Berkeley Database (BDB) Transactional Data Store, MySQL, in-memory).
  - Most use BDB Transactional Data Store.

Implementation: Usage Patterns

Dynamo is used by a number of services with drastically different usage patterns:

- **Business logic specific reconciliation**: Many-node replication, with client handling reconciliation. e.g. shopping cart logic.
- **Timestamp based reconciliation**: “Last write wins”. e.g. customer session service.
- **High-performance read engine**: Services with a high read-request rate, small number of updates. e.g. product catalogs.

Value in allowing applications to tune R and W (affecting consistency, durability, availability)

- – Common: (N:3, R:2, W:2)
- – High-performance read: (N:3, R:1, W:3)
- – High-performance write: (N:3, R:3, W:1) – dangerous

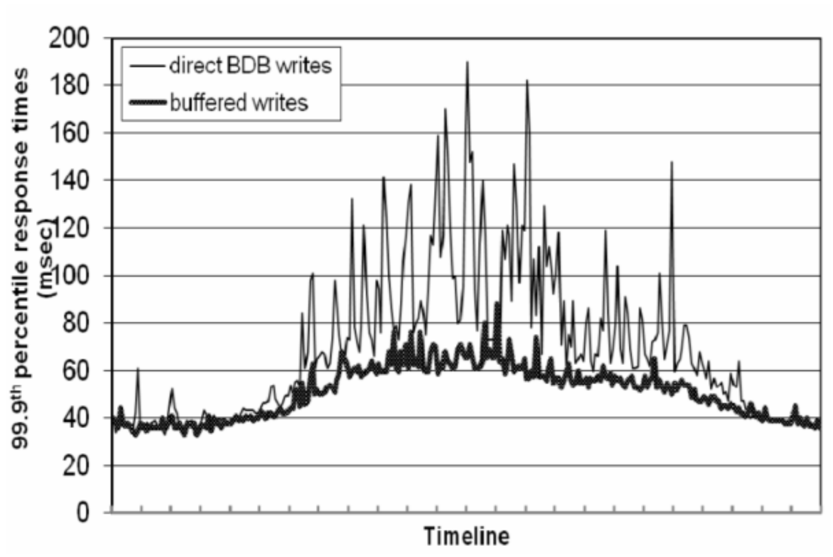
Implementation: Performance Tradeoffs

“A typical SLA required of services that use Dynamo is that 99.9% of the read and write requests execute within 300ms.“

- Commodity hardware makes I/O challenging.
- Multiple storage nodes constrains performance to the slowest node.

To achieve higher-performance on writes, an optional writer thread can be used to buffer writes.

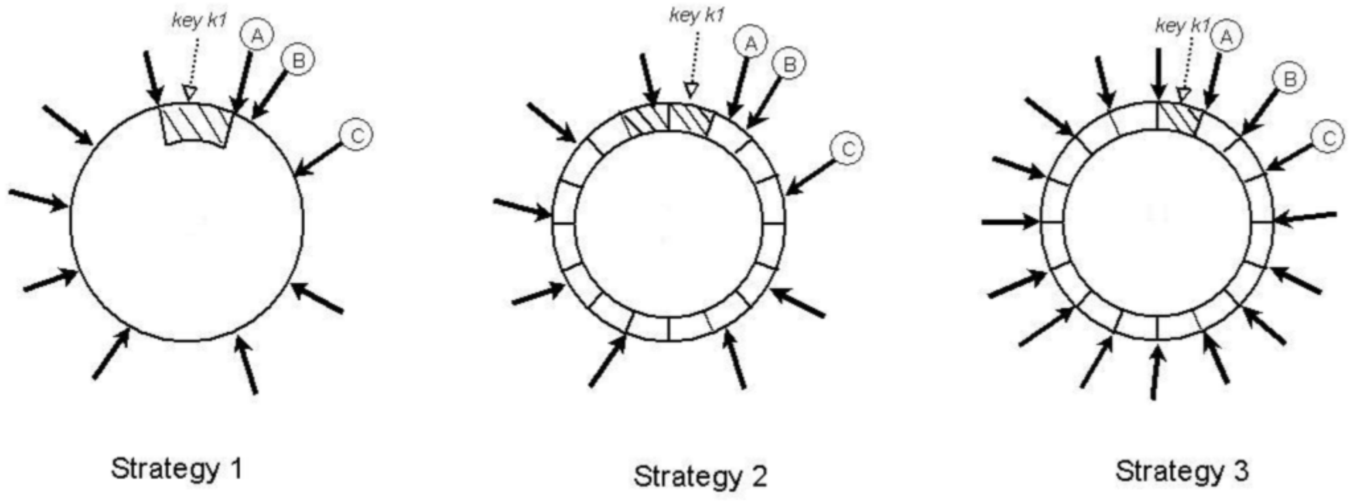
- Improves latencies at the risk of losing data (i.e. server crash).
- Can mitigate by having a node assigned to “durable writes”.



Implementation: Load Distribution

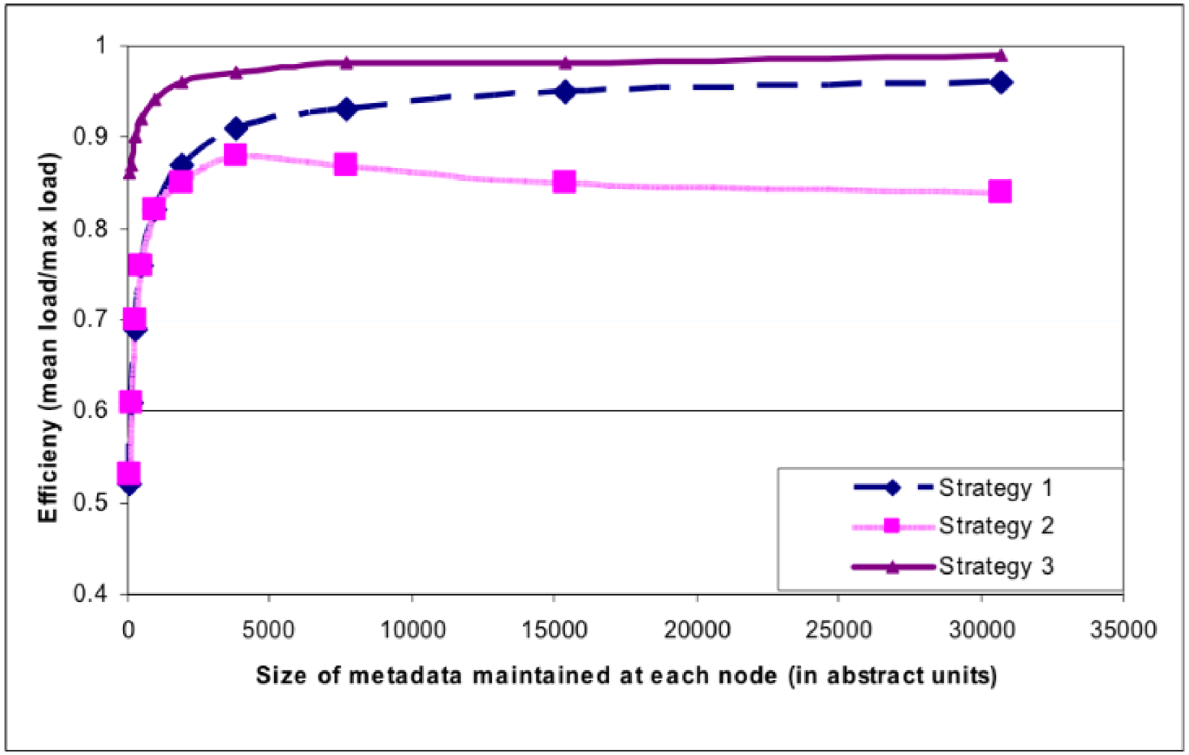
- Dynamo uses consistent hashing to partition it’s key space across all of the available nodes.
  - This results in a semi-uniform distribution.
  - The assumption is that there are enough nodes at either end of the distribution to handle any skewed access patterns (i.e. “popular” requests at peak times).
- Is there a better way of partitioning keys?

Load Distribution Strategies



1) T random tokens per node and partition by token value
   - New nodes need to “steal” key ranges from exiting nodes. Changing key ranges invalidates Merkle trees. Difficult to archive.
2) T random tokens per node and equal-sized partitions
   - Decoupling of partitioning and partition placement. Enables the possibility of changing the placement scheme at runDme.
3) Q/S tokens per node, equal-sized partitions
   - When a node leaves the system, its tokens are randomly distributed to the remaining nodes. When a node joins the system it "steals" tokens from other nodes.

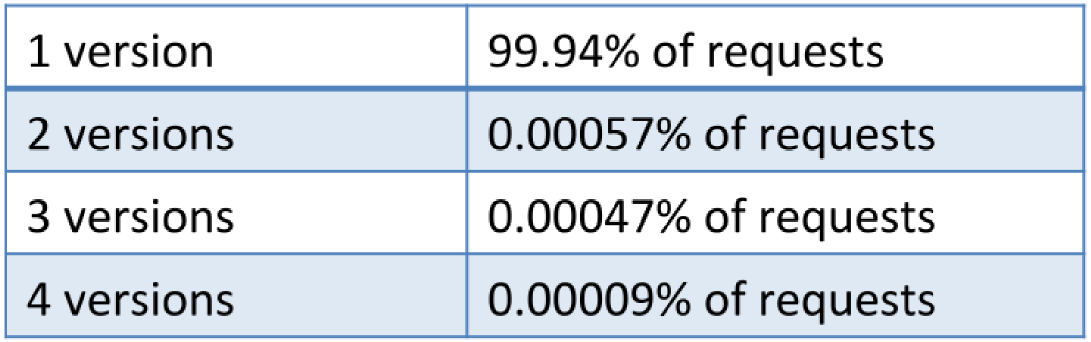
Load Balancing Efficiency



- Strategy 2 is the worst, Strategy 3 is the best.
- Compared to Strategy 1, Strategy 3 achieves better efficiency and reduces the size of membership information maintained at each node by three orders of magnitude.
- Strategy 3 is faster to bootstrap (fixed partitions) and archive.

Divergent Versions

- Divergent versions of a data item occur when
  - Failures are happening, such as node or data center failures.
  - A large number of writes against the same data are happening, and multiple nodes are handling the updates.
- The number of versions returned to the shopping cart service was profiled for a period of 24 hours.

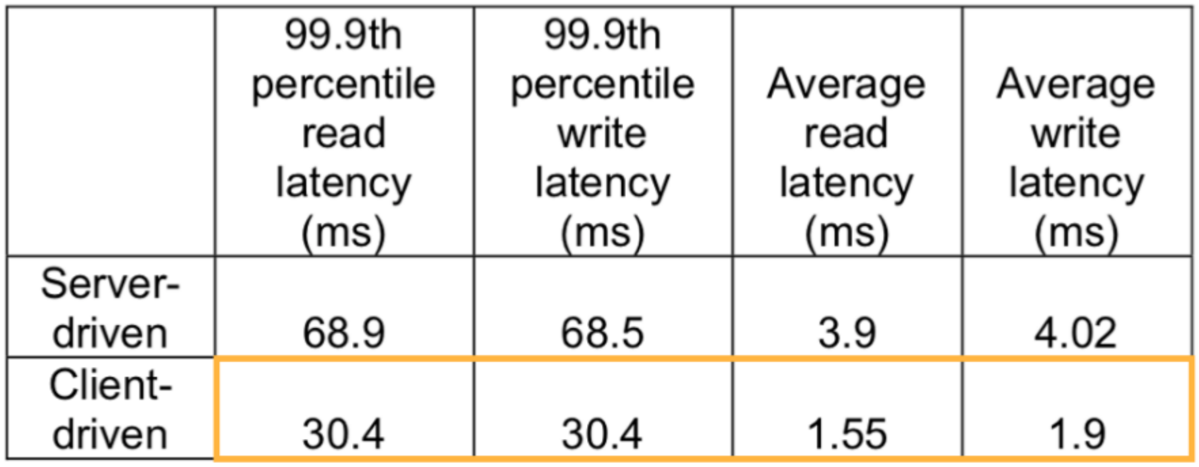


1 version	99.94% of requests
2 versions	0.00057% of requests
3 versions	0.00047% of requests
4 versions	0.00009% of requests

Coordination

There are two ways of locating a node to service a request:

- A load-balancer can determine the node for a given key/request. The burden is on the load-balancer/system.
- The client can periodically sample a random node (every 10 seconds), grab its membership state and use that to query nodes directly.



	99.9th percentile read latency (ms)	99.9th percentile write latency (ms)	Average read latency (ms)	Average write latency (ms)
Server-driven	68.9	68.5	3.9	4.02
Client-driven	30.4	30.4	1.55	1.9