Dynamo: Amazon’s Highly Available Key-value Store

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

Amazon.com

Presented by Chengyu Chen

Table of contents

1. Introduction
2. Background and motivation
3. System design
4. Conclusion and discussion

1. Introduction

- Dynamo is a set of techniques that together can form a highly available key-value distributed data store.
- Fun fact #1: **Dynamo != Amazon DynamoDB**
- DynamoDB (2012) is developed based on some principles of Dynamo (2007).
- Dynamo enables **Leaderless Replication**, whereas DynamoDB favors **Single Leader Replication**.

1. Introduction

- Dynamo keywords: highly available, **eventual consistency**, and highly scalable.
- Eventual consistency: if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value

2. Background and motivation

- Amazon was using commercial databases and pushed them to their limits many years ago
- Led to a huge **Holiday Season Outage** in 2004
- **“A number of outages at the height of the 2004 holiday shopping season can be traced back to scaling commercial technologies beyond their boundaries**.”
  – Amazon CTO Werner Vogels
- Dynamo was born out of the need for a highly reliable, ultra-scalable key/value database

2. Background and motivation

- Amazon is a huge! And it values customers' experience
  → **highly available and scalable** distributed storage system
- Luckily to some services, strong consistency is not a must
  → **eventual consistency**
- Relational database is an overkill and is not friendly to distribution
  → **NoSQL data scheme**

3. System design

Four key principles:

1. **Incremental scalability**: scale out one node at a time with minimal impacts
2. **Symmetry**: no special roles; all nodes are equal
3. **Decentralization**: decentralized peer-to-peer techniques over centralized control
4. **Heterogeneity**: work distribution must be proportional to the capabilities of the individual servers

3. System design

Table 1: Summary of techniques used in *Dynamo* and their advantages.

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.

3. System design

1. Data partitioning

Typically, a distributed DB map the data uniformly across all nodes using hash functions.

But what would happen if we want to scale it out?

It needs re-distribute all the data! Therefore, Dynamo takes a different approach:

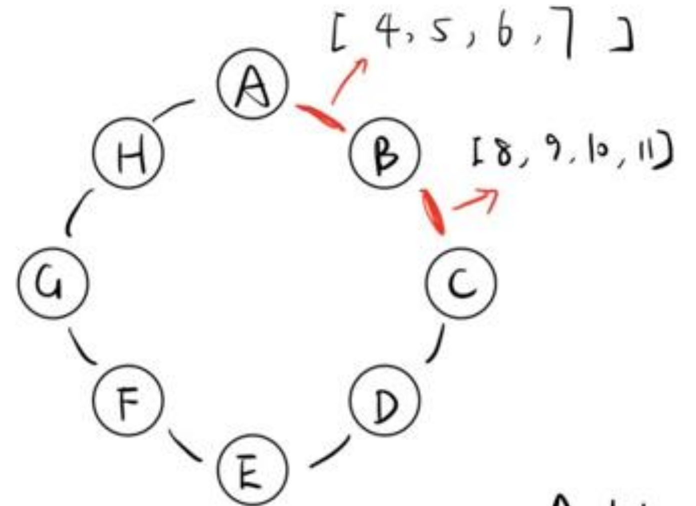
3. System design

1. Data partitioning (that supports incremental scaling)

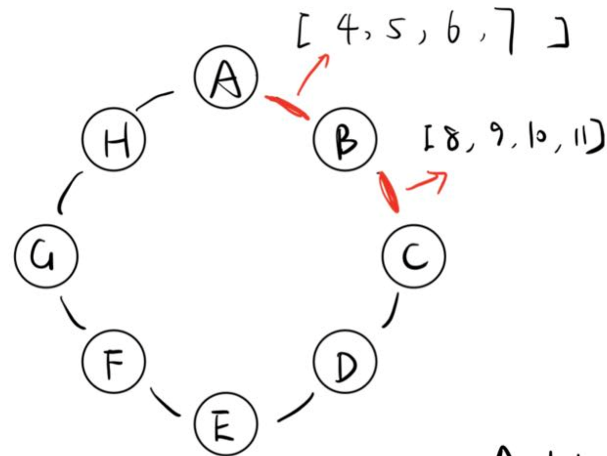
Introduce consistent hashing:

- Each node handles a range of data in the ring
- For instance, node B handles range (A, B]; node A handles range (H, A]

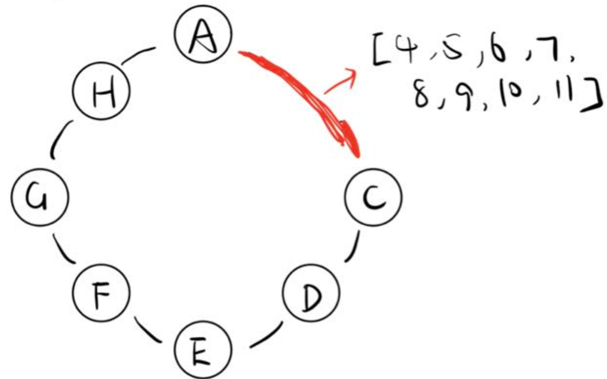
Therefore, when a node is removed or added, only the neighbors are affected.



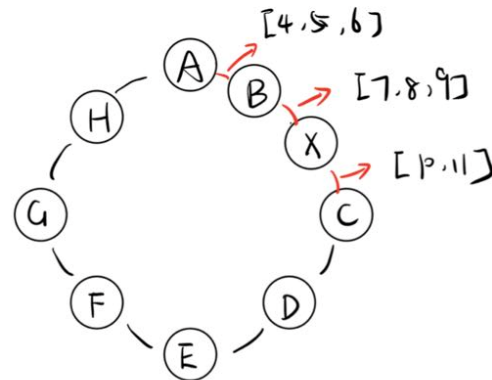
3. System design



Remove node B:



Add node X:



3. System design

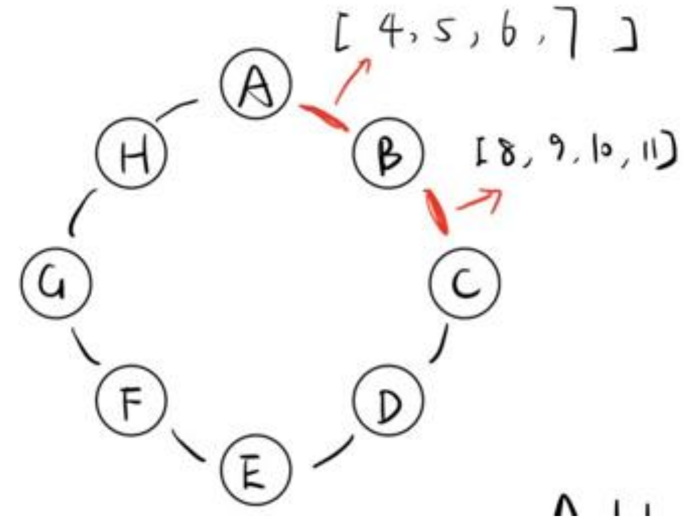
1. Data partitioning (that supports incremental scaling)

But the basic consistent hashing presents some challenges:

1. Nodes may be not distributed uniformly when the number of nodes change
2. Does not account for heterogeneity of node server

Need a smaller unit – “**virtual node**”. Each node may consist of multiple virtual nodes.

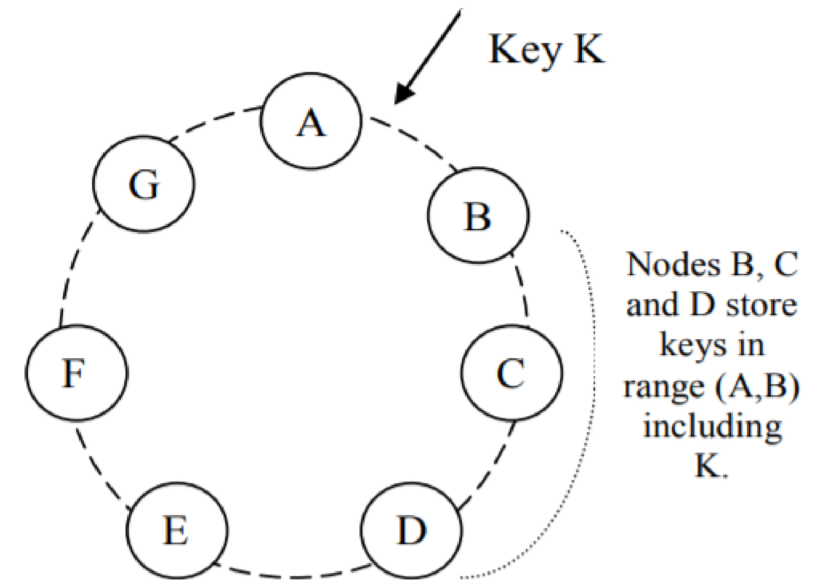
Effectively, each physical node can be mapped to multiple locations on the ring



3. System design

2. Replication

- Dynamo replicates its data in next N-1 successive hosts
- The original node (that this data maps to) serves as a **Coordinator Node** and handles the replications.
- For instance, if N = 3, node B replicates key K at nodes C and D in addition to storing it locally.



3. System design

3. Eventual Consistency

- Allows updates propagate asynchronously in the background
- Highly available writes!
- Might read an old value; might have two different writes on the same object!
- Solution: **Data Versioning**

3. System design

3. Eventual Consistency and Data Versioning

- A key-value object can have different versions that forms a version history
- Ignore when reading an old version
- Use a **Vector Clock** to resolve conflicting versions

3. System design

3. Eventual Consistency and Data Versioning

- Use a **Vector Clock** to resolve conflicting versions
- Example on the right.

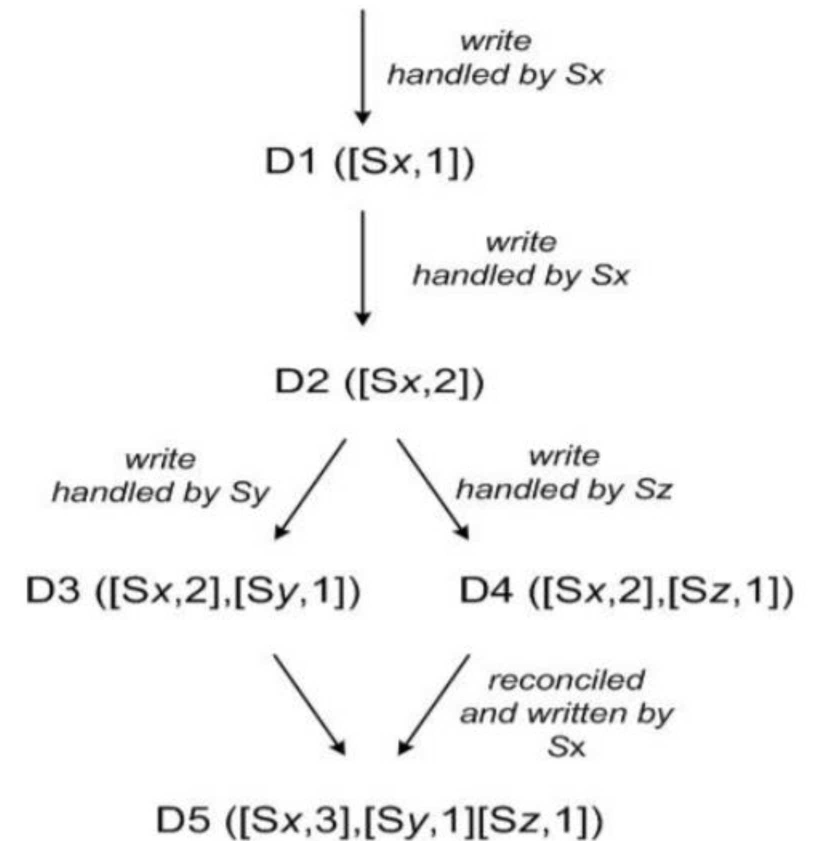


Figure 3: Version evolution of an object over time.

3. System design

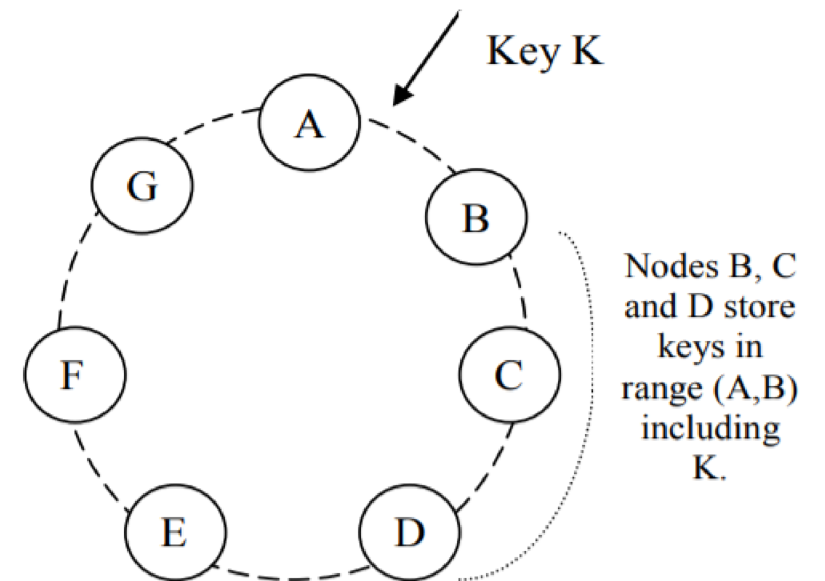
3. Ensuring consistency

- A quorum like system: R is the minimum number of nodes that must participate in a successful read operation; likewise for W in a successful write operation.
- As long as $R + W > N$, data consistency is ensured.
- By changing values of R and W, we can prioritize read or write operations (availability vs durability)
- But what if some nodes in N fail?

3. System design

4. Handling failures: hinted handoff

- But what if a few nodes in N fail? Use a "Sloppy quorum"!
- All read and write operations are performed on the first **N healthy nodes from the preference list**
- Example: when node A dies, a replica that would normally have lived on X will now be sent to D! Then access these N healthy nodes.
- Improves availability



3. System design

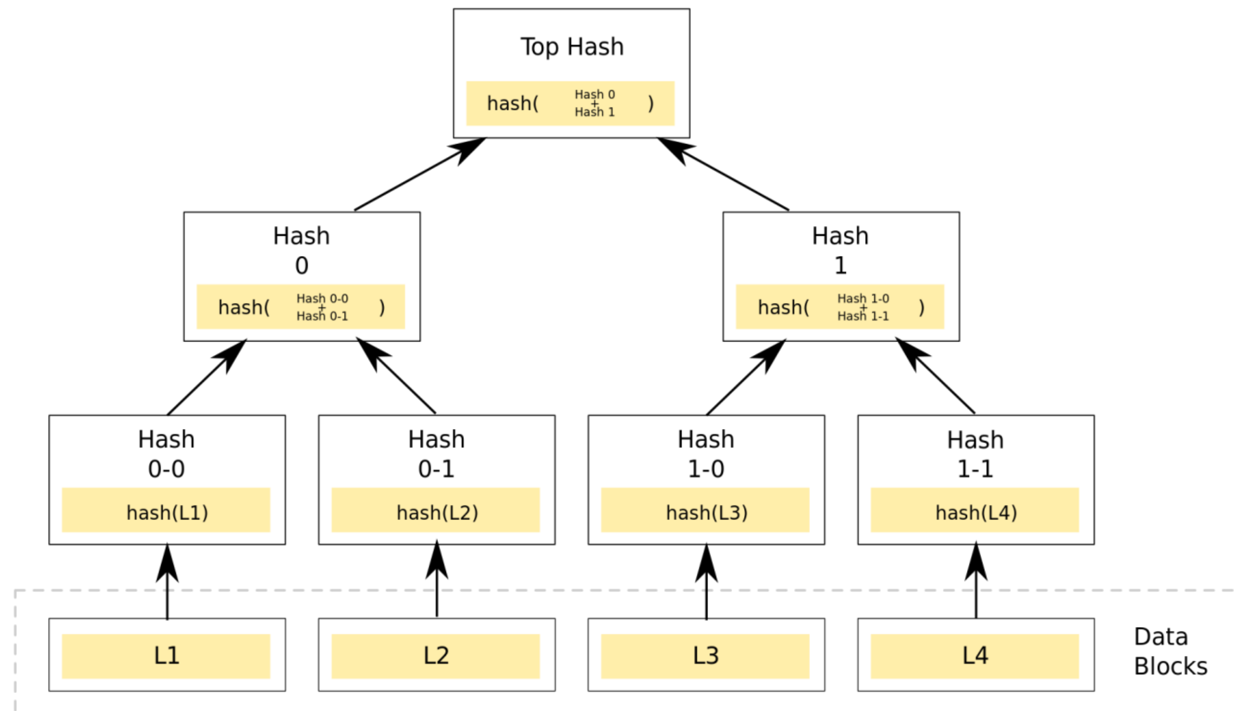
5. Handling permanent failures: replica synchronization

- If there are too many failures or node failures are not transient enough: need replica synchronization
- Challenge: how to detect inconsistency efficiently?
- Dynamo uses **Merkle trees (Hash trees)**

3. System design

5. Handling permanent failures: replica synchronization

- **Merkle trees: A hash tree where leaves are hashes of the values of individual keys and where parents are hashes of their children**



3. System design

5. Handling permanent failures: replica synchronization

- **Merkle trees: A hash tree where leaves are hashes of the values of individual keys and where parents are hashes of their children.**
- Therefore, only need to compare Merkle trees of replicas from top to bottom. If it's consistent from the start, only a comparison of roots is performed.

4. Conclusion and discussion!

- Incrementable scaling
- Allow services to decide tradeoff between performance, availability and durability by changing N, R and W.
- Note that dynamo exposes data consistency and reconciliation logic issues to developers
- Only works well with up to hundreds of nodes

UNIVERSITY OF MICHIGAN

Questions?