Dynamo

Amazon's Highly Available Key-value Store

Suma Dodmani, Austin Longo, Swathi Upadhya

Dynamo

- Highly Available, and Scalable Distributed NoSQL Database service built on Amazon’s platform
- Key - Value Storage Model
- Designed for Massive Scalability
- Decentralized System with minimum manual Intervention
- Dynamo used by Applications to provide “Always - On” experience
- Great fit for Mobile, Gaming, IoT, Web-Scale Applications, Social Networks

Dynamo Applications

SAMSUNG

COMCAST

lyft

tinder

amazon

Expedia®

NETFLIX

Adobe®

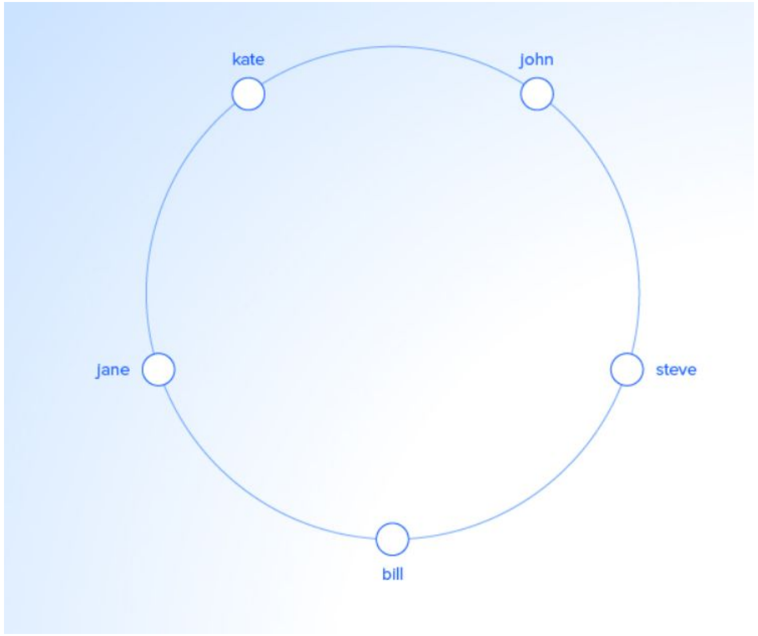
CapitalOne®

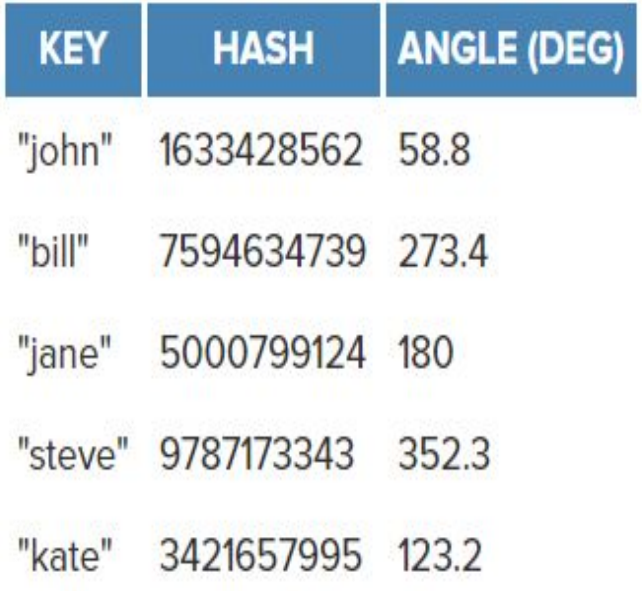
airbnb

Key Design Techniques of Dynamo

- Consistent Hashing for data replication
- Object Versioning to achieve consistency
- Quorum to maintain data consistency through reads and writes
- Gossiping for failure detection
- Decentralized System with very little manual intervention

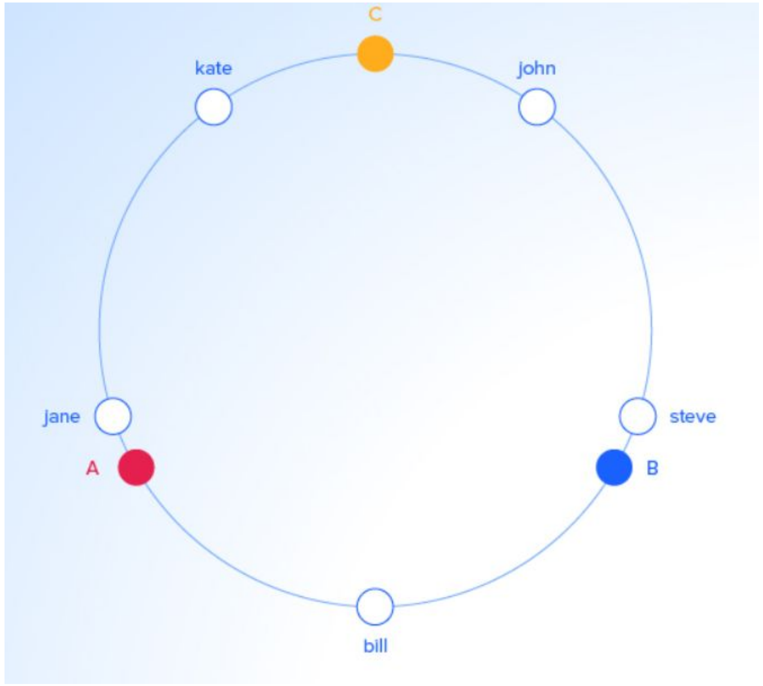
Consistent Hashing





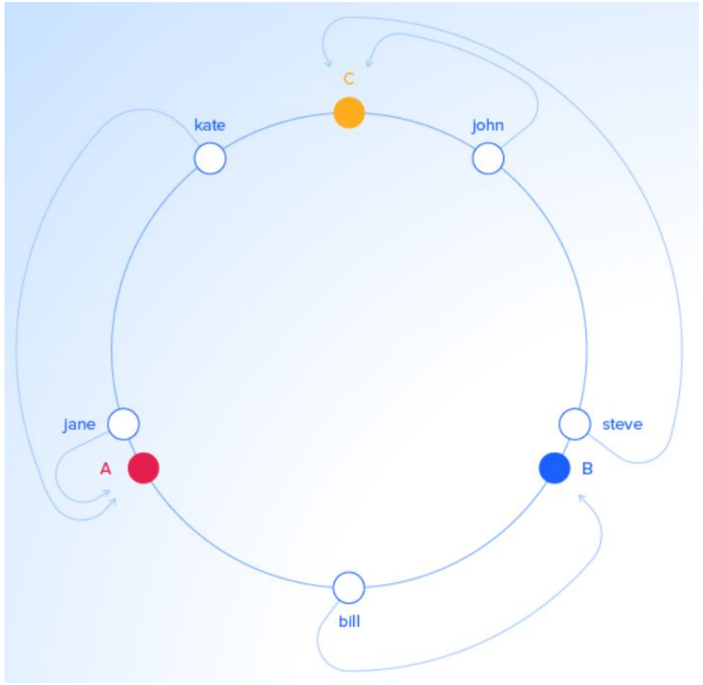
KEY	HASH	ANGLE (DEG)
"john"	1633428562	58.8
"bill"	7594634739	273.4
"jane"	5000799124	180
"steve"	9787173343	352.3
"kate"	3421657995	123.2

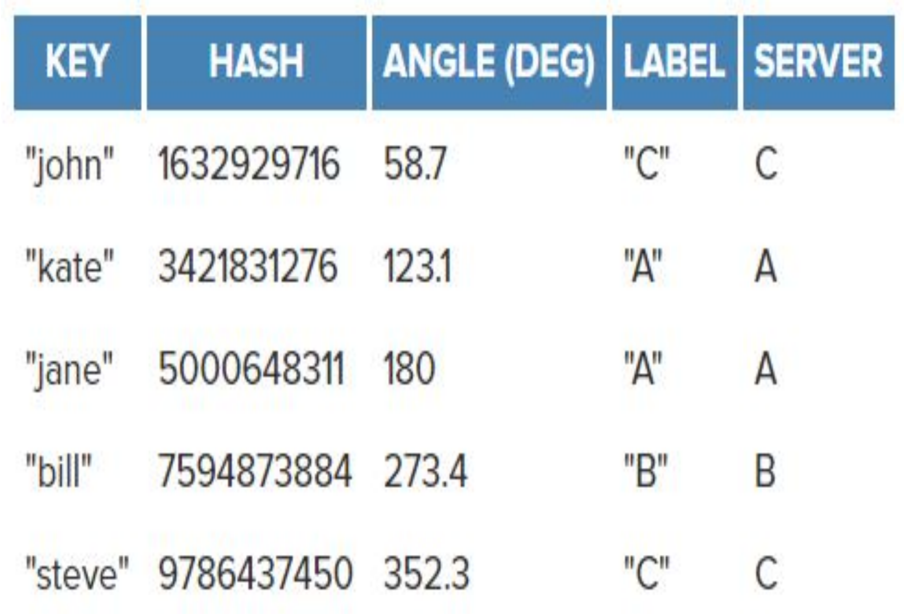
Consistent Hashing



KEY	HASH	ANGLE (DEG)
"john"	1633428562	58.8
"bill"	7594634739	273.4
"jane"	5000799124	180
"steve"	9787173343	352.3
"kate"	3421657995	123.2
"A"	5572014558	200.6
"B"	8077113362	290.8
"C"	2269549488	81.7

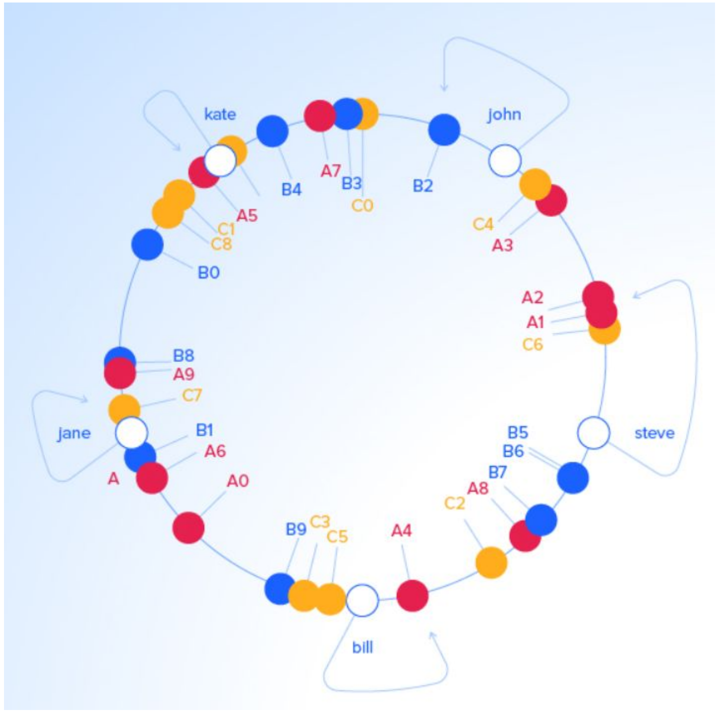
Consistent Hashing

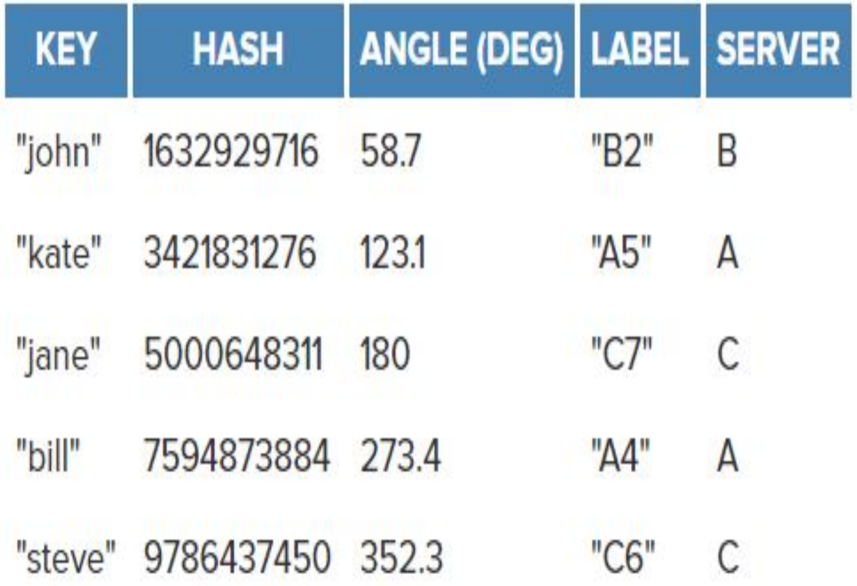




KEY	HASH	ANGLE (DEG)	LABEL	SERVER
"john"	1632929716	58.7	"C"	C
"kate"	3421831276	123.1	"A"	A
"jane"	5000648311	180	"A"	A
"bill"	7594873884	273.4	"B"	B
"steve"	9786437450	352.3	"C"	C

Consistent Hashing

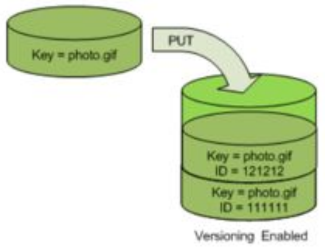




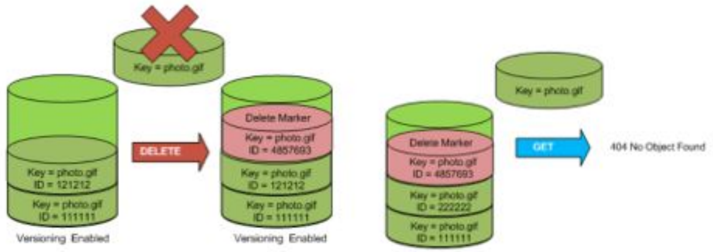
KEY	HASH	ANGLE (DEG)	LABEL	SERVER
"john"	1632929716	58.7	"B2"	B
"kate"	3421831276	123.1	"A5"	A
"jane"	5000648311	180	"C7"	C
"bill"	7594873884	273.4	"A4"	A
"steve"	9786437450	352.3	"C6"	C

Object Versioning

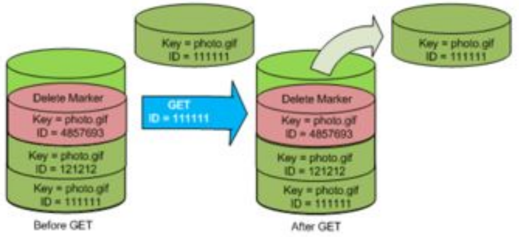
Object Versioning in Amazon Dynamo



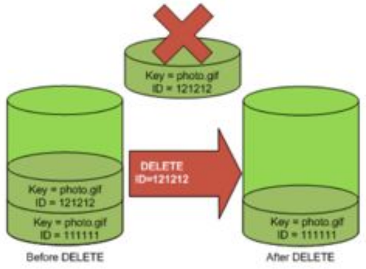
Deleting an Object


Key = photo.gif
Key = photo.gif ID = 121212
Key = photo.gif ID = 111111
Versioning Enabled
DELETE
Delete Marker
Key = photo.gif ID = 4857693
Key = photo.gif ID = 121212
Key = photo.gif ID = 111111
Versioning Enabled
Key = photo.gif
Delete Marker
Key = photo.gif ID = 4857693
Key = photo.gif ID = 222222
Key = photo.gif ID = 111111
GET
404 No Object Found

Getting an older/previous version of object


Delete Marker
Key = photo.gif ID = 4857693
Key = photo.gif ID = 121212
Key = photo.gif ID = 111111
Before GET
Key = photo.gif ID = 111111
GET ID = 111111
Delete Marker
Key = photo.gif ID = 4857693
Key = photo.gif ID = 121212
Key = photo.gif ID = 111111
After GET
Key = photo.gif ID = 111111

Deleting an object using version ID


Key = photo.gif ID = 121212
Key = photo.gif ID = 121212
Key = photo.gif ID = 111111
Before DELETE
DELETE ID=121212
Key = photo.gif ID = 111111
After DELETE

System Assumptions and Requirements

Query Model - Simple Read/ Write Operations to data item that is uniquely identified by key

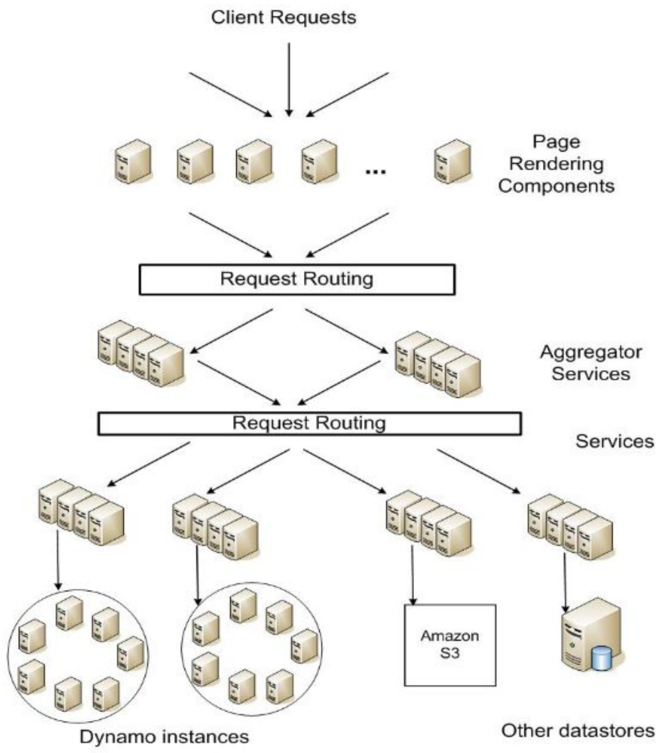
ACID - Prefers Availability over Consistency, No Isolation guarantees

Efficiency - Stringent Latency Requirements to meet SLAs

Service Level Agreements (SLAs)

Application delivers the functionality in bounded time. So, each and every dependency in the platform will have to deliver it's functionality with even tighter bounds.

Evaluating Performance Oriented SLA - 99.9th percentile of Distribution



Design Considerations

- Prefers Availability over Consistency
- Eventually Consistent with data updation to replicas in background
- Always writable data store, update conflicts resolved during reads - “Always Writeable”
- System or Application perform conflict resolution
- Incremental Scalability
- Symmetry
- Decentralization
- Heterogeneity

When is Dynamo used?

- Always writable data store
- Built for infrastructure within Single Administrative Domain - Trusted nodes
- No support for Hierarchical Namespaces - Simple Key/Value Store
- Built for latency sensitive applications
- Zero - Hop Distributed Hash Table
- Does not focus on data integrity and security

System Architecture

System Interface

`get(key)`

- *return*: Object associated with key & context

`put(key, context, object)`

- Places the object on replicas determined from the MD5 hash of the key

Techniques

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

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.

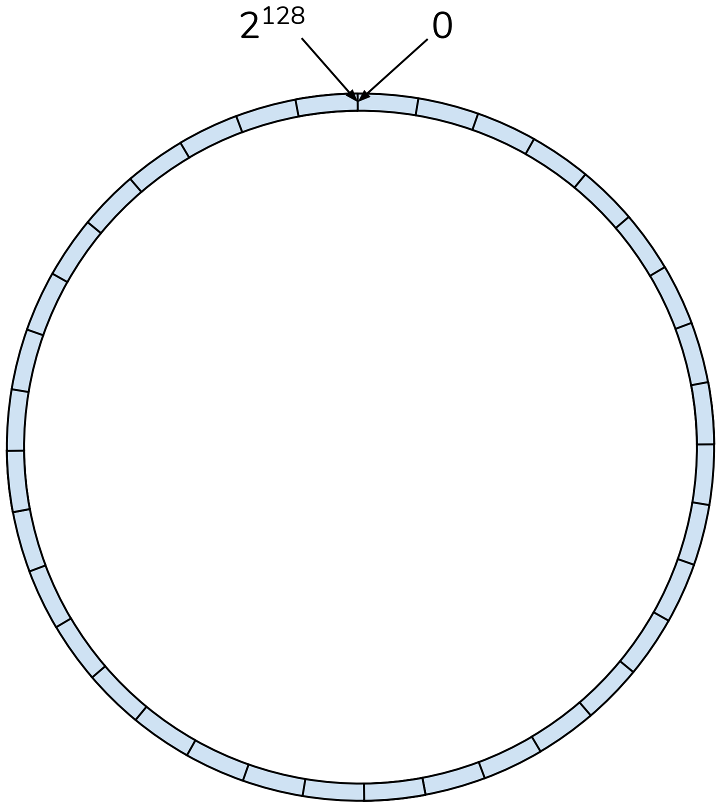
Partitioning Algorithm

Provides a scheme for **distributing load** across all participating storage hosts.

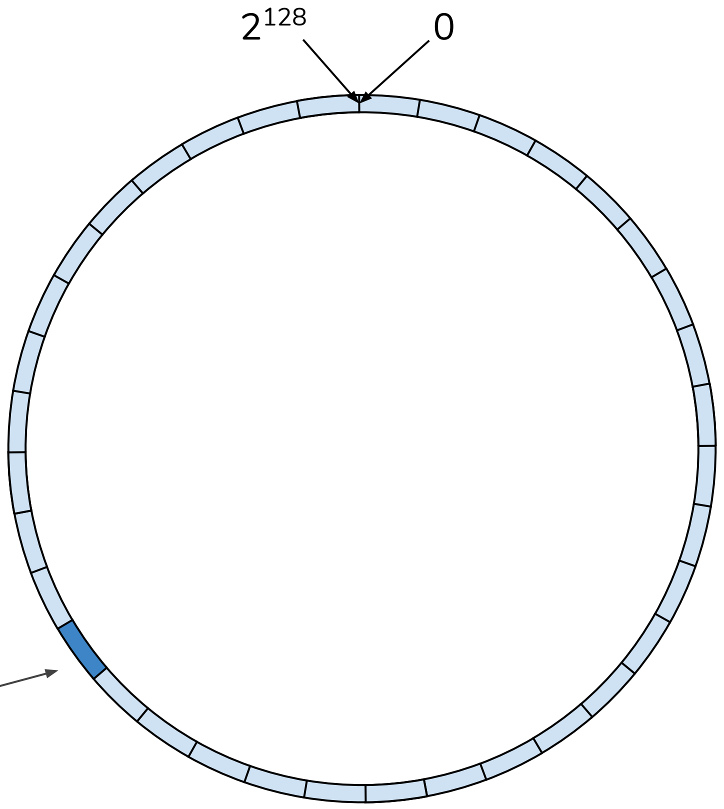
Uses the MD5 128-bit hash function on an object’s key to determine which server(s) should host the object.

$2^{128} \approx 340,282,366,920,938,000,000,000,000,000,000,000,000$

The Ring

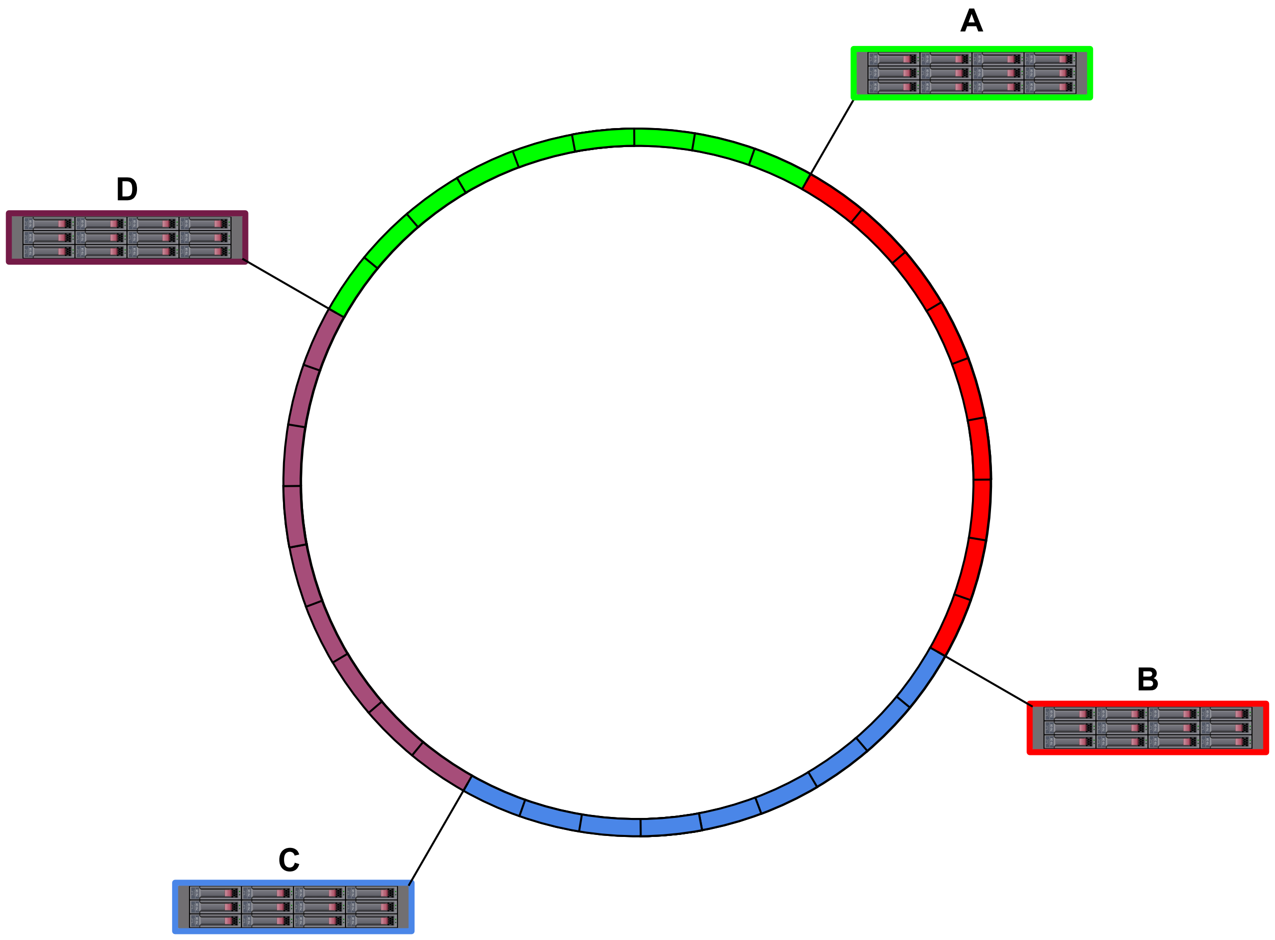


Hashing

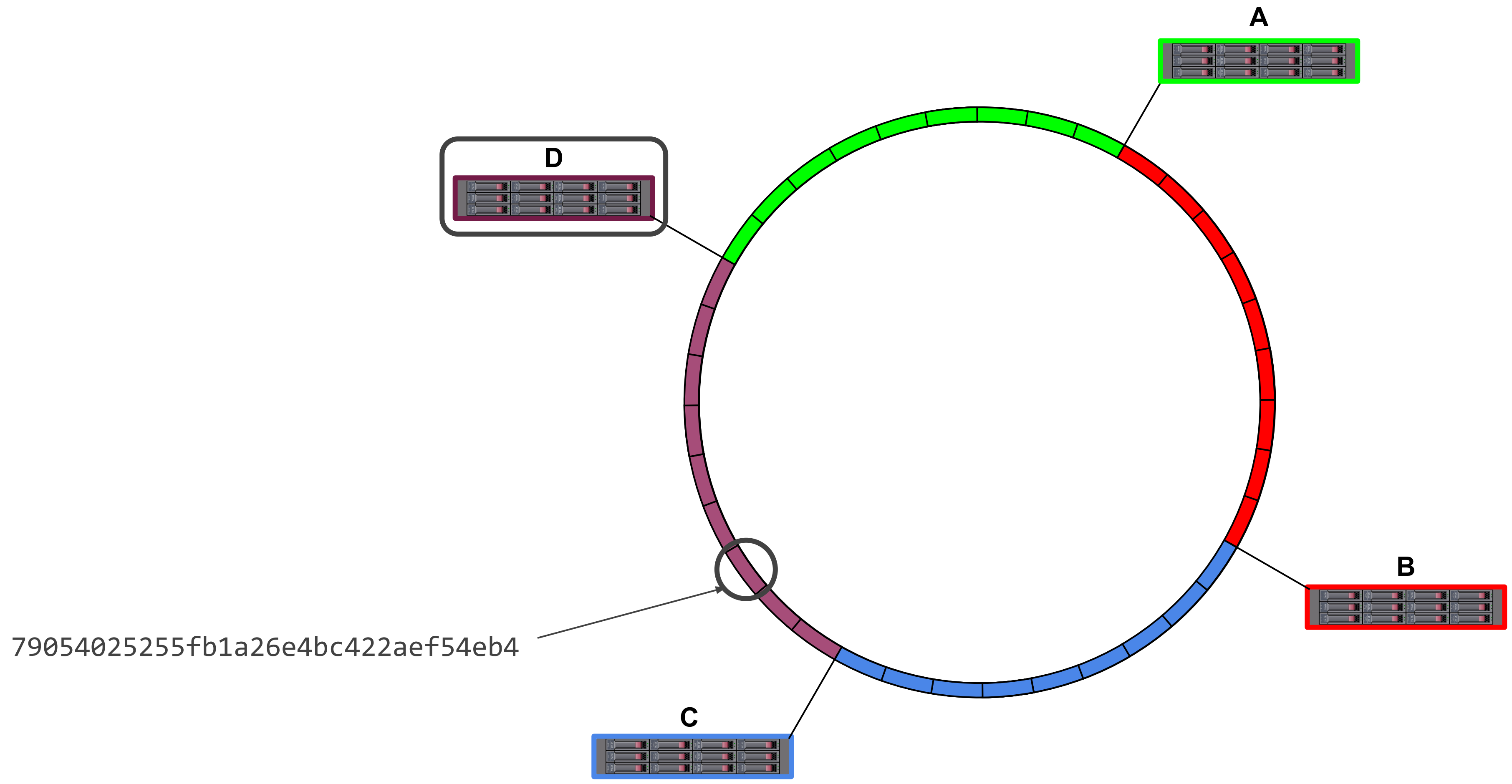


```
hash(“ProdID-101”)

↓

79054025255fb1a26e4bc422aef54eb4
```

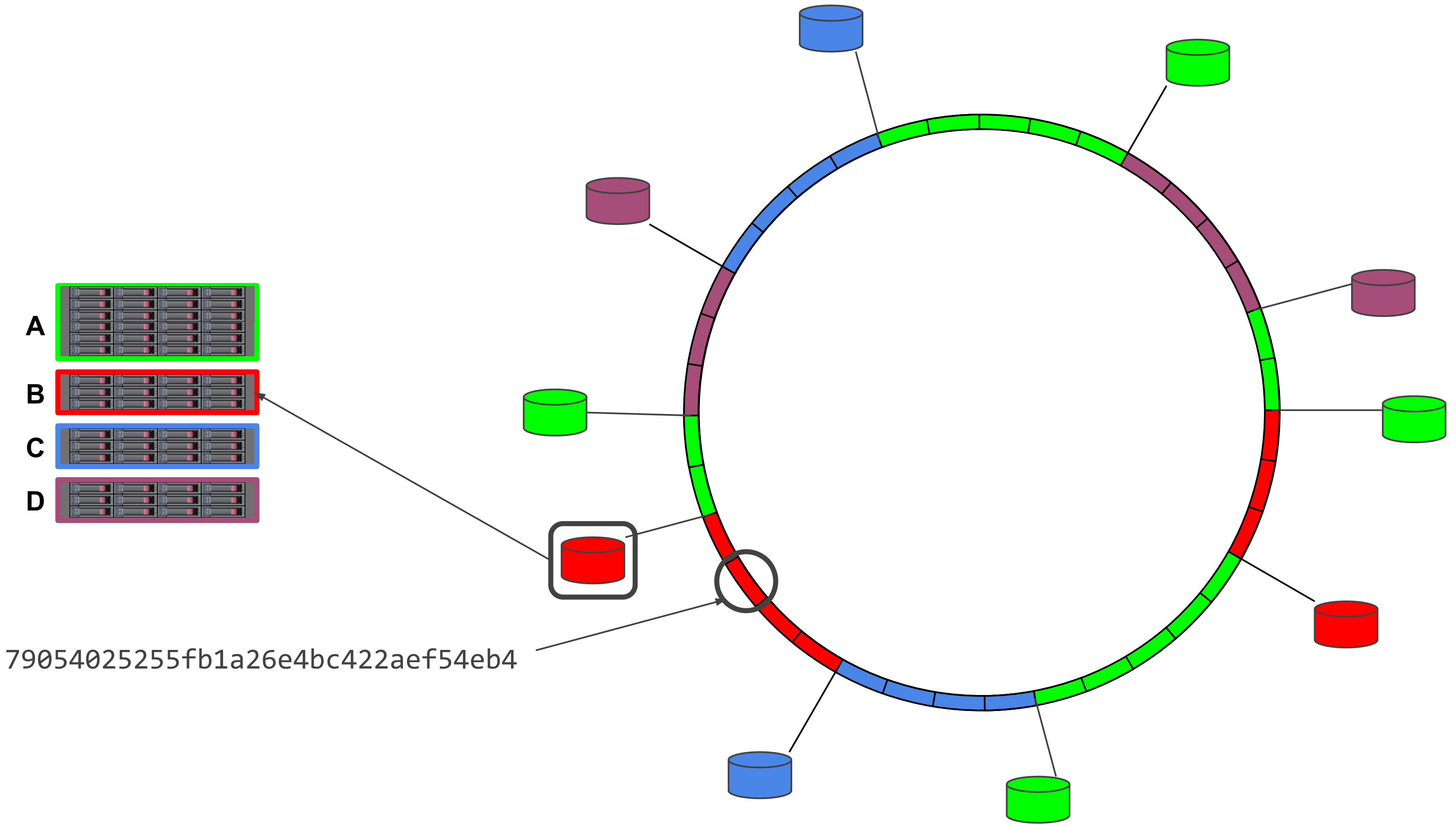
Partitioning



Distribution



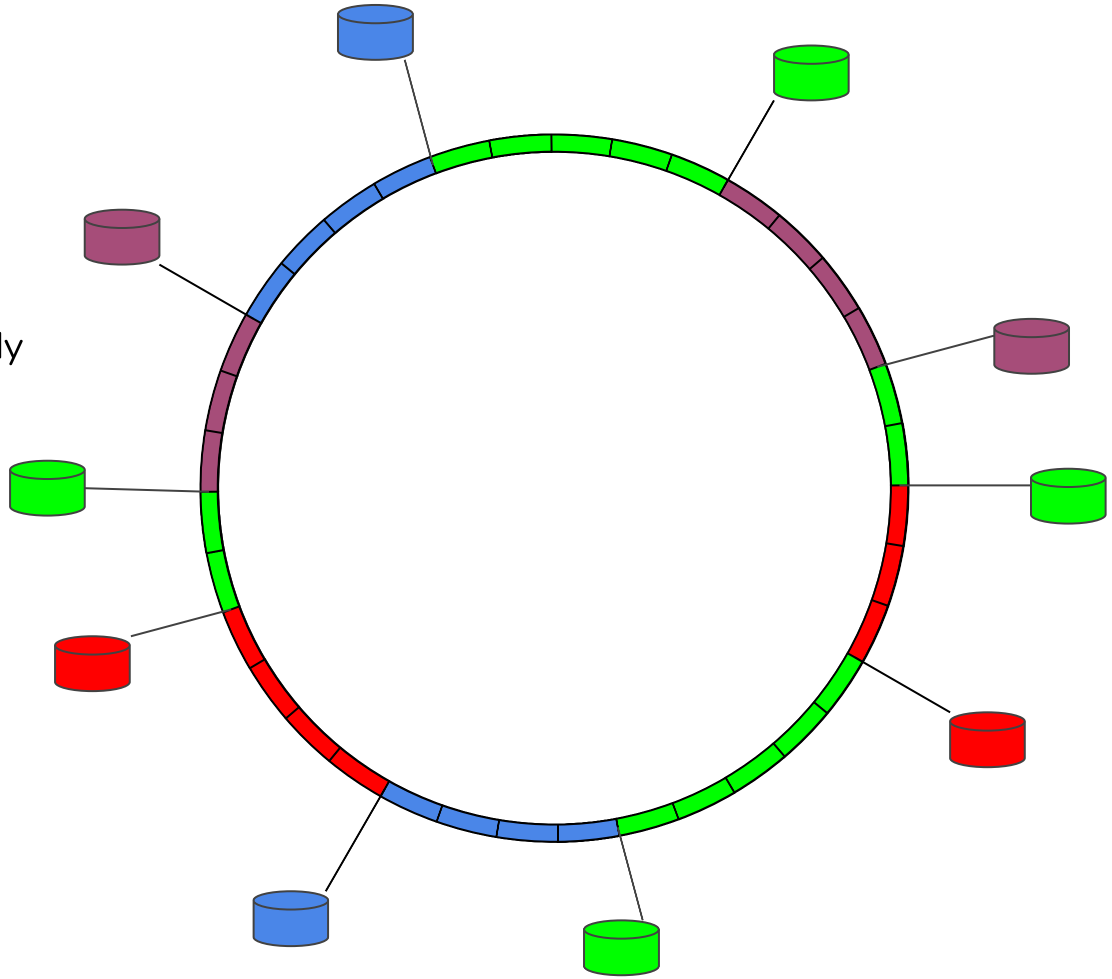
Virtual Nodes



Virtual Nodes

Advantages:

- Node failure proportionately affects all other nodes
- Node addition proportionately alleviates load on all other nodes
- Can account for heterogeneous node sizes.

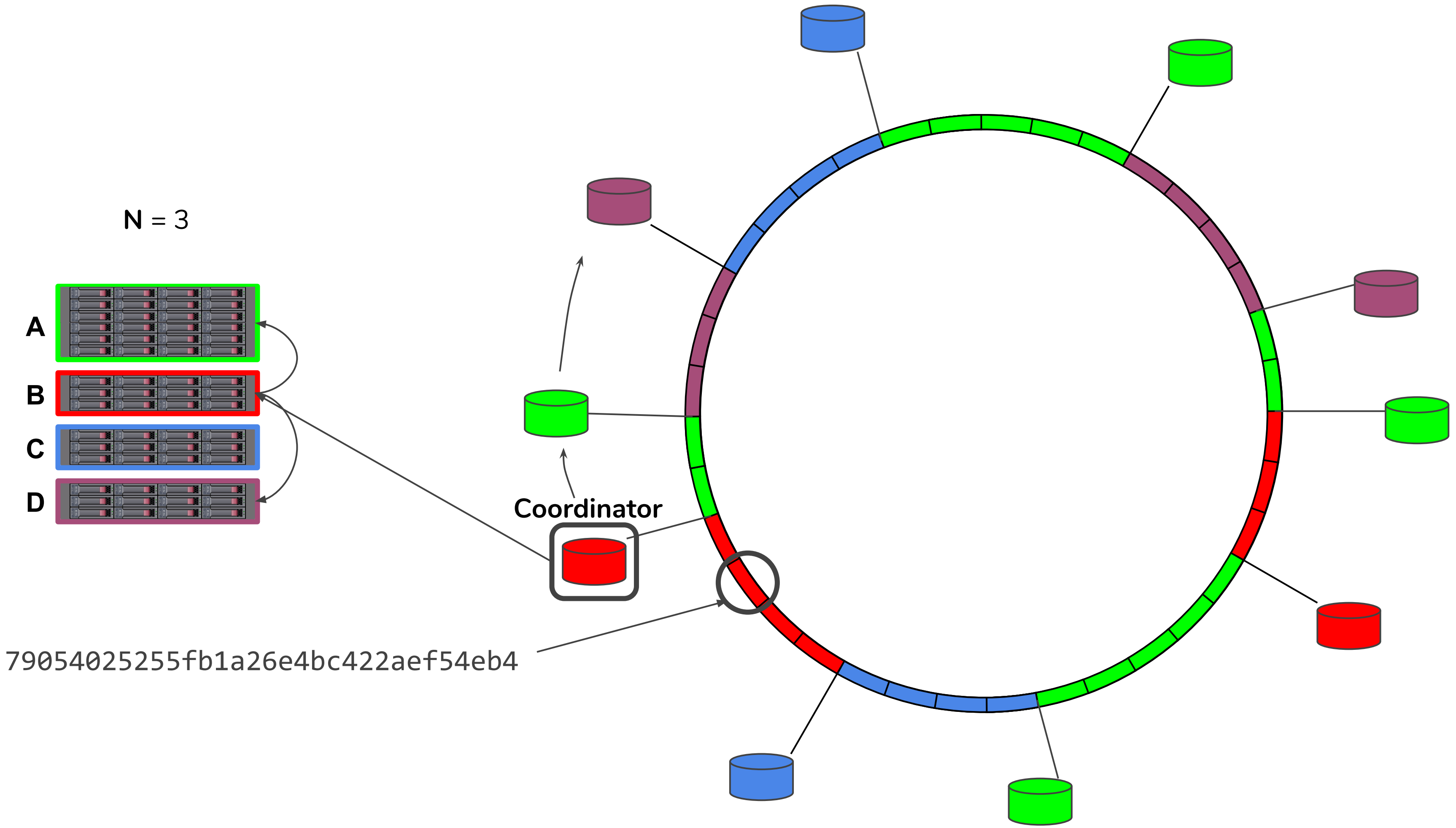


Replication

Increases data durability by replicating *N* copies of data, where *N* is configurable.

Each key is assigned a **coordinator node** who has the job of replicating the data to its *N-1* successors in the ring.

Replication



get() & put()

get() and put() requests can be received by any node, but will ultimately be forwarded to one of the top N nodes in the key’s preference list.

Request Forwarding:

1. Generic load balancer
2. Partition-aware routing

Data Versioning

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.

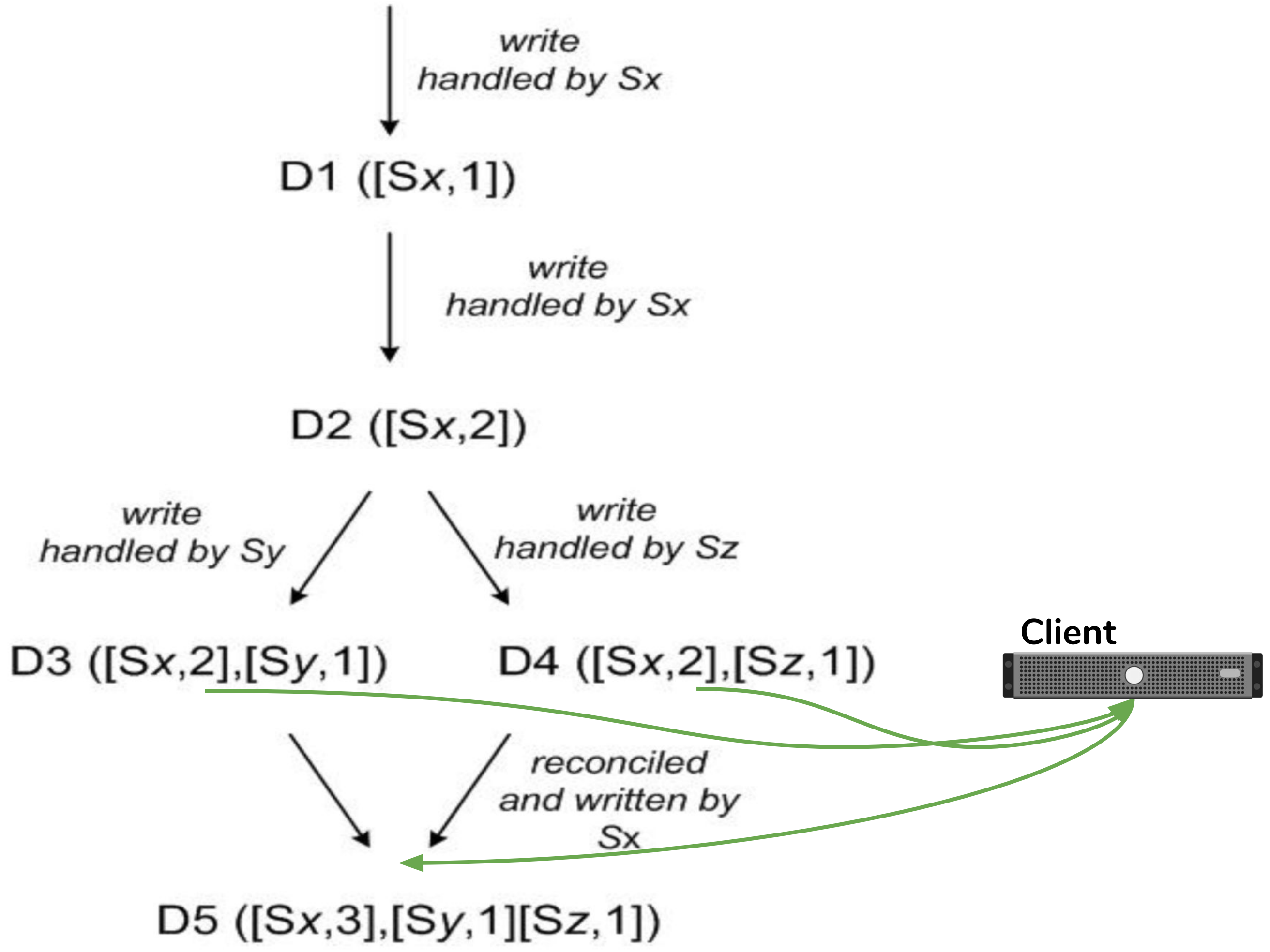
Data Versioning

Provides a mechanism for **eventual consistency** that increases availability for writes.

Allows multiple versions of the same data to exist simultaneously.

Uses vector clocks to capture causality between versions.

Resolves version branching via **syntactic reconciliation** (automatically) or **semantic reconciliation** (manually by the client).


write
handled by Sx
D1 ([Sx,1])
write
handled by Sx
D2 ([Sx,2])
write
handled by Sy
write
handled by Sz
D3 ([Sx,2],[Sy,1])
D4 ([Sx,2],[Sz,1])
Client
reconciled
and written by
Sx
D5 ([Sx,3],[Sy,1][Sz,1])

Quorum

Consistency is maintained using a quorum protocol:

$$R + W > N$$

R: Number of healthy replicas that need to participate in a successful read operation.

W: Number of healthy replicas that need to participate in a successful write operation.

Reduces response time when $R < N$ or $W < N$.

Hinted Handoff

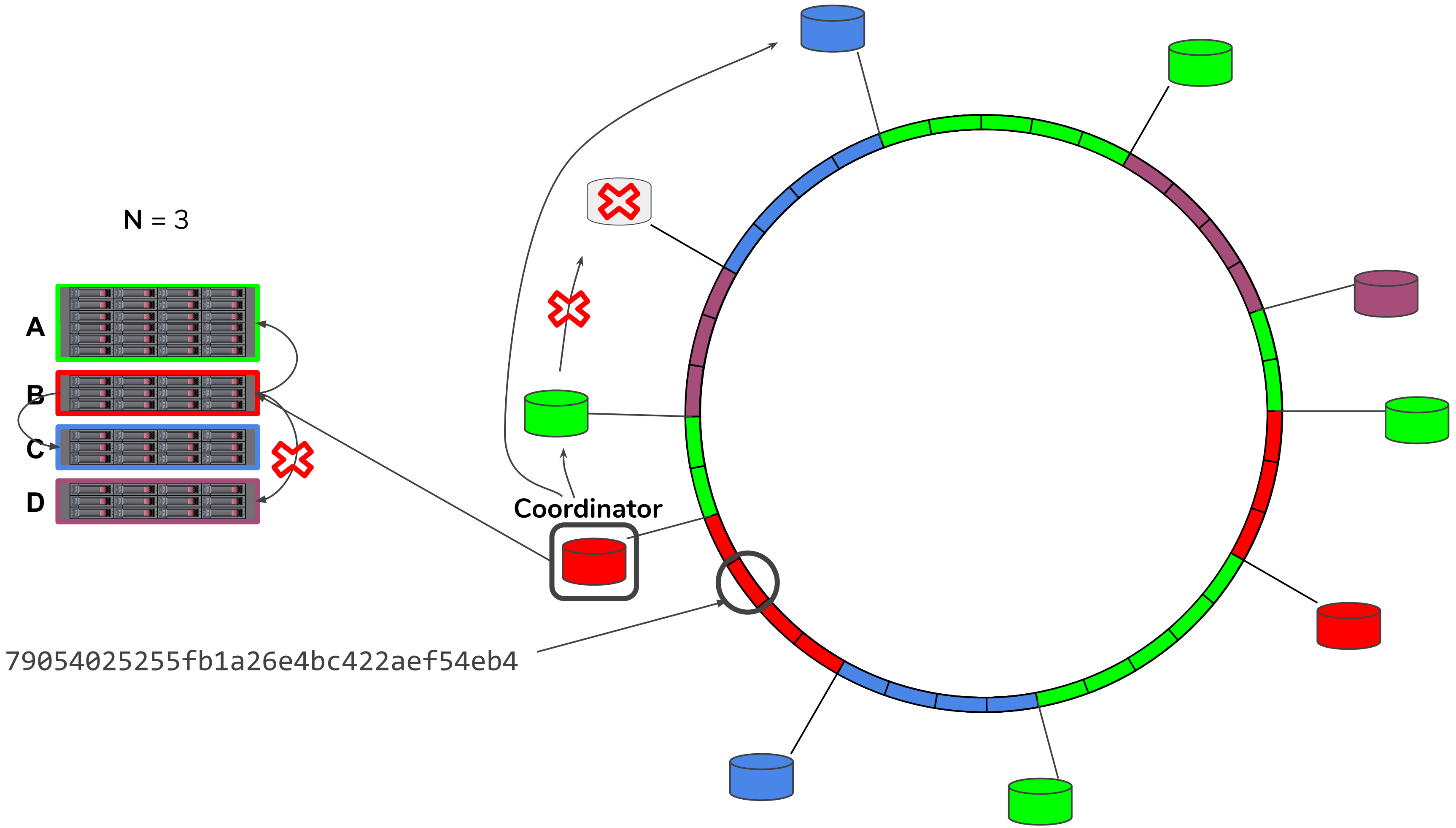
Problem	Technique	Advantage
Partitioning	Consistent Hashing	Incremental Scalability
High Availability for writes	Vector clocks with reconciliation during reads	Version size is decoupled from update rates.
Handling temporary failures	Sloppy Quorum and hinted handoff	Provides high availability and durability 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.

Hinted Handoff

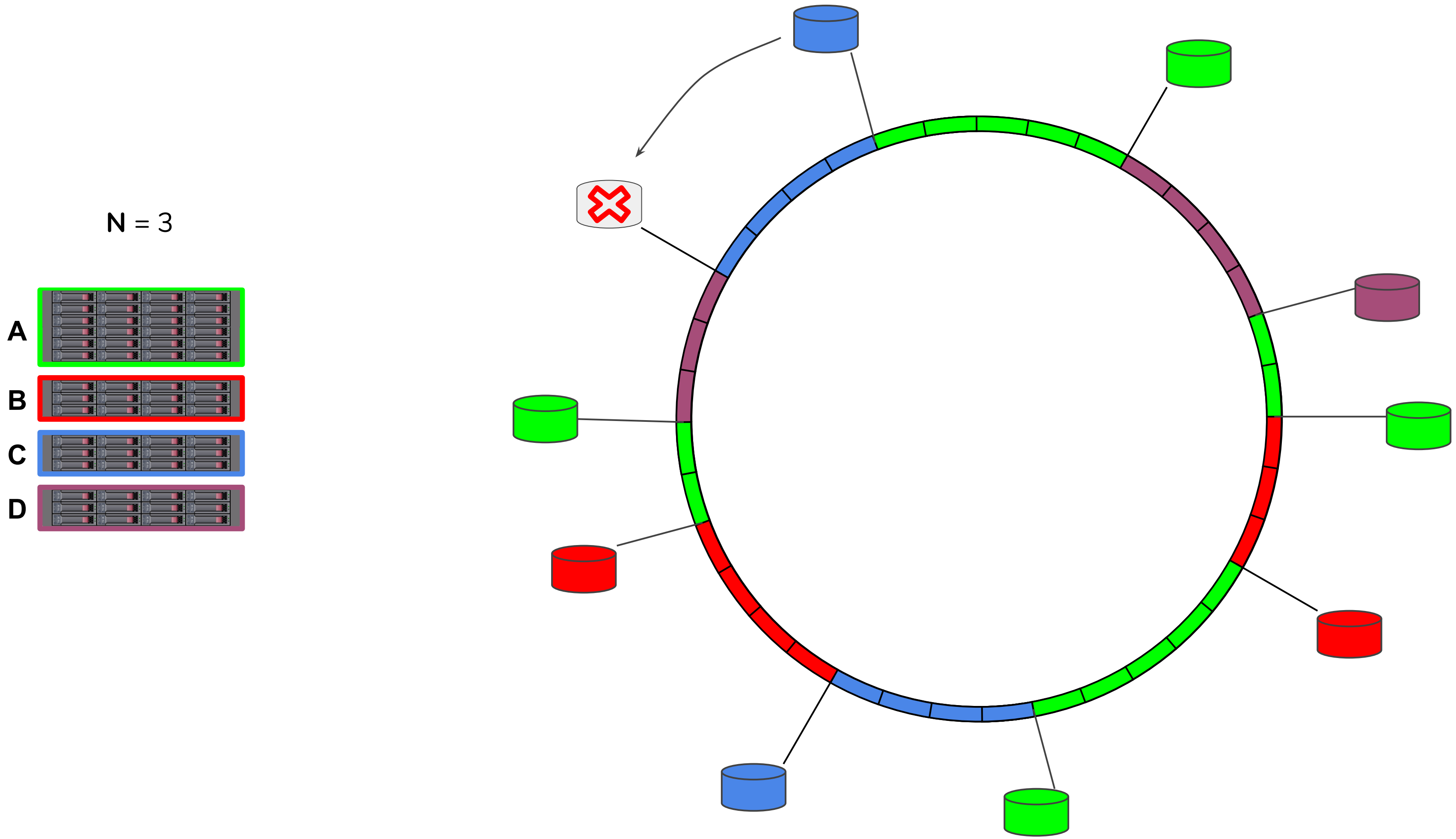
Sloppy quorum only requires the top N *healthy* nodes to participate in the operation.

In the event that one or more of the top N nodes for a key are unavailable, **hinted handoff** stores temporary copies of that key’s object on nodes further down the list. These copies contain a hint to allow them to be sent back to the correct node when it becomes available.

Hinted Handoff



Hinted Handoff



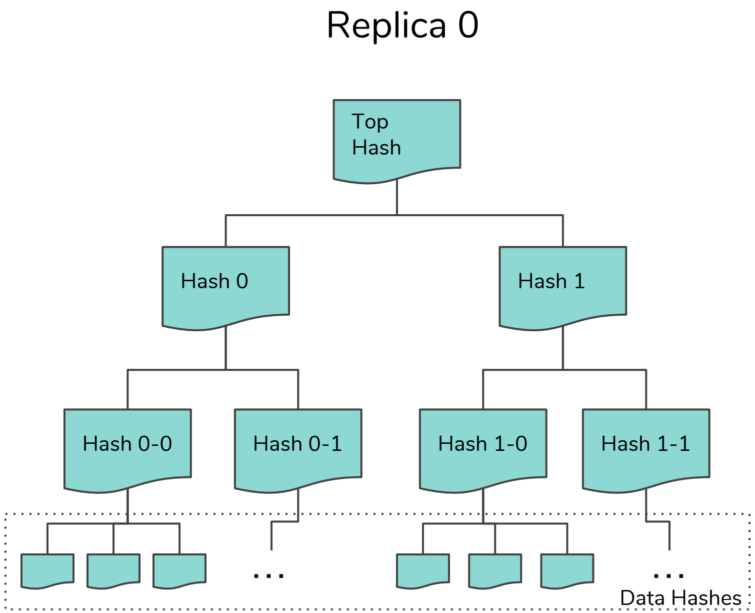
Replica Synchronization

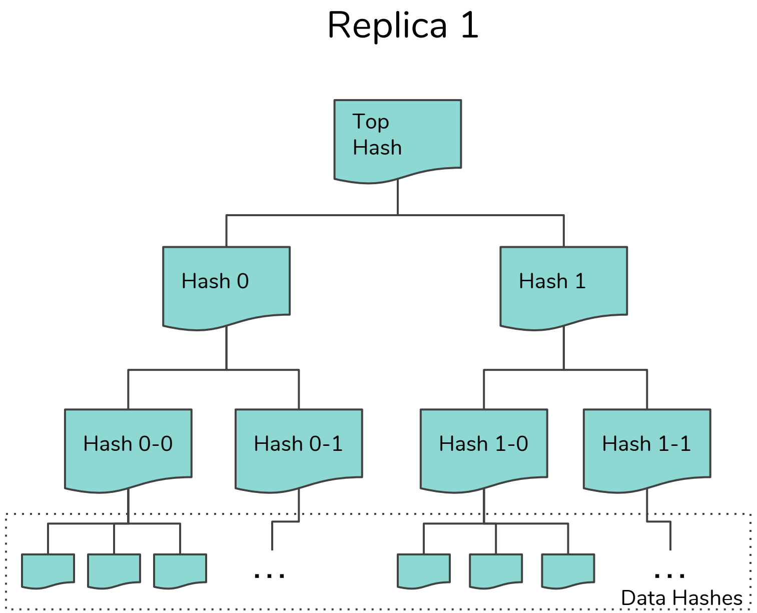
Problem	Technique	Advantage
Partitioning	Consistent Hashing	Incremental Scalability
High Availability for writes	Vector clocks with reconciliation during reads	Version size is decoupled from update rates.
Handling temporary failures	Sloppy Quorum and hinted handoff	Provides high availability and durability 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.

Replica Synchronization

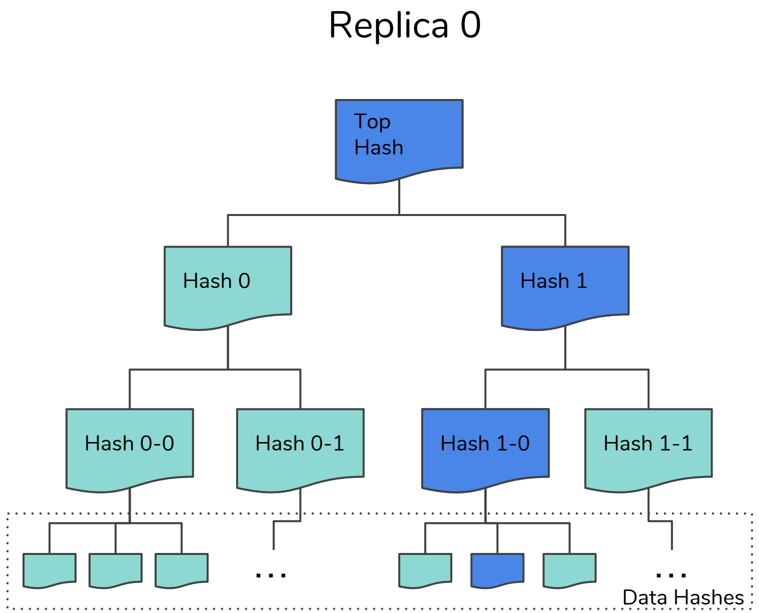
To maintain durability replicas are kept in sync using **Merkle trees**.

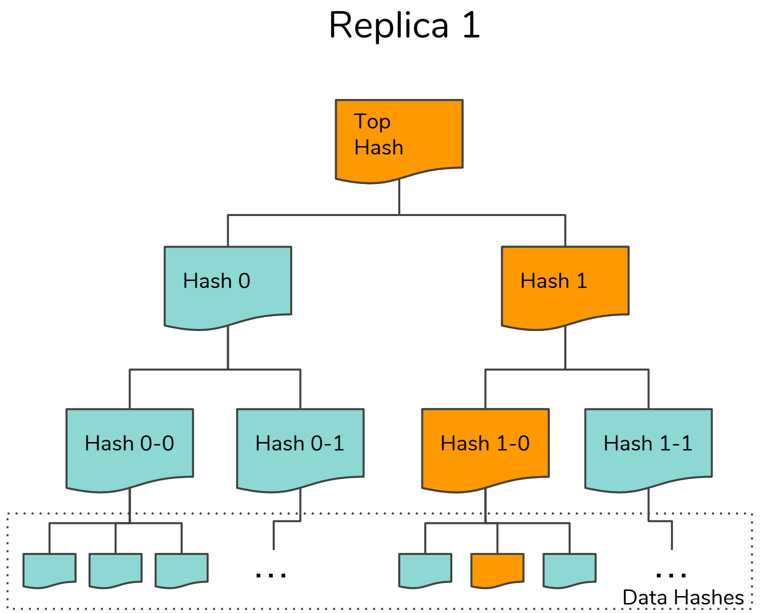
Merkle Syncing



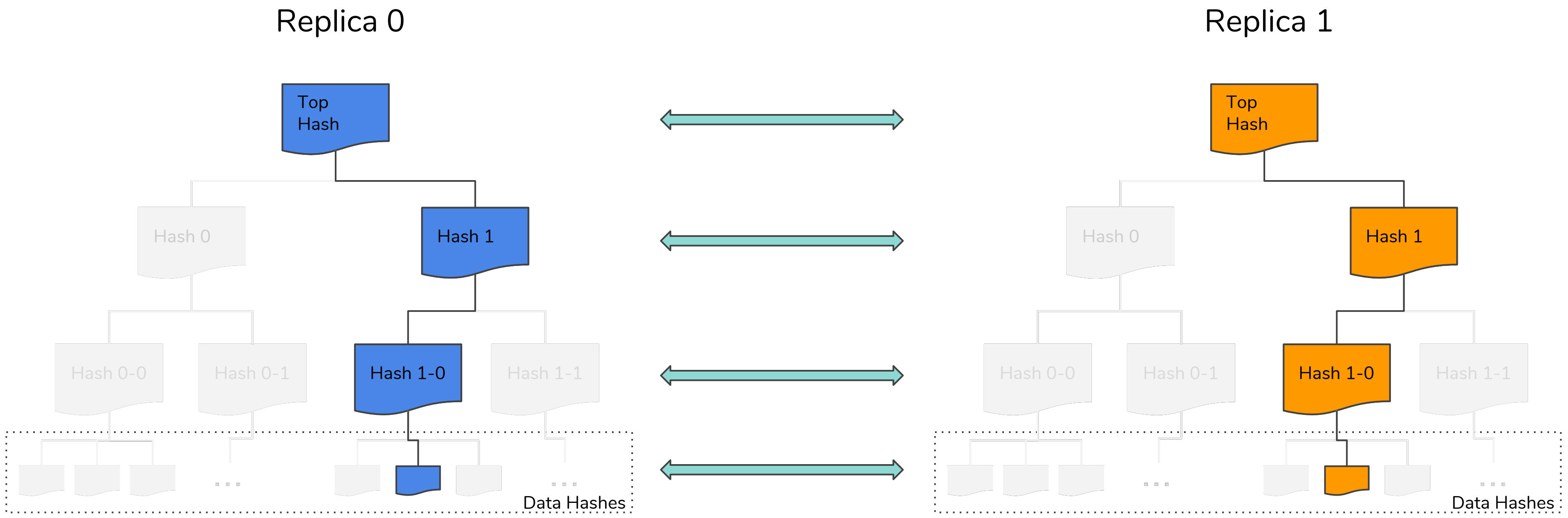


Merkle Syncing





Merkle Syncing



Membership & Failure Detection

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.

Failure Detection

Node failures are always treated as temporary, unless manually removed.

A purely **local (node-to-node) sense of failure** is sufficient; driven by client requests.

Adding/Removing Nodes

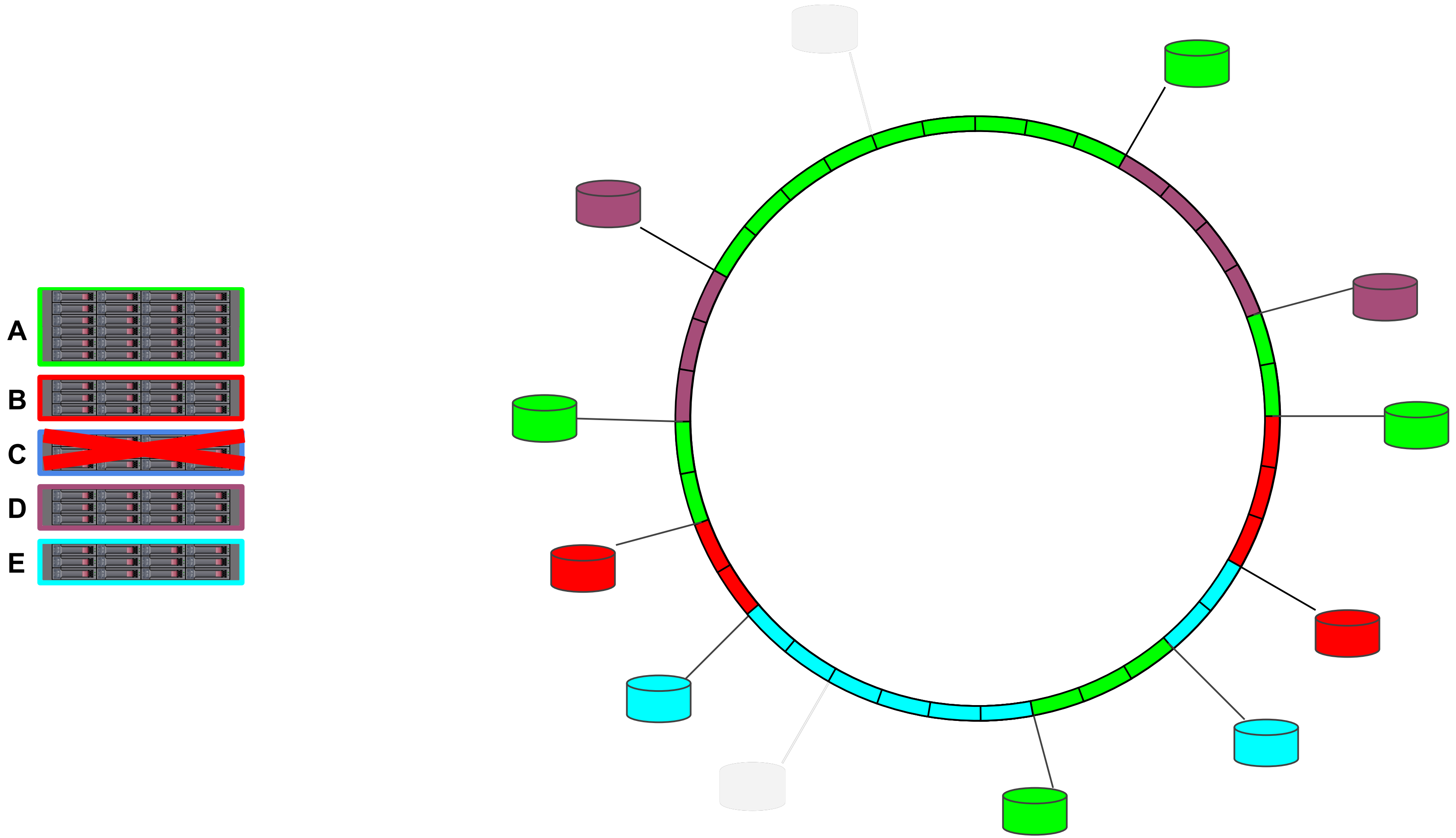
Adding and removing nodes are manual processes.

Addition results in the node being assigned a set of tokens and responsibility for key ranges.

Removal results in all other nodes gaining responsibility for additional key ranges.

Node membership histories are **gossiped** periodically between nodes.

Virtual Nodes



Implementation

Software Components

- Request coordination
- Membership and failure detection
- Local persistence engine

1. Berkeley Database (BDB) Transactional Data Store

2. BDB Java Edition

3. MySQL

4. In-memory buffer with persistent backing store

Request Coordination

- Built on top of an event driven messaging substrate.
- The Coordinator:

  - Executes read/write based on the request by the client.

- Each client request results in creation of State Machine on the node that received the client request.
- The State Machine contains:

  - logic for identifying nodes responsible for a key, send request, wait for response, process reply.

State Machine (Read)

- Each state machine instance handles one client request.
- For example, the read operation implements the following:
  1. Send read request to nodes
  2. Wait for minimum number of required requests
  3. If very few replies then discard/ fail the request
  4. Otherwise determine the data to be returned
  5. If versioning is enabled, perform syntactic reconciliation
  6. Wait for small time to receive outstanding response.
  7. **Read repair**

State Machine (Write)

- Write requests are coordinated by one of the top N nodes in the preference list.
- Drawback of selecting first node to coordinate the writes
- Select the node that has data that was read by preceding read [Read your writes consistency]

Experiences & Lessons Learned

The Patterns

- Client applications can tune the values of N, R and W according to the need.
- N : durability of each object.
- Values of W and R impact object availability, durability and consistency.
- W = 1
- Low values of W and R decreases durability.
- Vulnerability Window
- (N,R,W) : (3,2,2)

The Patterns

- **Business logic specific reconciliation :**

  Each data object is replicated across multiple nodes.

- **Timestamp based reconciliation:**

  In case of divergent version, dynamo performs simple timestamp - “last write wins”

- **High performance read engine.**

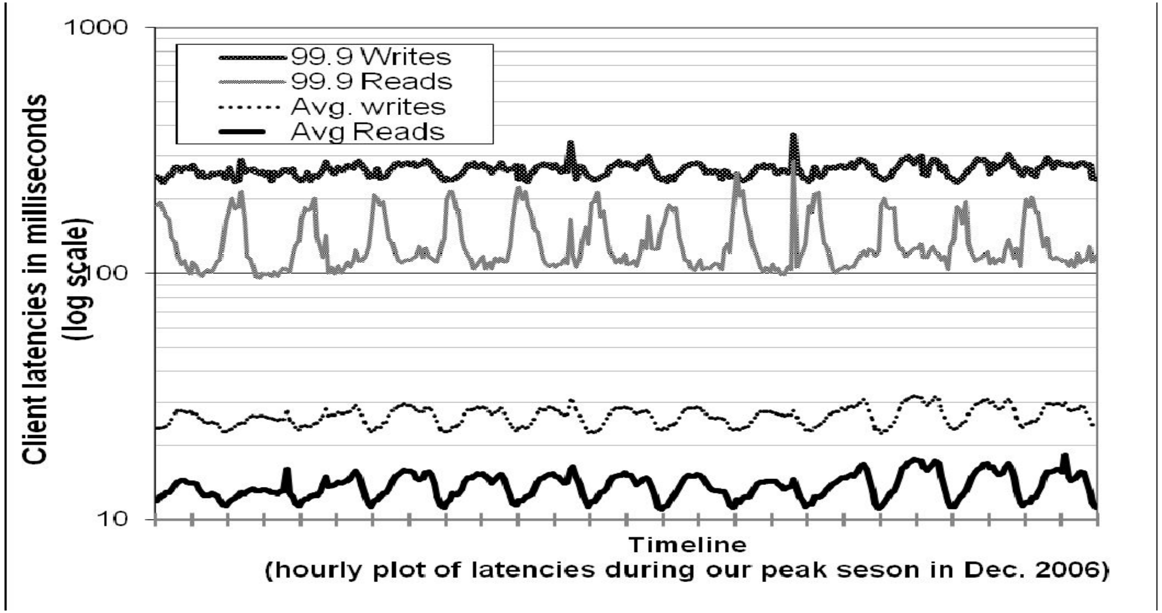
  High read request rate. R =1 and W = N

Balancing Performance and Durability

Principle design goal: To build highly available data store.

- A typical SLA is that 99.9% read and write execute within 300ms.
- Maintaining Object Buffer for higher levels of performance.

Improving performance at 99.9 percentile



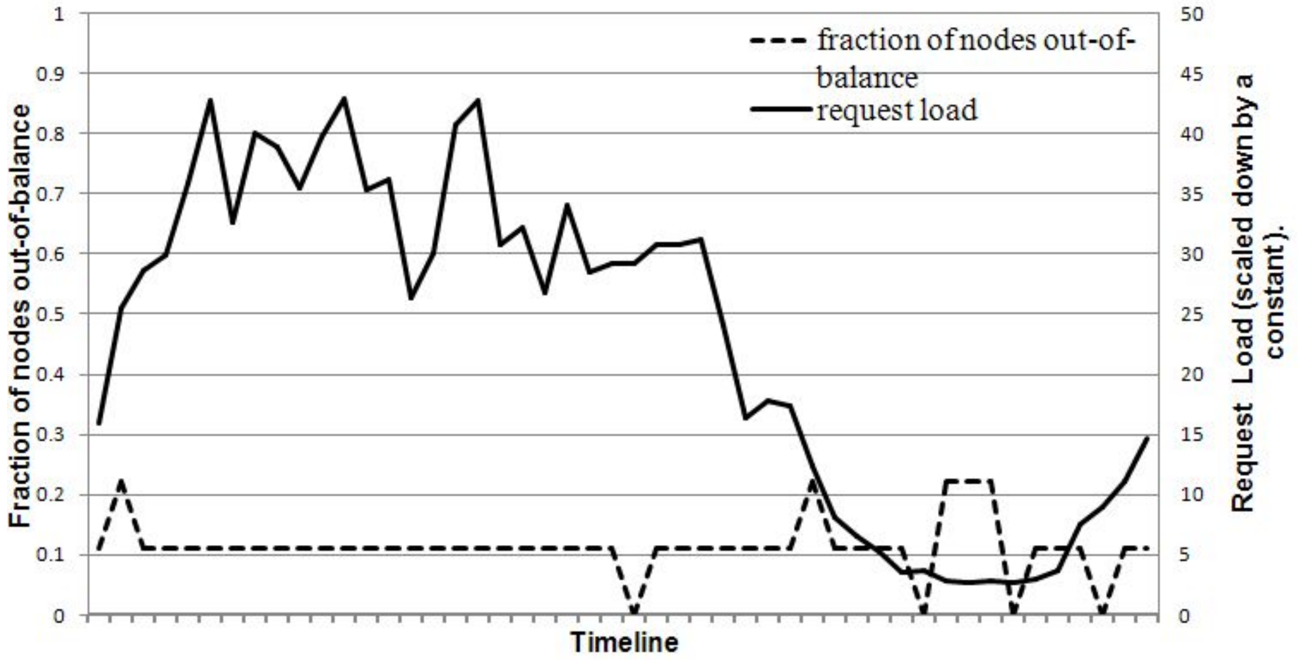
Ensuring Uniform Load distribution

- Dynamo uses consistent hashing to partition its key space across it’s replicas.
- **Assumption:** Enough keys so that load of handling popular keys can be spread across the nodes uniformly.
- **Measure imbalance ?**

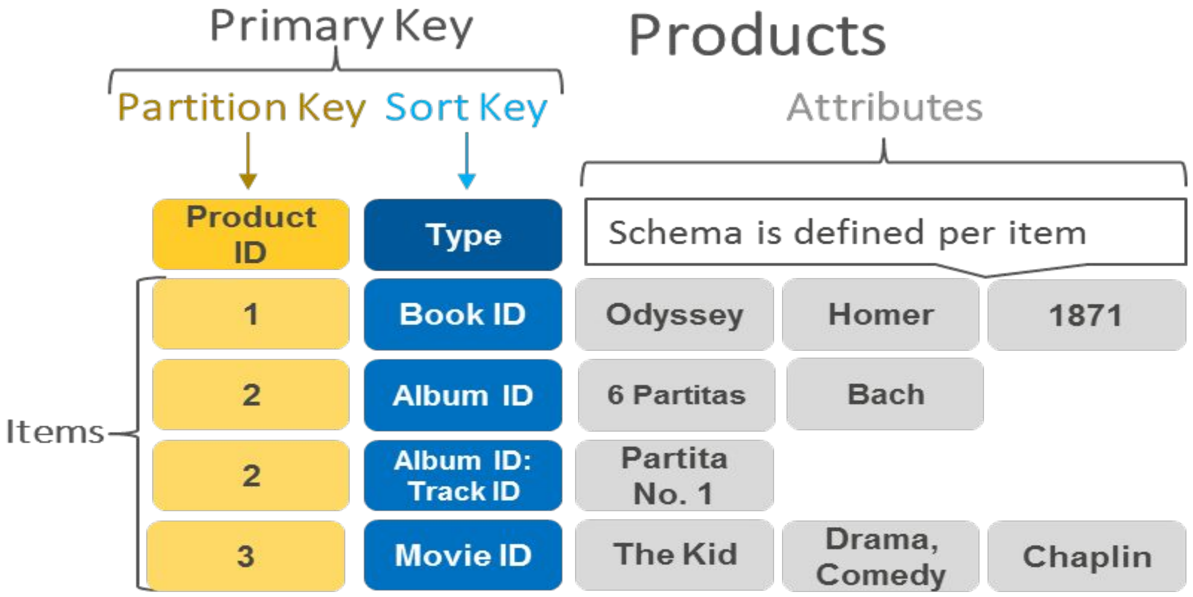
  - Node’s request load deviates from average by a value less than a certain threshold

- High load vs low load

Out of Balance: High Load vs Low Load

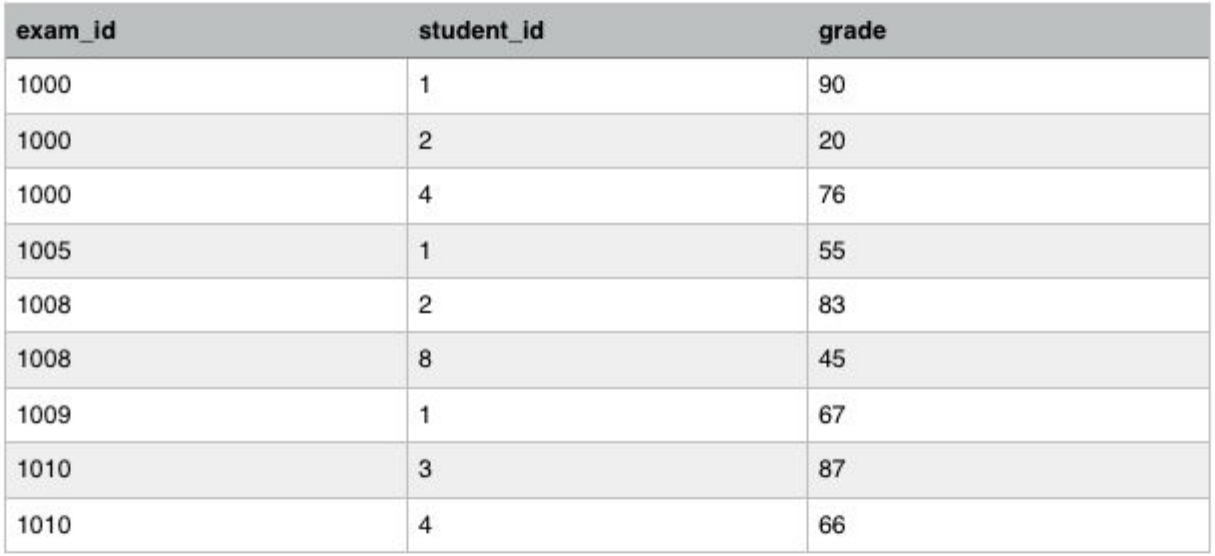


Partitioning Key



Source: Choosing the right dynamodb partition key (aws.amazon.com)

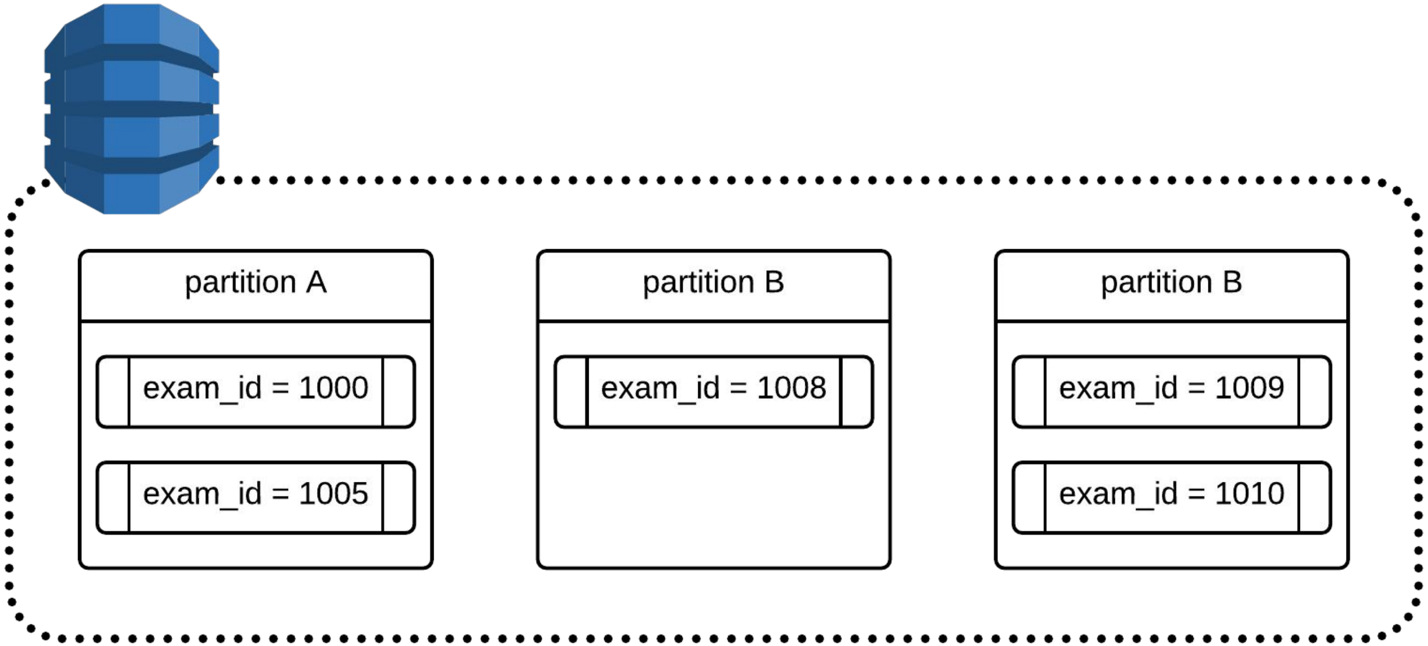
Partitioning Key



exam_id	student_id	grade
1000	1	90
1000	2	20
1000	4	76
1005	1	55
1008	2	83
1008	8	45
1009	1	67
1010	3	87
1010	4	66

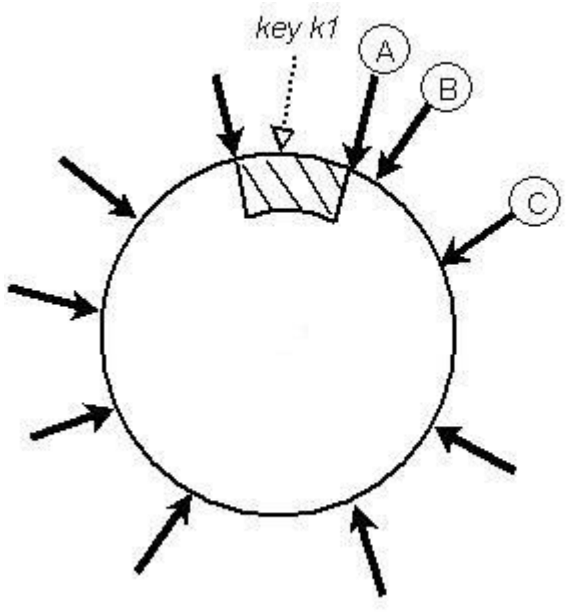
Source: Deep dive into dynamoDB solutions [shinesolutions.com]

Partitioning Key



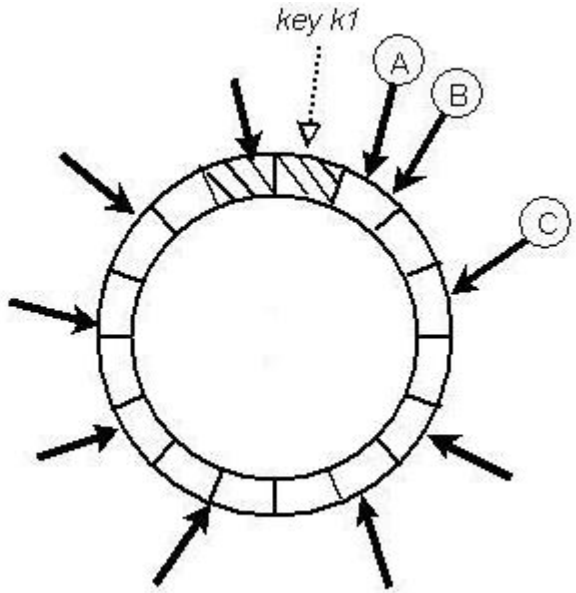
Source: Deep dive into dynamoDB solutions [shinesolutions.com]

Partitioning Key: Strategy 1



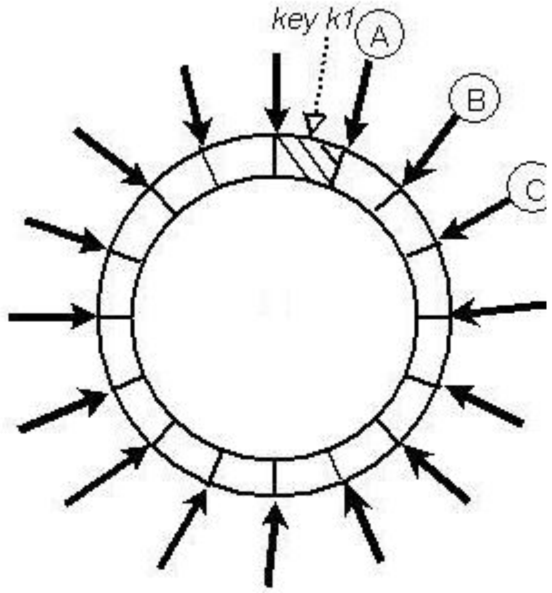
T random tokens per node and partition by token value

Partitioning Key: Strategy 2



T random tokens per node and equal sized partition

Partitioning Key: Strategy 3



Q/S tokens per node, equal sized partitions

Client- driven vs Server Driven coordination

	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

Conclusion

- Incrementally Scalable
- Allow users to customize their storage system to meet their desired performance.
- **Eventual - Consistent Storage system** can be a building block for highly available applications

Thank You