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, 2007

Group 7

Quyen Tran, Pujith Veeravelli, Sean Jung, Saatvik Agrawal, Sarvesh Kasture

Introduction and Motivation

Motivation

Scalability and Performance

- Traditional relational databases face challenges to meet Amazon's fast-growing demand

Reliability and availability

- Infrastructure failing is not an exception, but rather a norm in large-scale distributed systems.
- Any downtime or performance degradation has big financial impacts

Introduction

Dynamo: Amazon's internal technology

- Distributed data storage
- No SQL
- Highly available
- Highly scalable

In 2012, Amazon combined the best parts of **Dynamo** (scalability and predictable high performance) with the best parts of **SimpleDB** (ease of administration, consistency , and a table-based data model) to introduce **DynamoDB** as AWS database system.

Amazon Prime Day 2021, the peak load to the database reached 89 million requests per second *

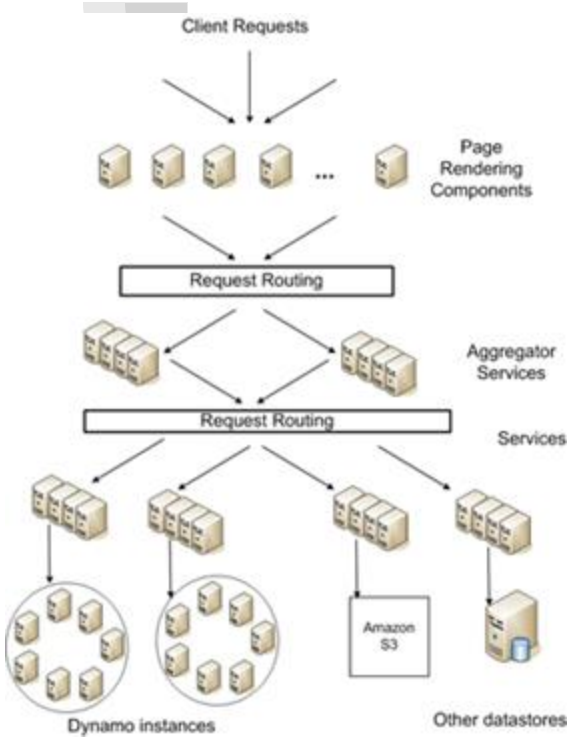
Elhemali, Mostafa, et al. "Amazon DynamoDB: A Scalable, Predictably Performant, and Fully Managed NoSQL Database Service." 2022 USENIX Annual Technical Conference, USENIX Association, 11-13 July 2022, Carlsbad, CA, USA, www.usenix.org/system/files/atc22-elhemali.pdf.

Background

System Assumptions and Requirements

- **Query Model**
  - Assumption: Many Amazon's services does not require complex data relationships
  - No need for relational schema like SQL databases
  - Does not support operations spanning multiple data items
  - Each operation (read or write) only interacts with a **single data item** identified by a unique key.
  - Each **unique keys** can retrieve data stored as binary object (BLOB). BLOBs are flexity as it can hold many types of data
- **ACID Properties (Atomicity, Consistency, Isolation, Durability)**
  - ACID are properties that guarantee database reliability
  - Assumption: Strong ACID guarantees often impact availability
  - Relax **Consistency** if it yields better availability
  - No guarantee in **Isolation** so it permits only single-value update
- **Efficiency**
  - System needs to run on standard, low-cost hardware
  - Must delivery fast and reliable data access at level 99.9th percentile
- **Other Assumptions**
  - Assumption: Dynamo is exclusively used within Amazon
  - Security measures like authentication and authorization are unnecessary

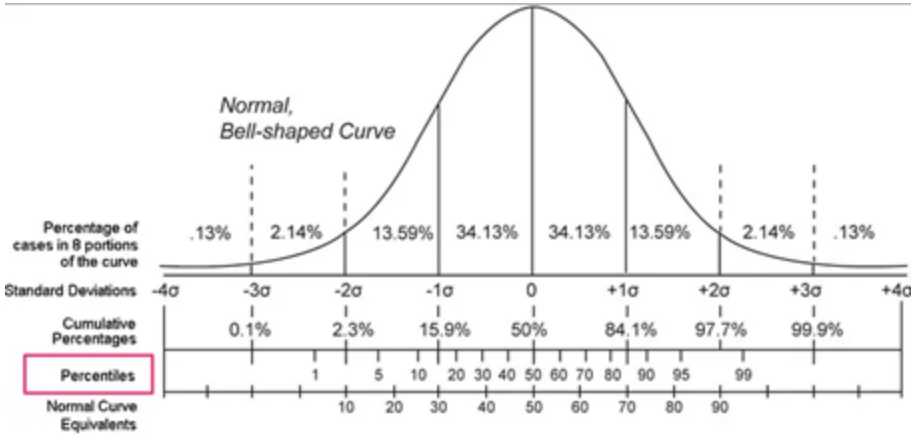
Service Level Agreements (SLA)



SLA: A **performance contract** between the client and service

High requirement for response time: **99.9th percentile** of the distribution

- Focus on customer experience
- Cost-benefit analysis



Quyen Tran

https://blog.exploratory.io/how-to-calculate-percentile-for-each-group-in-r-and-exploratory-4e6021f7fe10

Design Considerations

Conflict resolution: Data replication → Data conflict

- When to resolve update conflict → during reads or writes
- Who to resolve update conflict → data store or application

Incremental scalability: ability to scale out one storage host (called node) at a time

Symmetry: each node has the same set of responsibilities as its peers

Decentralization: distributed control

Heterogeneity: work distribution, must be proportional to the capabilities of the individual servers

Related Work

Peer-to-peer Systems

P2P consists of a fully decentralized network where nodes directly communicate.

- Unstructured - Search query floods through system to find peers which share data.
- Structured - Uses a globally consistent protocol for efficient routing of search queries.
- Pastry/Chord - Routing mechanisms to ensure queries will be answered within a bounded number of hops
- Oceanstore - Global, transactional, persistent storage service supporting updates on widely replicated data with conflict resolution using total order.
- PAST - Abstraction of Pastry for persistent and immutable object, the application must handle storage semantics such as mutable files.

Distributed File Systems and Databases

DFS has many machines, but generally with some level of coordination.

- Unlike P2P, hierarchical files are supported.
- Update conflicts require resolution procedures.
- Consistency is guaranteed even in the case of disconnected operations and issues such as network partitions.
- Distributed block storage systems split large objects into highly-available blocks.

Discussion

Unlike the mentioned decentralized storage systems, Dynamo is:

- Targeted at applications needing an “always writable” data store where no updates are rejected.
- Intended for an environment where all nodes are trusted.
- Not expected to support for hierarchical namespaces or complex relational schemas.
- Built for extremely latency sensitive applications

System Architecture

Architecture

- Architecture is complex because it is within a production setting
- Requirements:
  - Data persistence
  - Scalable and robust solutions for many problems
    - Load balancing, membership and failure detection, failure recovery, etc.
- Paper focuses on the core distributed techniques utilized by Dynamo:
  - Partitioning
  - Replication
  - Versioning
  - Membership
  - Failure handling and scaling

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.

Table 1 presents a summary of the list of techniques Dynamo uses and their respective advantages.

System Interface

- Dynamo stores objects associated with a key
- The interface provides two operations:
  - get(key)
    - Finds object replicas associated with the key
    - Returns either
      - Single object
      - or list of objects from multiple versions along with their context
  - put(key, context, object)
    - Determines where object replicas should be placed based on the key
    - Writes data

System Interface

- Context:
  - Contains system metadata about the object
    - Versioning information
  - Not directly accessed or analyzed by the caller
  - Helps validate objects
  - Handles conflicts between objects returned by the put operation
- Hashing:
  - Dynamo uses a MD5 hash for the object keys to determine which storage node to use
    - MD5 hash generates a 128-bit hash value

Partitioning Algorithm

- Dynamo must scale incrementally
  - Data must be dynamically distributed amongst storage hosts (nodes)
- Dynamo uses consistent hashing
  - Output hash space is represented as a ring
    - Largest possible value wraps around to the smallest possible value
  - Each node in the system is assigned a random hash value, this value represents the node's position on the ring
  - Objects are hashed based on their key value, and then the ring is traversed clockwise to find the first largest node
- Advantage of consistent hashing is that removal or insertion of a node affects only the direct neighbors, allowing incremental scaling

Partitioning Algorithm

- However, consistent hashing has some issues
  - Random position of the nodes can lead to non-uniform load distribution
  - Consistent hashing does not account for the different performance capabilities of the nodes
- Thus, Dynamo uses a consistent hashing variant
  - Each node is assigned to multiple points in the ring
  - These are considered “virtual nodes”
- Virtual nodes have several advantages:
  - If a node fails, its load is distributed amongst the other nodes
  - A new node receives an even/equivalent amount of load from the other existing nodes
  - The number of virtual nodes assigned can vary depending on the performance capability of the node

Replication

- Dynamo replicates its data across multiple nodes
  - Ensures high availability and durability
- Each item is stored on N nodes, where N is a configurable parameter
- Each key is assigned to a coordinator node
  - Node is responsible for replicating its data items
  - Item is stored locally, and replicated in N-1 clockwise successor nodes
- Essentially, each node becomes responsible for the region of the ring between it and the node N places before it
  - Just an extension of the consistent hashing method

Replication

- Each key k has a corresponding list of nodes that is responsible for storing k
  - This is called the **preference list**
- Every node in the system can determine which nodes should be on this list
  - I.e., every node understands where each key should be stored
- Preference list contains more than N nodes to account for node failure
  - If one of the first N nodes is down, the next node from the preference list is used to store the item
- To account for virtual nodes, the preference list skips positions in the ring that correspond to physical nodes that have already been utilized
  - For example, if an item should be replicated onto virtual nodes A, B, and C, but A and C map to the same physical node, the item will instead be replicated onto virtual nodes A, C and D

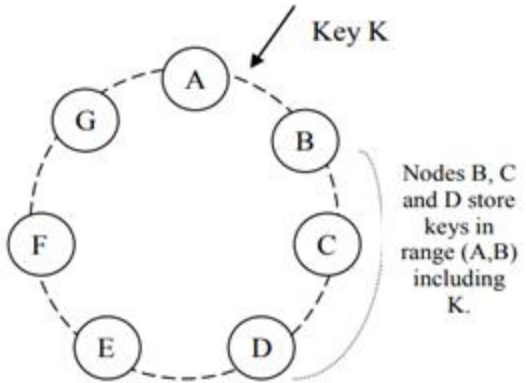


Figure 2: Partitioning and replication of keys in Dynamo ring.

Data Versioning

- Dynamo prioritizes availability over consistency
- Updates can happen during: network partitions, server outages, concurrent client modifications
- Result: Multiple versions of same object may exist
- How can we track and reconcile versions?

Vector Clocks

- Track causality between different versions
- Structure: List of (node, counter) pairs
  - Example: [(Sx, 1), (Sy, 2)]
    - Node Sx: counter value 1
    - Node Sy: counter value 2
- Each version of object has associated vector clock
- Used to determine relationship between versions:
  - Causally related (one is newer)
  - Concurrent modifications (conflict)

Version Evolutions Example

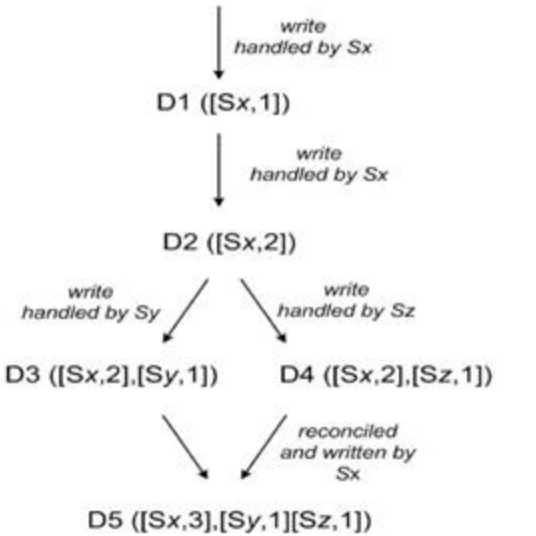


Figure 3: Version evolution of an object over time.

1. Initial Write (D1):
   - Clock: [(Sx, 1)]
   - First write handled by node Sx
2. Second Write (D2):
   - Clock: [(Sx, 2)]
   - Same node handles update
   - D2 descends from D1 (can overwrite D1)
3. Third Write (D3):
   - Clock: [(Sx, 2), (Sy, 1)]
   - New node Sy handles this update
   - Builds on D2's version
4. Concurrent Write (D4):
   - Clock: [(Sx, 2), (Sz, 1)]
   - Different client updates D2
   - Handled by node Sz
5. Reconciliation Write (D5):
   - Clock: [(Sx, 2), (Sy, 1), (Sz, 1)]
   - Client reconciles the versions and writes back new version D5

Put() and Get() Requests

- Any node in Dynamo can receive a client request
- Key parameters:
  - N: Number of replicas
  - R: Minimum nodes for successful read
  - W: Minimum nodes for successful write

Put Operation Flow

1. Client sends put(key, value) to any Dynamo node
2. That node becomes the coordinator for this request
3. Coordinator:
    - Generates new vector clock for this version
    - Saves locally
    - Sends to N-1 other nodes (based on preference list)
4. Waits for W-1 responses
5. If it gets W-1 responses, tells client "success"

Get Operation Flow

1. Client sends get(key) to any Dynamo node
2. That node becomes coordinator
3. Coordinator:
    a. Requests key from N highest-ranked reachable nodes
    b. Waits for R responses
4. If there are multiple versions (conflicts):
    a. Returns all conflicting versions to client
    b. Client must reconcile
5. If versions can be reconciled syntactically (one is clearly newer):
    a. Returns the newest version

Handling Permanent Failures

- Nodes can fail permanently or be down so long that hinted handoff hasn't worked
- Result: Replicas become inconsistent across nodes
- How can we detect and fix these inconsistencies?
- Can't simply compare all keys across nodes (too expensive)
- Can't transfer all data between nodes (too much bandwidth)
- Need a way to quickly identify which keys are different

Merkle Trees

- Smart fingerprinting system for data
- Each node maintains a Merkle tree for its key ranges
- Properties:
  - Can compare large datasets by just comparing “fingerprints”
  - If fingerprints differ, can drill down to find exactly which keys are different

How Dynamo Manages Nodes

- Gossip protocol for membership sharing
- Explicit node join/leave (no auto-join)
- Data automatically redistributed when nodes added/removed

Implementation

Implementation

- Each storage node has request coordination, membership/failure detection, and a local persistence engine.
- Different storage engines may be plugged in, allowing for customization.
- Request coordinator executes read/write requests by collecting/storing data at one or more nodes.
- Every request results in a state machine to handle logic, failure handling, and retries.
- Read repair fixes any nodes which return stale data on a read.
- Writes are coordinated by a node in the top N of the preference list. Usually, the node which replied fastest to the preceding read is selected.

Results

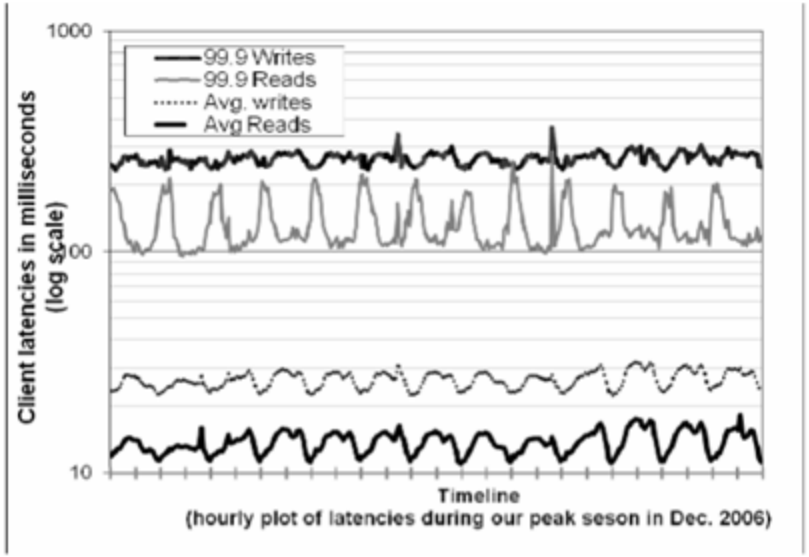


Figure 4: Average and 99.9 percentiles of latencies for read and write requests during our peak request season of December 2006. The intervals between consecutive ticks in the x-axis correspond to 12 hours. Latencies follow a diurnal pattern similar to the request rate and 99.9 percentile latencies are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

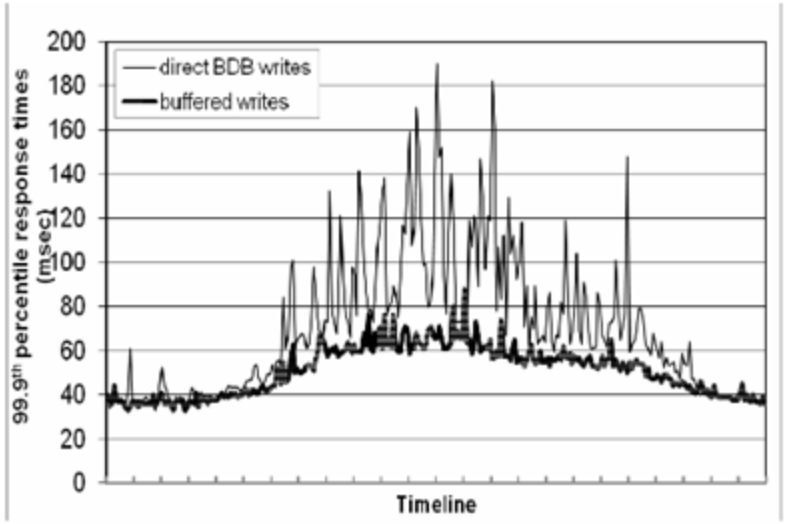


Figure 5: Comparison of performance of 99.9th percentile latencies for buffered vs. non-buffered writes over a period of 24 hours. The intervals between consecutive ticks in the x-axis correspond to one hour.

Experiences and Lessons Learned

N,R and W values

- N: Number of Replicas
  - **Higher N:** Increases durability, decreases performance.
  - **Lower N:** Improves performance, reduces durability.
- R: Read Quorum
  - **Higher R:** Increases consistency, higher latency.
  - **Lower R:** Reduces consistency, improves availability/latency.
- W: Write Quorum
  - **Higher W:** Increases consistency and durability, decreases availability.
  - **Lower W:** Increases availability, decreases consistency/durability.

Practical Use Cases

- **Business Logic-Specific Reconciliation**: Services like shopping carts, where custom reconciliation logic (merging versions) is used.
- **Timestamp-based Reconciliation**: Services requiring simple “last write wins” reconciliation, e.g., session management.
- **High-Performance Read Engine**: Services with minimal updates, tuned for high read throughput with low latency.

Uniform Load Distribution

- Consistent Hashing
  - **Data Partitioning**: Uses a hashing ring to assign data to nodes based on hash values.
- Virtual Nodes
  - **Multiple Tokens per Node**: Each node is assigned multiple points (virtual nodes) on the ring.
- Dynamic Load Adjustment
  - **Rebalancing:** As nodes join or leave, key ranges are dynamically redistributed to maintain uniform load.
  - **Proportional Assignment:** Nodes with more resources handle a larger portion of the load.
- Load Distribution Strategies
  - **Random Tokens**: Initially random assignment of tokens across nodes.
  - **Equal-Sized Partitions**: Divides the hash space into equal partitions, improving load balance.

Divergent Versions and Client-Side vs. Server-Side coordination

- **Divergent Versions:** Low rates of multiplicity with 99.94% having one version
- **Client-Driven:** Client handles routing, more flexible but increases complexity.
- **Server-Driven:** Server manages routing, simplifies client logic but adds load to the server.

Balancing Background and Foreground Tasks

- **Foreground Tasks**: High-priority tasks that interact directly with the user and require quick completion.
- **Background Tasks**: Lower-priority tasks that don't require immediate user feedback, can run asynchronously or during idle periods.
- **Task Prioritization**: Foreground tasks should be prioritized to ensure responsive UI and optimal user experience, while background tasks can be deferred or run in parallel without affecting user-facing processes.
- **Resource Management**: System resources like CPU and memory should be allocated dynamically, ensuring foreground tasks are given the necessary resources without background tasks causing delays or affecting performance.

Conclusion

Conclusion

- Dynamo is a highly available and scalable data store that stores the state of Amazon's core services.
- Desired levels of availability and performance have been achieved while handling failures.
- Tuning of N, R, and W enables customization.
- Production use has demonstrated the success of combining decentralized techniques and shows that an eventual-consistent storage system can be a building block for highly-available applications.

Study questions

1) Given two versions of data with vector clocks [(Sx, 1), (Sy, 3)] and [(Sx, 2), (Sz, 1)], explain step by step how Dynamo determines if these versions are in conflict. Then explain what would happen in a GET request that receives these two versions as responses.
2) Consider a Dynamo system configured with N=3, R=2, W=2. If an application needs to handle a sudden 10x increase in write traffic while maintaining availability, what configuration changes might you recommend? Explain how your changes would affect consistency, durability, and latency.