Data Models
Key/Value: Amazon Dynamo

Part 1

Thanks to Dave Maier, M. Grossniklaus, & K. Tufte

Motivation

- Lessons learned at Amazon
  - lack of reliability and scalability has significant financial consequences
  - reliability and scalability depend on how application state is managed
  - key/value data model is sufficient for many applications: bestseller lists, shopping carts, customer preferences, session management, etc.
- RDBMS are not an ideal solution
  - most features are not used
  - scales up, not out
  - availability limitations due to transactional processing
- Consistency vs. availability
  - high availability is very important
  - user-perceived consistency is very important
  - trade off strong consistency in favor of higher availability

System Assumptions and Requirements

- Query Model
  - simple read and write operations only
  - small objects (BLOB) are uniquely identified by their key
  - no operations span multiple data items
- ACID Properties
  - trade off weaker consistency for higher availability
  - no isolation guarantees and single key updates only
- Efficiency
  - latency requirements measured at the 99.9th percentile of distribution
  - configurable to consistently achieve required latency and throughput
  - trade off performance, cost efficiency, availability, and durability

System Assumptions and Requirements

- Scalability
  - each service (application) uses a distinct Dynamo instance

    What does this say about consistency across instances?
  - requires scale up thousands of nodes in modern implementations
- Security
  - Originally assumed a non-hostile operational environment, but AWS has since implemented robust security features.
  - Authentication and authorization are now standard via IAM roles and policies.

Amazon’s Platform Architecture

- Decentralized, loosely-coupled, service-oriented architecture
  - **page rendering components** generate dynamic web content and query many other services
  - (stateless) **aggregator services** use other services to produce composite response
  - **stateful services** own and manage their own state using different data stores, which are only accessible within its service boundaries
- Availability is paramount
- System scales dynamically to meet demand.

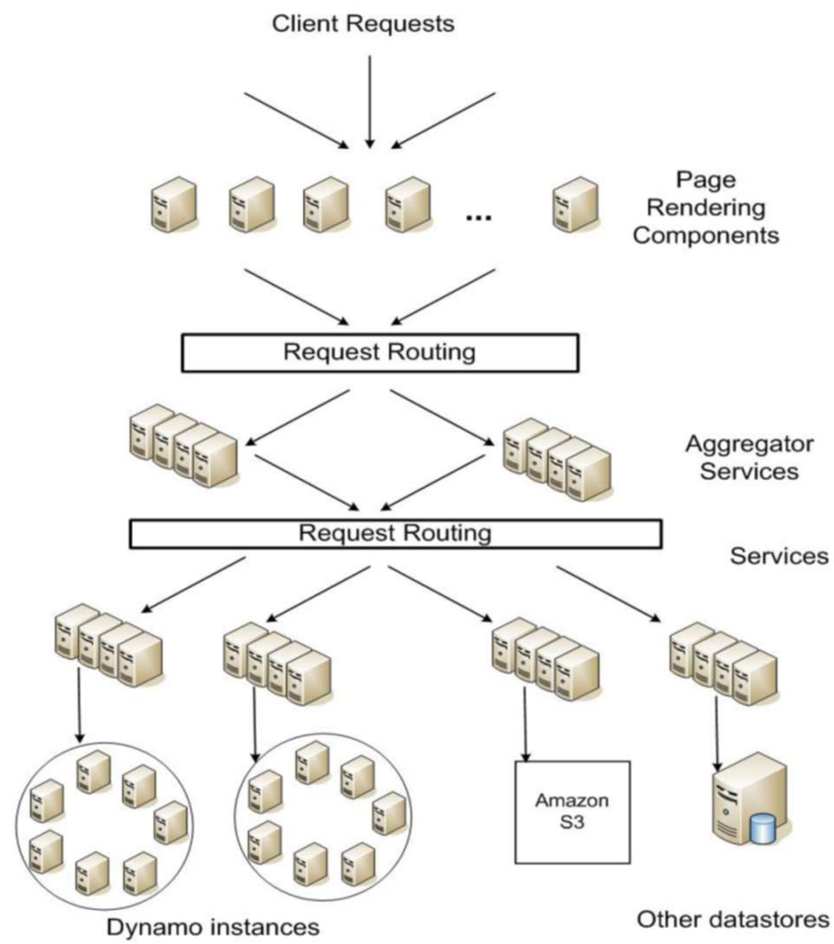


Figure Credit: G. DeCandia et al.: Dynamo: Amazon’s Highly Available Key-Value Store, Proc. SOSP, 2007.

Participation
Q1ipation
Question 1

- Imagine you are designing a **real-time leaderboard for an online gaming platform**. The system needs to quickly update and retrieve player scores with minimal delay.
- Discuss why a **key-value store like DynamoDB** is a better choice than a traditional relational database for this use case.
  - What advantages does DynamoDB offer in terms of **latency, scalability, and availability**?
  - Are there any trade-offs in using a key-value store for this type of application?

Service Level Agreements

- Formal contract between service and client.
- Agreement on system-related characteristics:
  - Example: "This service guarantees to provide a response within 300ms for 99.9% of requests under peak load of 500 requests per second.“
- SLAs in Amazon’s service-oriented infrastructure:
  - Up to 300+ services may be contacted to process an e-commerce request.
  - Services depend on other services, forming a multi-level call graph.
  - Tight SLA contracts ensure overall system performance.

Design Considerations

- Replication for high availability and durability
  - replication technique: synchronous or asynchronous?
  - conflict resolution: when and who?
- Dynamo’s goal is to be “always writable”
  - rejecting writes may result in poor Amazon customer experience
  - data store needs to be highly available for writes, e.g., accept writes during failures and allow write conversations without prior context
- Design choices
  - optimistic (asynchronous) replication for non-blocking writes
  - conflict resolution on read operation for high write throughput
  - conflict resolution by client (application) for user-perceived consistency

Key Design Principles

- Incremental Scalability
  - scale out (and in) one node at a time
  - minimal impact on both operators of the systems and the system itself
- Symmetry
  - every node should have the same set of responsibilities
  - no distinguished nodes or nodes that take on a special role
  - symmetry simplifies system provisioning and maintenance
- Decentralization
  - favor decentralized peer-to-peer techniques
  - achieve a simpler, more scalable, and more available system
- Heterogeneity
  - Load distribution proportional to server capabilities.
  - Supports heterogeneous infrastructure with mixed capacities.

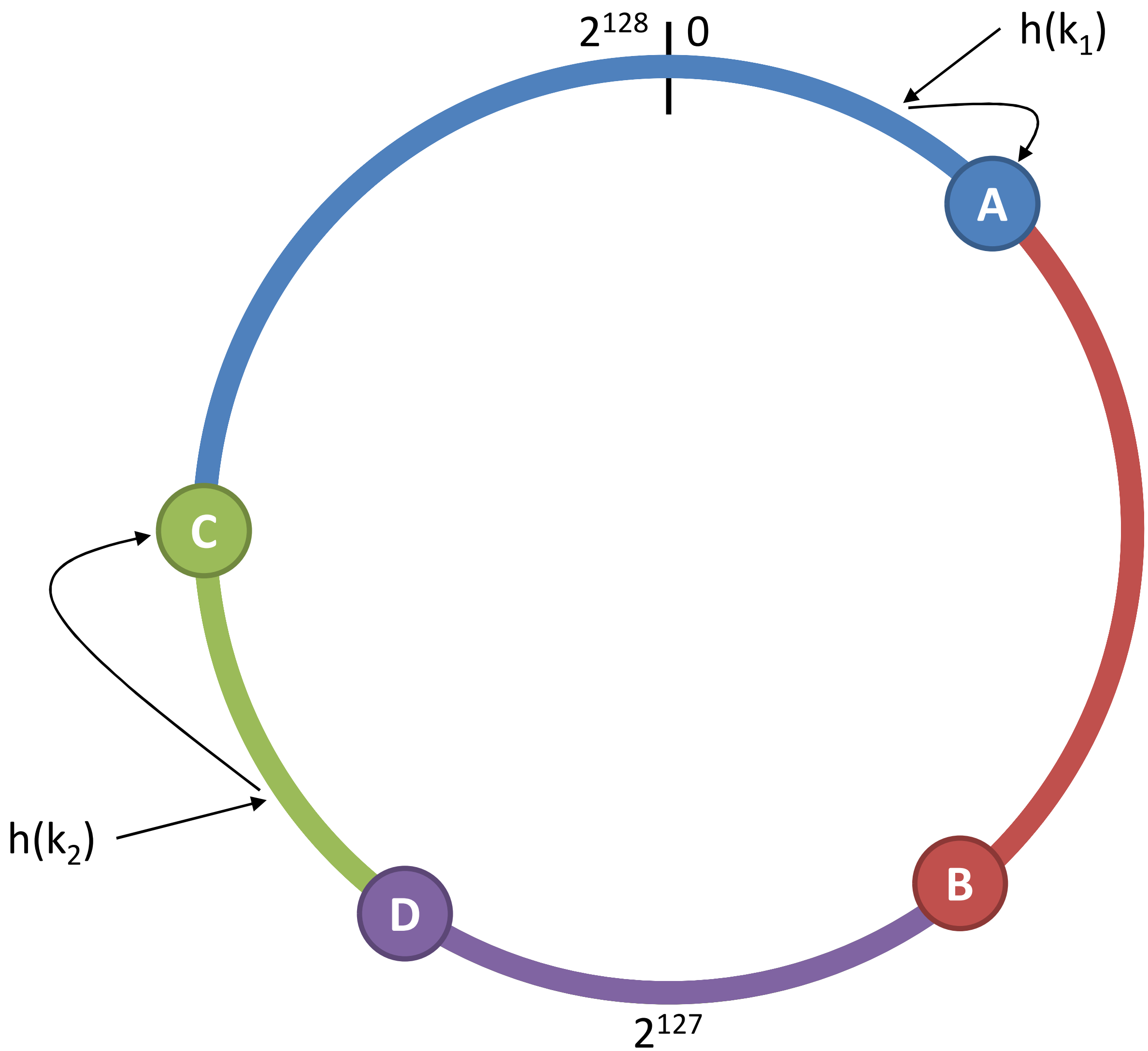
System Interface

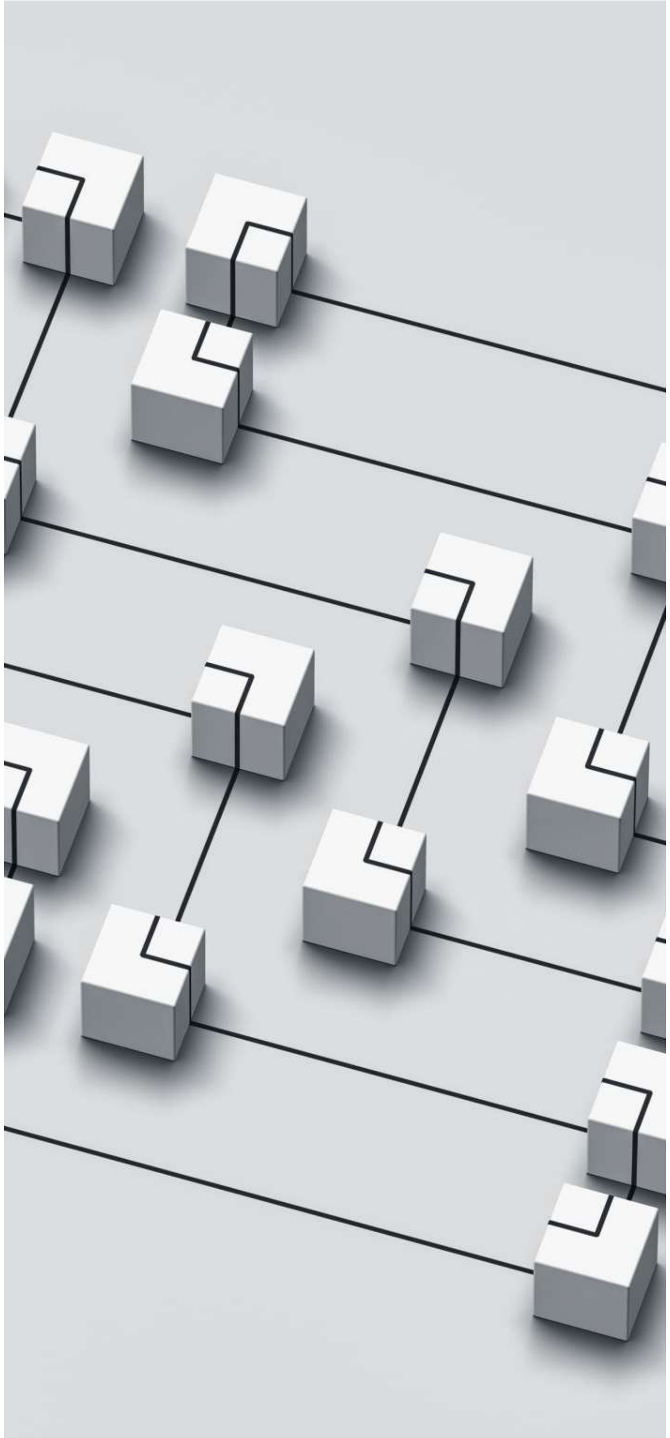
- Dynamo stores objects identified by a key *k* through a simple interface that exposes two operations
- Get operation
  - get(*k*) → [object(s), context]
  - locates object replicas associated with the key *k* in the storage system
  - returns a single object or a list of objects with conflicting versions along with a context
- Put operation
  - put(*k*, context, object)
  - determines where the replicas of the object should be placed based on its key *k* and writes the replicas to disk
- Context
  - encodes system metadata about the object that is opaque to the caller
  - includes information such as the version of the object (vector clock)

Partitioning

- Partitioning based on **consistent hashing**
  - output range of hash function treated as a fixed circular space or **ring**, i.e., the largest hash value wraps around to the smallest hash value
  - each node in the system is assigned a random value within this space, which represents its **position** on the ring
  - each data item identified by a key *k* is assigned to a node by
    1. hashing the data item's key to find its position on the ring
    2. "walking" the ring clockwise to the first node with a position larger than the data item's position
- Each node is responsible for the region in the ring between itself and its predecessor node on the ring
- Arrival and departure of nodes only affects its immediate neighbors and other nodes remain unaffected

Partitioning





Participation Question 2

- Your team is designing a **distributed URL shortening service** (like Bit.ly) that maps long URLs to short unique keys and retrieves them efficiently.
- Discuss how **consistent hashing** can be used to distribute shortened URLs across multiple storage nodes in DynamoDB.
- How does this method ensure **even load distribution and scalability** as the number of URLs grows?
- What challenges might arise in terms of **hotspots** or node failures, and how does DynamoDB mitigate these issues?

Load Balancing

- Problems with basic partitioning algorithm
  - assigning nodes a random position on the ring can lead to non-uniform data and load distribution
  - algorithm does not consider heterogeneity in the performance of nodes
- Virtual nodes
  - each **physical node** gets assigned multiple positions (tokens) in the ring
  - a **virtual node** behaves like a single node in the system
- Advantages
  - if a node becomes **unavailable**, its load can be evenly distributed onto the remaining available nodes
  - if a node becomes **available**, it gets a roughly equivalent share of the load from each of the other available nodes
  - number of virtual nodes that a physical node is responsible for can be decided based on its capacity, accounting for heterogeneity in the physical infrastructure

Load Balancing