Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels

Amazon.com

Presented by: Haowei Cai

Dynamo

By Amazon

Huge infrastructure; Reliability is key. Customer oriented business.

Published @ October 2007 @ SOSP

Linkedin -> Voldemort, Facebook -> Cassandra

Authors nearly got fired from Amazon for publishing

DynamoDB @ 2012

Motivation

Amazon:

- Huge infrastructure
- Customer oriented business
- Different kinds of application
- Primary-key access

Motivation

Traditional:

- Relational Database Management System (RDBMS)
  - Expansive & Complex
  - Not flexible

Motivation

Build a distributed storage system:

Huge infrastructure -> Scale, Highly available

Customer oriented business -> Guarantee Service Level Agreements (SLA)

Different kinds of application -> Flexible

Primary-key access -> Simple: key-value

Service Level Agreement (SLA)

Application can deliver its functionality in a bounded time:

Every dependency in the platform needs to deliver its functionality with even tighter bounds.

Example: service guaranteeing that it will provide a response within 300ms for 99.9% of its requests for a peak client load of 500 requests per second.

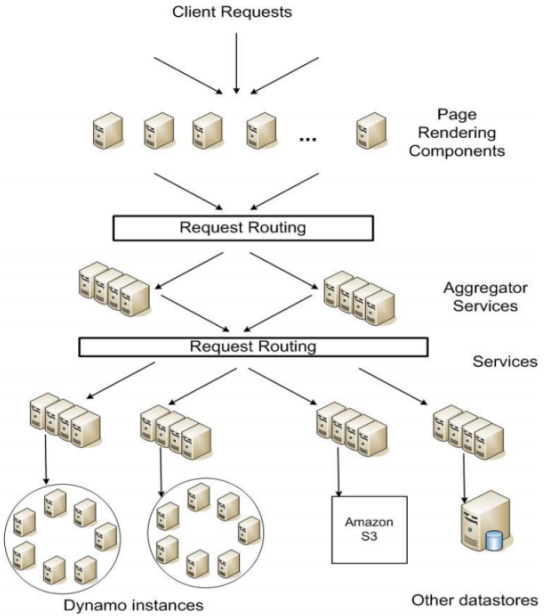


Figure 1: Service-oriented architecture of Amazon's platform

System Assumption and Requirements

Query Model: simple read and write operations to a data item that is uniquely identified by a key.

ACID Properties: Atomicity, Consistency, Isolation, Durability.

Efficiency: latency requirements which are in general measured at the 99.9th percentile of the distribution.

Other Assumptions: operation environment is assumed to be non-hostile and there are no security related requirements such as authentication and authorization.

Design Consideration

Sacrifice strong consistency for availability

Conflict resolution is executed during read instead of write, i.e. “always writeable”.

Other principles:

- Incremental scalability
- Symmetry
- Decentralization
- Heterogeneity

System Interface

Objects stored with a key using:

- Get(key): locates object with key and returns object or list of objects with a context
- Put(key, context): places an object at a replica along with the key and context
- Context: metadata about object

System Architecture

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	Preservers symmetry and avoids having a centralized registry for storing membership and node liveness information

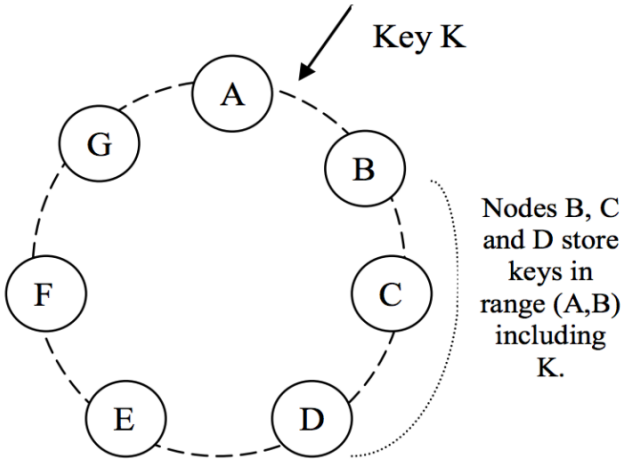
System Architecture

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	Preservers symmetry and avoids having a centralized registry for storing membership and node liveness information

Partitioning Algorithm

Consistent hashing: the output range of a hash function is treated as a fixed circular space or “ring”.

Virtual Nodes: Each node can be responsible for more than one virtual node.



Advantage of using virtual nodes

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, the newly available node 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.

System Architecture

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	Preservers symmetry and avoids having a centralized registry for storing membership and node liveness information

Replication

Each data item is replicated at N hosts (N is configured by developer)

preference list: The list of nodes that is responsible for storing a particular key

Data Versioning

A put() call may return to its caller before the update has been applied at all the replicas

A get() call may return many versions of the same object.

Challenge: an object having distinct version sub-histories, which the system will need to reconcile in the future.

Solution: uses vector clocks in order to capture causality between different versions of the same object.

Vector Clock

A vector clock is a list of (node, counter) pairs.

Every version of every object is associated with one vector clock.

If the 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.

Vector Clock Example

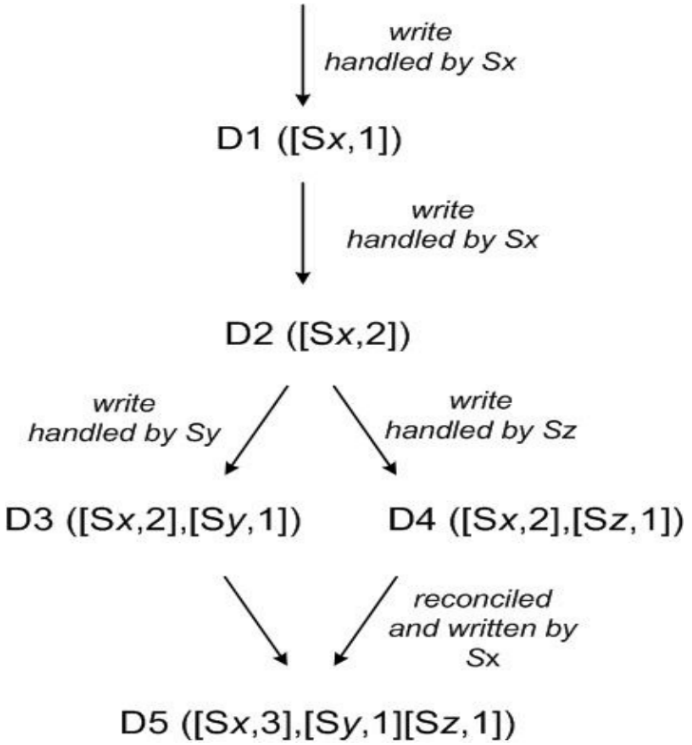


Figure 3: Version evolution of an object over time.

System Architecture

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	Preservers symmetry and avoids having a centralized registry for storing membership and node liveness information

Execution of get() and put() operations

Route its request through a generic load balancer that will select a node based on load information.

Use a partition-aware client library that routes requests directly to the appropriate coordinator nodes.

Sloppy Quorum

R/W is the minimum number of nodes that must participate in a successful read/write operation.

Setting R + W > N yields a quorum-like system.

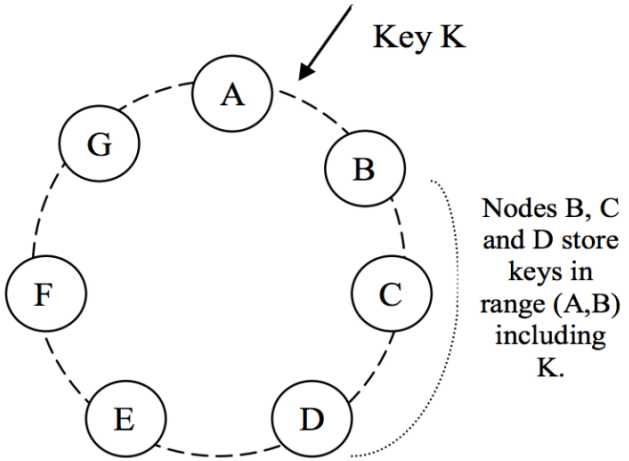
In this model, the latency of a get (or put) operation is dictated by the slowest of the R (or W) replicas. For this reason, R and W are usually configured to be less than N, to provide better latency.

Hinted Handoff

Assume N = 3. When A is temporarily down or unreachable during a write, send replica to D.

D is hinted that the replica is belong to A and it will deliver to A when A is recovered.

Again: “always writeable”



System Architecture

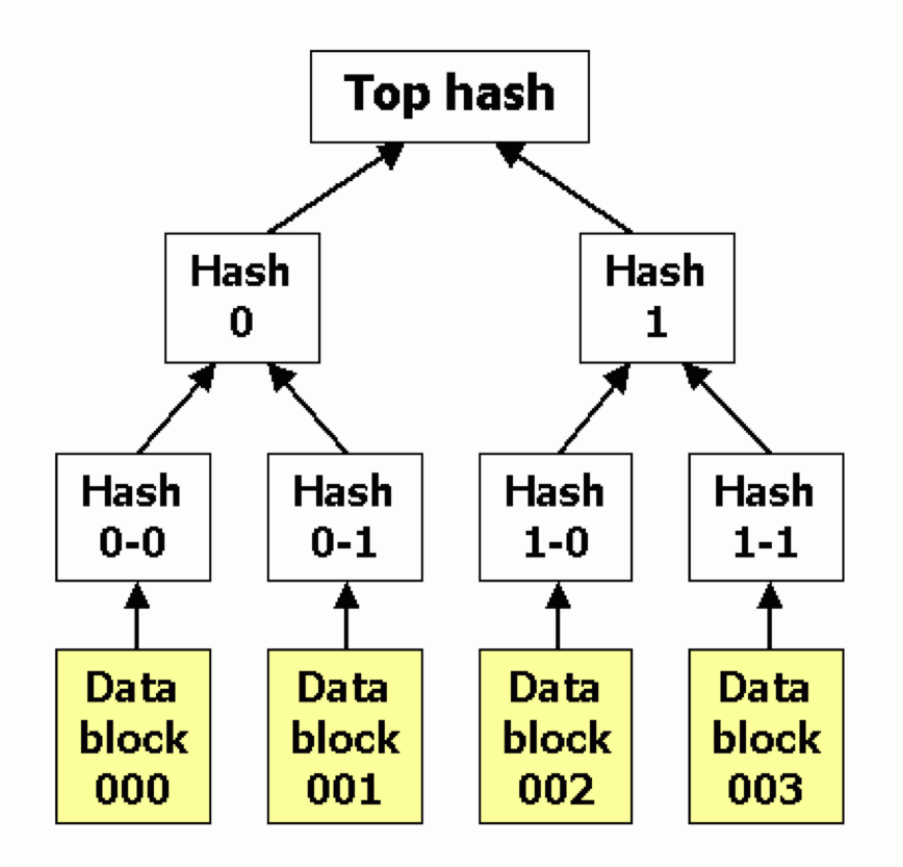
Problem	Technique	Advantage
Partitioning	Consistent Hashing	Incremental Scalability
High Availability for writes	Vector clocks with reconciliation during reads	Version size is decoupled from update rates
Handling temporary failures	Sloppy Quorum and hinted handoff	Provides high availability and durability 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	Preservers symmetry and avoids having a centralized registry for storing membership and node liveness information

Replica Synchronization

Anti-entropy using Merkle tree

A hash tree where leaves are hashes of the values of individual keys

Parent nodes higher in the tree are hashes of their respective children



System Architecture

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	Preservers symmetry and avoids having a centralized registry for storing membership and node liveness information

Membership and Failure Detection

Internal: Gossip based protocol leads to eventual consistent membership list

External: Seed nodes, known by all nodes in system

Implementation

Each storage node has three main software components:

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

Request Coordination

Local persistence engine

Local persistence component allows for different storage engines to be plugged in:

Berkeley Database (BDB) Transactional Data Store: object of tens of kilobytes

MySQL: object of > tens of kilobytes

BDB Java Edition, etc.

Experiences & Lessons Learned

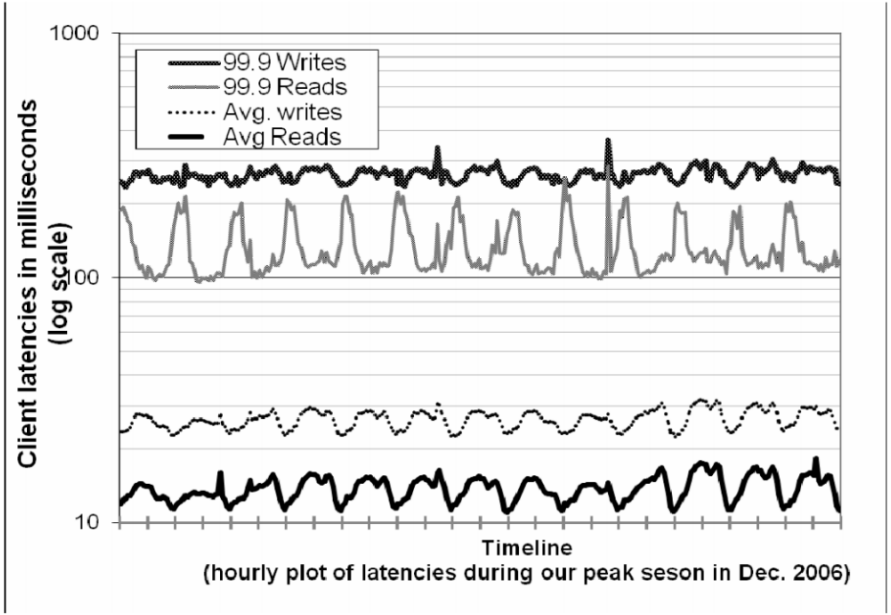


Figure 4: Average and 99.9 percentiles of latencies for read and write requests during our peak request season of December 2006. The intervals between consecutive ticks in the x-axis correspond to 12 hours. Latencies follow a diurnal pattern similar to the request rate and 99.9 percentile latencies are an order of magnitude higher than averages

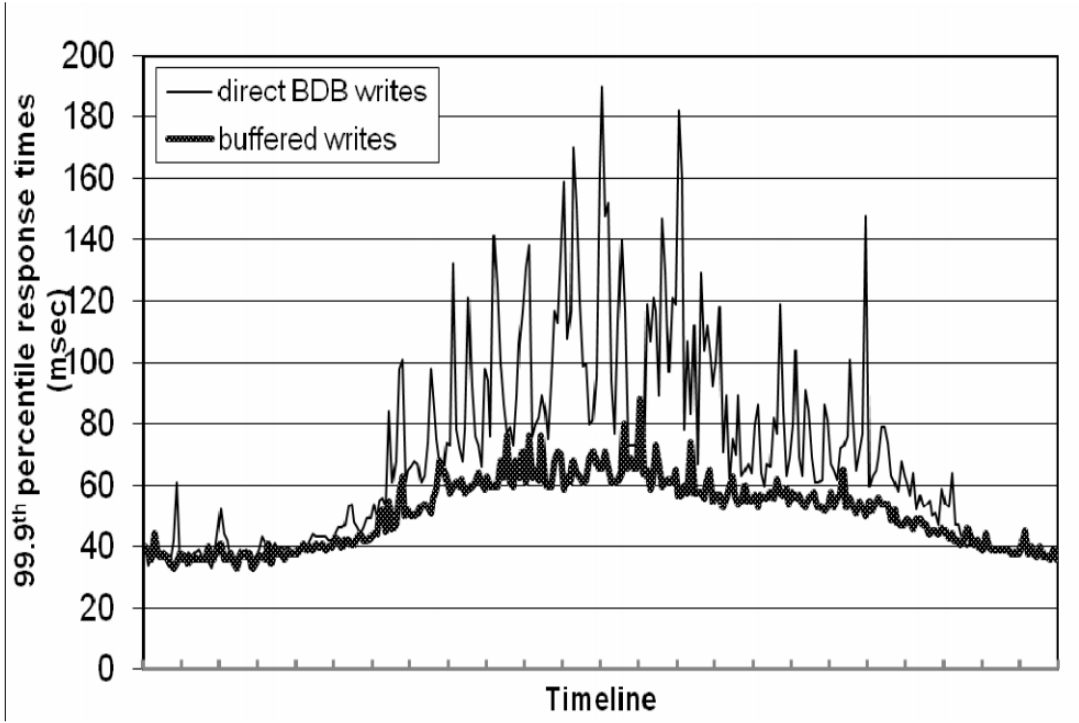


Figure 5: Comparison of performance of 99.9th percentile latencies for buffered vs. non-buffered writes over a period of 24 hours. The intervals between consecutive ticks in the x-axis correspond to one hour.

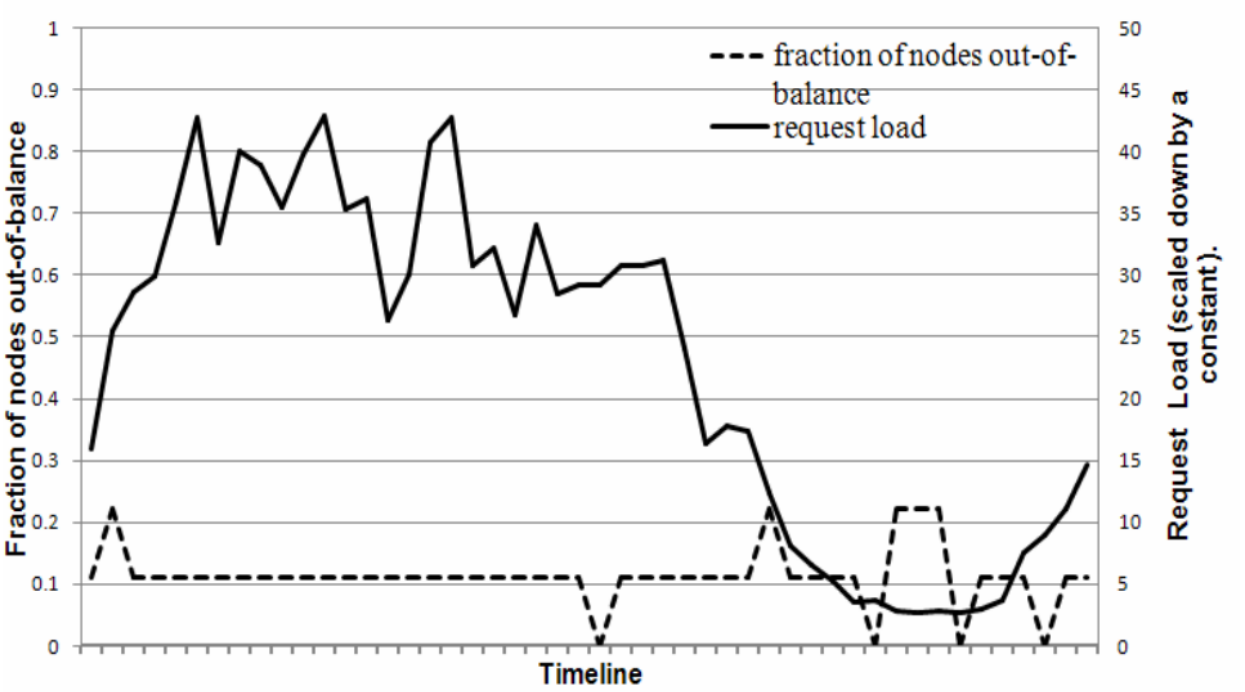


Figure 6: Fraction of nodes that are out-of-balance (i.e., nodes whose request load is above a certain threshold from the average system load) and their corresponding request load. The interval between ticks in x-axis corresponds to a time period of 30 minutes.

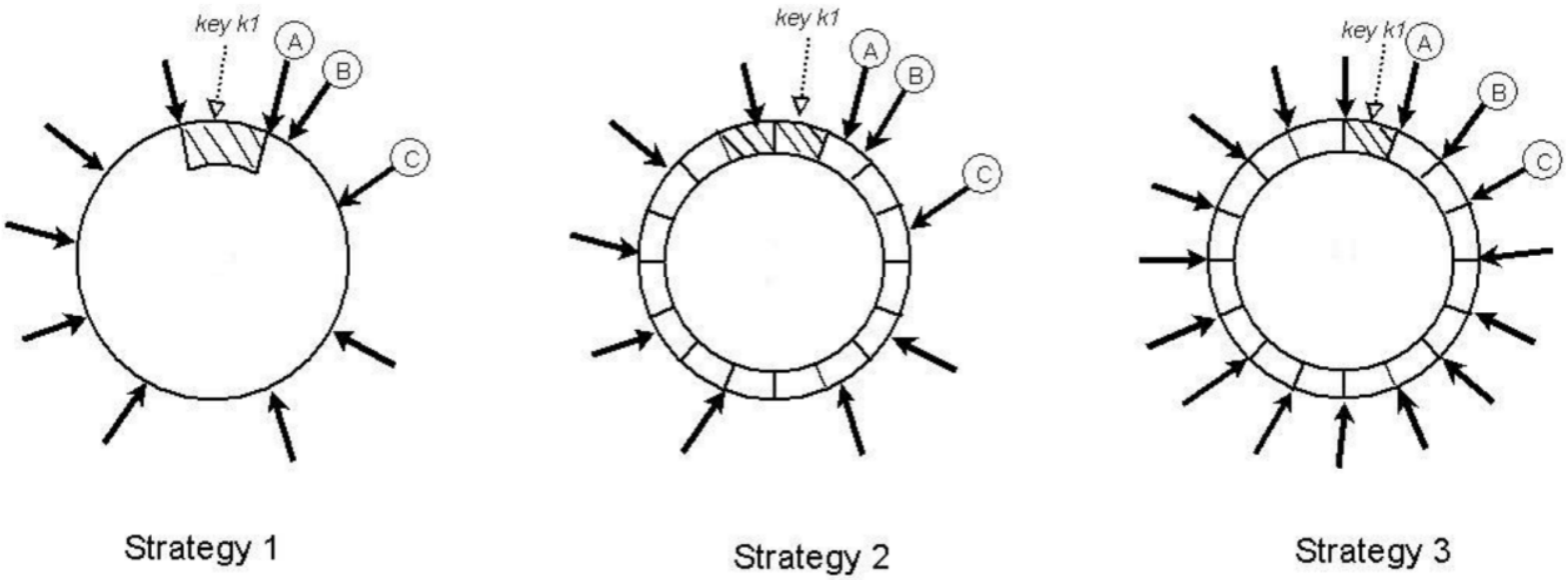


Figure 7: Partitioning and placement of keys in the three strategies. A, B, and C depict the three unique nodes that form the preference list for the key k1 on the consistent hashing ring (N=3). The shaded area indicates the key range for which nodes A, B, and C form the preference list. Dark arrows indicate the token locations for various nodes.

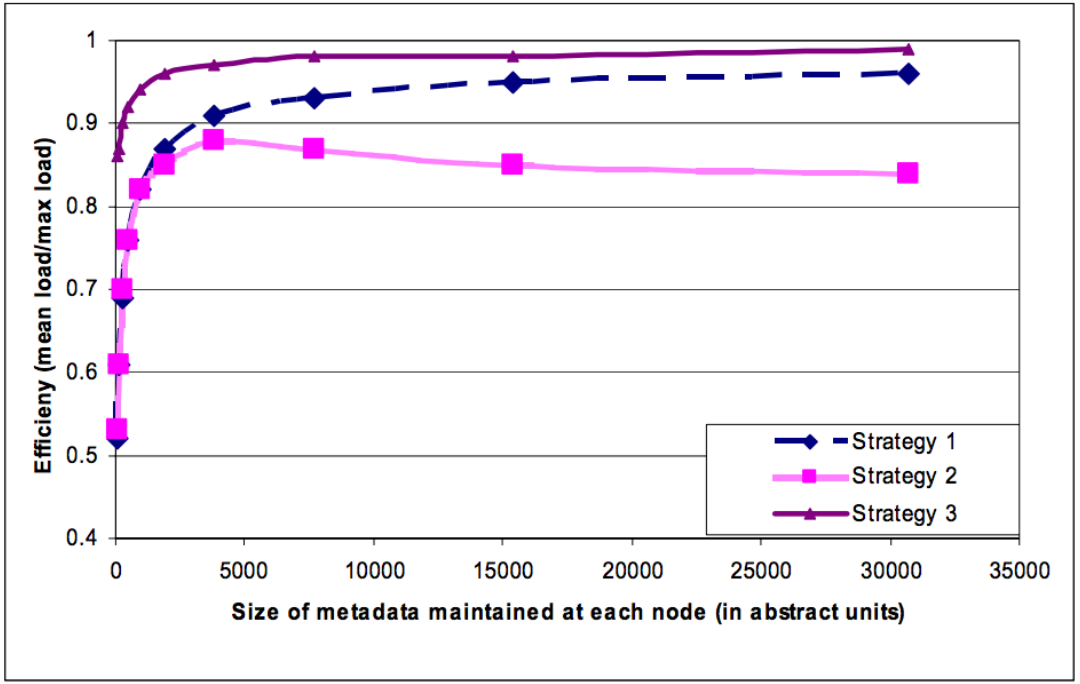


Figure 8: Comparison of the load distribution efficiency of different strategies for system with 30 nodes and N=3 with equal amount of metadata maintained at each node. The values of the system size and number of replicas are based on the typical configuration deployed for majority of our services.

Table 2: Performance of client-driven and server-driven coordination approaches.

	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

Divergent Versions: When and How Many?

Shopping cart service profiled for a period of 24 hours

99.94% of requests saw exactly one version; 0.00057% of requests saw 2 versions; 0.00047% of requests saw 3 versions and 0.00009% of requests saw 4 versions.

Concurrent writers > failures

Usually triggered by busy robots and rarely by humans

Discussion and Conclusions

Demonstrates that decentralized techniques can be combined to provide a single highly-available system

Provide three parameters of (N, R, W) to tune application based on their needs; exposes data consistency and reconciliation logic issues to the developers (flexible)

Problem: Overhead in maintaining the routing table increases with the system size

Solving by introducing hierarchical extensions to Dynamo

Thank you!

Q&A