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 Yogi Joshi.

What do the applications demand?

- High availability among failures of number of components.
- High scalability to facilitate growth.
- High performance.
- Strict control over the tradeoffs between consistency, availability, cost and performance
- Ability to configure such tradeoff as per the need of the applications.

Why not RDBMS ?

- Simple usage pattern : Only primary key access to data store.
- Examples : Shopping carts, Session management, Catalogs, etc.
- No complex querying is needed.
- Higher cost of maintaining a RDBMS.
- Most of the RDBMS systems choose consistency over availability.
- Limited replication options.
- Not easy to scale.

Contributions of the work…

- Demonstration of blending different techniques in a single system to meet the goals.
- Tuning different techniques to meet the diverse needs of different services.
- Successful and extensive usage of eventual consistency.

Assumptions and Requirements

- Query model is simple.
- Weaker consistency is ok.

- SLAs drive the stringent latency requirements, measured at 99.9th percentile of the distribution.
- Configurability of the tradeoffs.
- Only internal usage of Dynamo.

Design Considerations

- Use of eventual consistency for high availability.
- Conflict resolution is done at the time of 'read' operation. Example : Shopping carts.
- Flexible conflict resolution by the data store or the application itself.
- Incremental scalability, Symmetry, Decentralization and Heterogeneity.

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

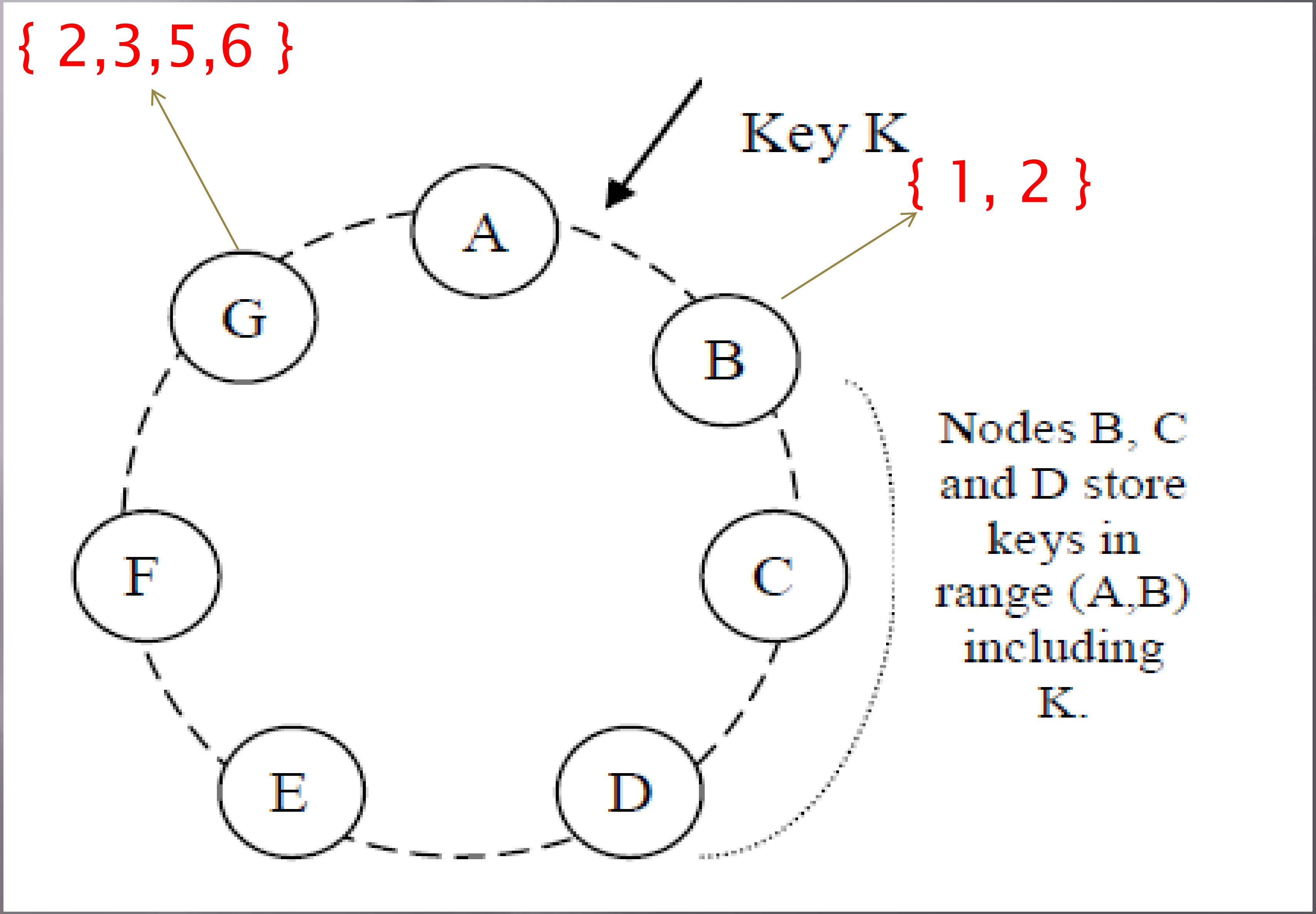
Interface

- Get() and put() operations
- Get(key)
- Put(key, context, object)
- Context – metadata information such as object version
- Key is hashed using MD5 to identify the storage node for the key.

Data partitioning

- Consistent hashing – output range of the function is a circular space.
- Each system node is assigned a position in the circular space.
- Key is hashed to identify its position in the circular space.
- A node is responsible for the keys between its predecessor and itself.
- Virtual nodes in order to account for uniform load distribution and heterogeneity.

Advantages of Virtual Nodes



Replication

- Each key has a coordinator node.
- The coordinator node replicates its keys to N-1 successive nodes on the ring when traversing in clockwise direction.
- A set of nodes, responsible for storing a key, constitute a ‘preference list’ of that key.
- A ‘preference list’ contains N distinct physical nodes.

Object Versioning

- Required due to eventual consistency mechanisms.
- Each modification of an object involves writing a new version. This causes multiple versions.
- Both systemic and application driven reconciliation.
- Vector clock - list of (node, counter).
- Client has to specify the version by passing the ‘context’ of earlier ‘read’.
- Size of vector clock is truncated periodically by keeping only a certain number of tuples. This can cause issues during reconciliation.

Example – Vector clocks

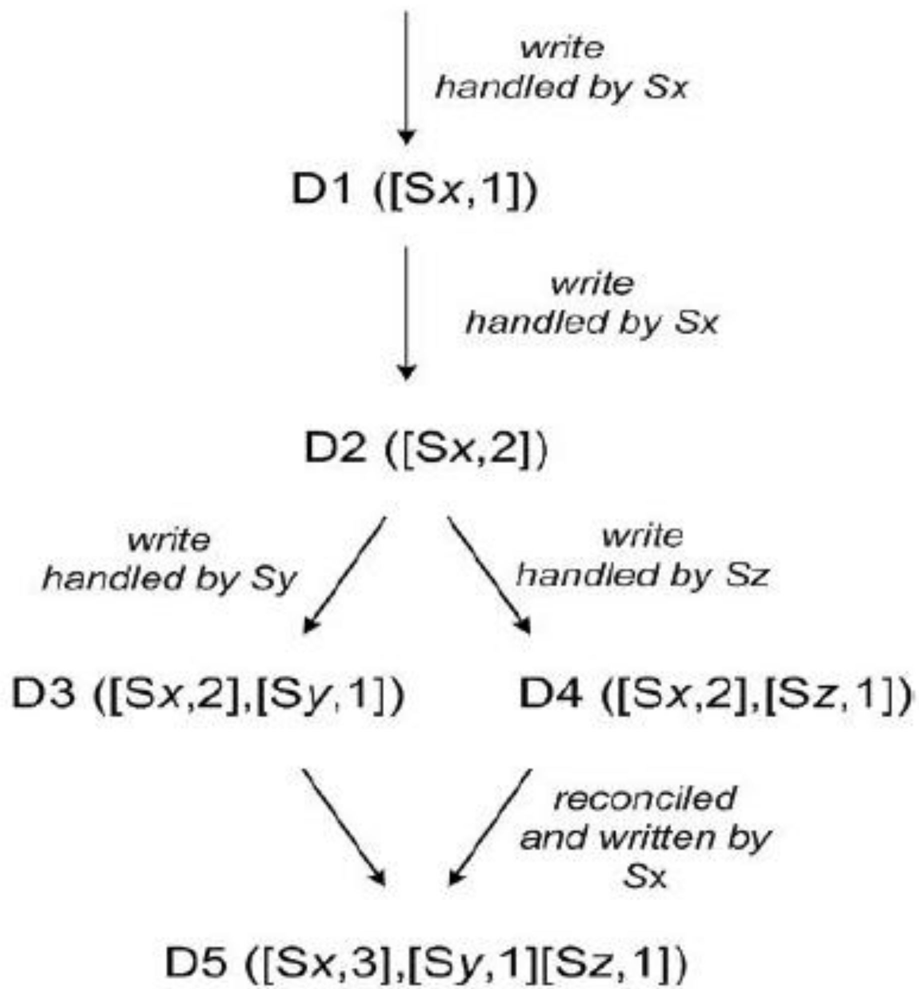


Figure 3: Version evolution of an object over time.

Execution of get() and put()

- Request is routed via load balancer or client is aware of the partitions.
- A coordinator is the first node in the preference list, and it serves the request.
- Consistency protocol like quorum systems.
- R , W i.e. read and write quorum sizes are configurable.
- A set of nodes in the preference list are accessed for the read and write operations.

Hinted handoff

- No strict quorum membership . This helps to tackle failures.
- If a node fails, the replicas supposed to be handled by it, are handed over to a different node in the ring with a ‘hint’.
- Once the failed node recovers, the ‘hint’ helps to relocate the previously moved replicas to that node.
- Replicas are stored across multiple data centers.

Replica synchronization

- Merkle Trees – Leaves are hashes of the values of individual keys, and parents are hashes of their individual children.
- A Merkle tree for each range of keys.
- Comparison involves only a part of the tree to be downloaded. For example: Only the root is downloaded initially.
- If two trees between the nodes are not in 'sync' then they are brought in sync using anti-entropy.

Membership and failures

- Administrator adds/removes nodes in the ring.
- The membership changes are persistently stored by the nodes.
- Gossip based protocol to propagate these changes in the ring.
- Each node contacts its peers randomly to download the ‘membership’ changes.
- This involves propagation of partitioning and placement information.
- Eventual reconciliation of membership information.
- Gossip based protocol subsumes global failure detection.

Implementation

- Choice of different storage engines such as MySQL, BDB, etc.
- Coordinator acts on behalf of the clients.
- A state machine gets created on the node, where a client's request arrives.
- Use of 'read-repairs' to update stale versions with the latest copy.
- The write operations is done on the replica, which responded fastest to the last read operation.

Experiences and lessons

- Business logic specific reconciliation.
- Timestamp based reconciliation.
- High performance read engines.
- Tuning of the read, write quorums sizes and replication factor.

Empirical results for latencies

- Diurnal pattern due to the difference between the request rates between daytime and nighttime.
- 99.9th percentile latencies are much higher than the average latencies.
- So, 'Object buffer' optimization is used, where the data is written to buffer in the replicas, but at least one replica has the data written to the persistent storage.
- The improvement shows lowering 99.9th percentile latency by a factor of 5.

Empirical results – Uniformity of the load distribution

- The number of ‘overloaded’ nodes increase as the number of requests increase.
- This happens because ‘popular’ keys are accessed more frequently when the number of requests grow.
- Further, during low loads, the number of ‘overloaded’ nodes increase as fewer popular keys are accessed, and this causes load imbalance.

Evolution of partitioning schemes

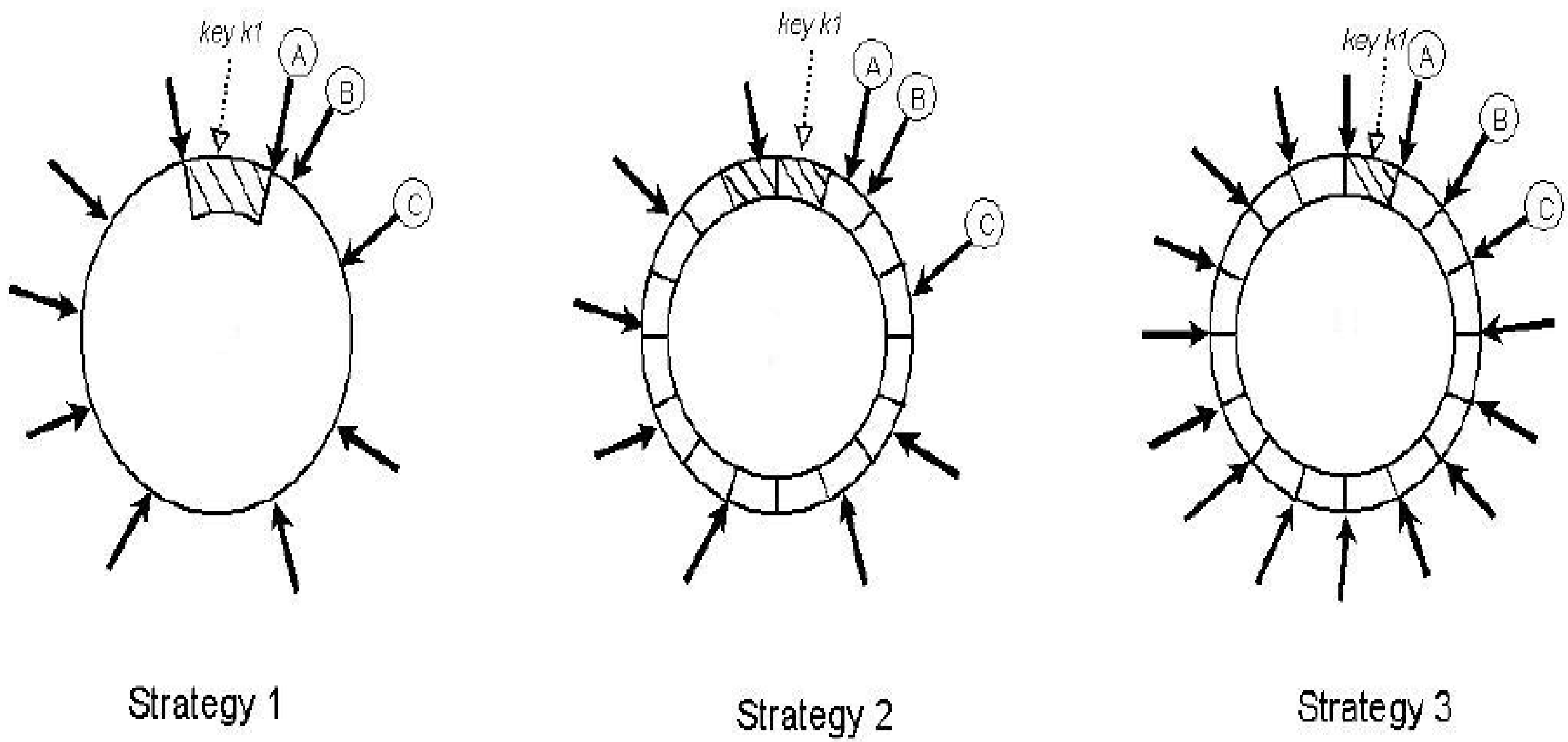


Figure 7: Partitioning and placement of keys in the three strategies. A, B, and C depict the three unique nodes that form the preference list for the key k1 on the consistent hashing ring (N=3). The shaded area indicates the key range for which nodes A, B, and C form the preference list. Dark arrows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

T random tokens per node and partition by token value:

- Tokens ordered by their values in the hash space.
- T randomly chosen tokens are assigned to a node.
- Bootstrapping is inefficient.
- Complicated archival due to random key ranges.
- Recalculating Merkle trees is inefficient as multiple key ranges are changed when a node joins or leaves the system.

T random tokens per node and equal sized partitions

- Hash space divided into Q equally sized partitions
- Each of the S nodes is assigned T random tokens.
- Partition placement is independent of partitioning scheme.
- Placement scheme can be changed at runtime.

Q/S tokens per node, equal-sized partitions

- Hash space divided into Q equally sized partitions
- Each of the S nodes is assigned Q/S tokens.
- Addition and removal of nodes is easy, and involves minimal changes to the membership information.

Comparison of strategies

- Third strategy is the most efficient strategy.
- Bootstrapping is easy as the ranges are fixed, so no need to access a node’s membership information for bootstrapping.
- Same applies to the archival.
- Third strategy requires extra coordination while adding or removing a node in order to preserve the property.

Coordination and background tasks

- Load balancer is used in server driven coordination.
- In client driven approach, the client application polls a node and downloads the membership information from, and it routes the requests accordingly.
- Client driven approach reduces the latency as server need not run the load balancer.
- Background tasks are scheduled after cleared by an admission controller.
- This controller checks latencies, queue wait times to assess resource availability for foreground tasks.

Critique - Strengths

- Ability to 'tune' the attributes such as consistency, availability and latency as per the application need. This enables scalability in terms of different application domains.
- Extensive usage of asynchronous tasks such as read-repairs and efficient replica synchronization, which reduce window of inconsistency in case of partial quorums.
- Emphasis on 99.9th percentile latency along with scalability. This assures that each segment of the consumers is duly taken into account i.e. truly "always-on" experience for almost all the clients.

Critique - Weakness

- No empirical results on the scalability, when the nodes are added or removed. How to estimate the efficiency to coordinate the removal/addition of a node ? More evidence needed about the corresponding latency values as well.
- Empirical results are based on a single strict quorum configuration. Analysis on partial quorums would make comprehensive discussion on the configurable tradeoffs for consistency and latency.
- No details about the execution of admission controller for background tasks. Does its constant execution affect the efficiency of foreground tasks ? Any empirical evidence to support its effectiveness ?
- Not enough information on the consistent hashing function(s) with reference to partitioning.
- Minor clarity issues related to the usage of English.

Extensions

- Usage of Merkle trees for propagating and comparing the membership and range information will enable more scalability in terms of number of nodes in a ring.

- Isolation of the ‘Admission controller’ to separate processing unit to provide efficient monitoring of important system attributes.

Questions