DS256:Jan18 (3:1)

CAP Theorem, BASE & DynamoDB

Yogesh Simmhan

Dynamo: Amazon's highly available key-value store

DeCandia G, Hastorun D, Jampani M, Kakulapati G, Lakshman A, Pilchin A, Sivasubramanian S, Vosshall P, Vogels W. *ACM SIGOPS symposium on Operating systems principles (SOSP)*, 2007

Distributed Hashtable

- Primary-key only access for read and write
  - Value is blob, <1MB
  - Shopping cart, best sellers list, user preferences
- High availability when failures are a given
  - disks, network, data center
  - Available across data centers
- E.g. Shopping cart serves 10M requests, 3M checkouts per day (2007)
- RDBMS: Cost hardware, skilled DBA, consistency over availability due to replication limits, limited load balancing

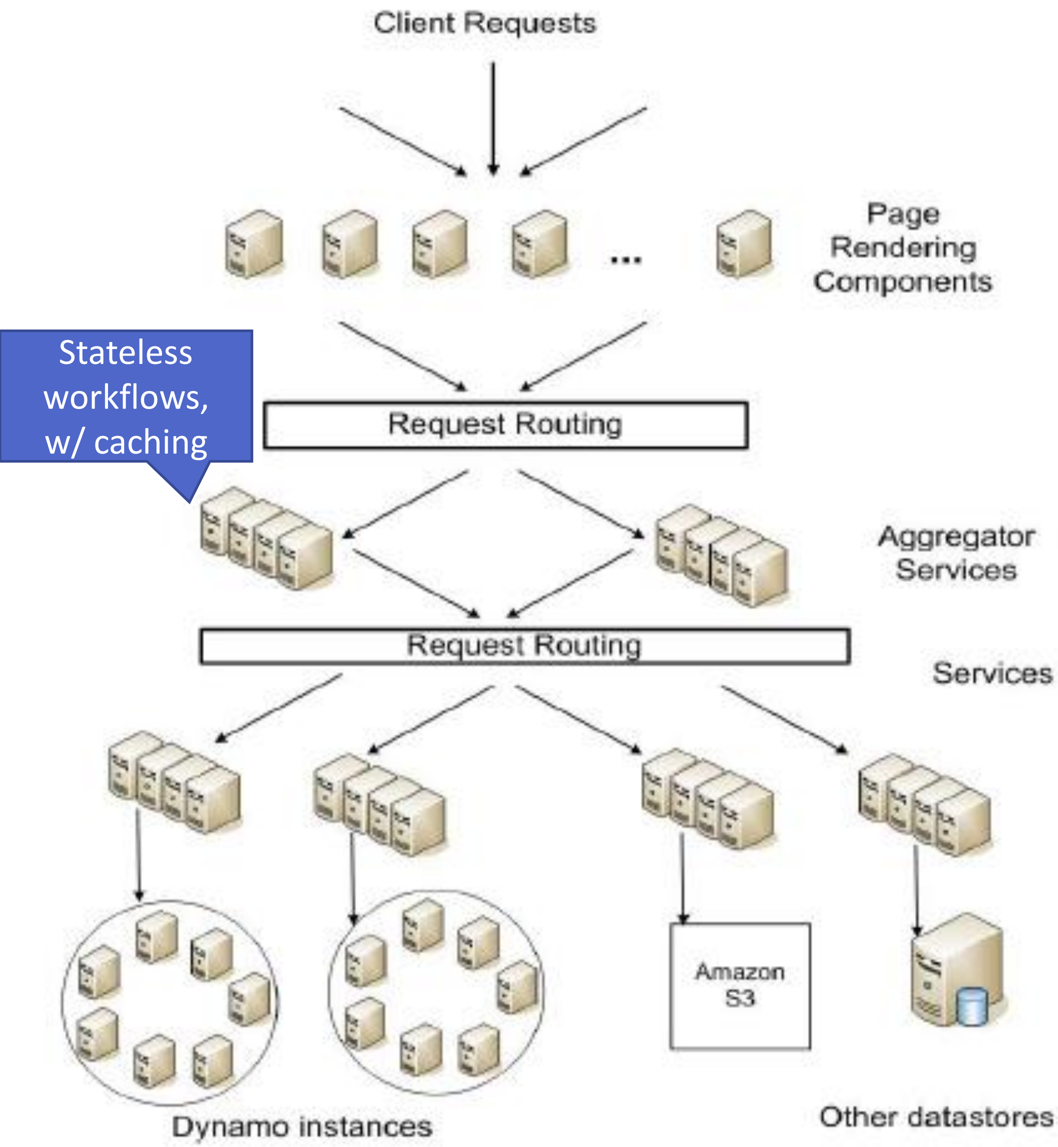


Figure 1: Service-oriented architecture of Amazon's platform

"build a system where ***all*** *customers have a good experience, rather than just the majority"* (or an average number)

"SLAs are expressed and measured at the 99.9th percentile of the distribution… based on a *<u>cost-benefit analysis</u>"*

Amazon's Dynamo DB

- Highly Available
  - Even the slightest outage has significant financial consequences
- Service Level Agreements
  - Guaranteeing response in 300ms for 99.9% of requests at a peak load of 500 req/sec
- vs. ACID
  - Weak consistency, no Isolation (since only 1 key op at a time)
- Non-hostile environment, security not a concern

Dynamo: Amazon's Highly Available Key-value Store, Giuseppe DeCandia, et al, SOSP, 2007

Design Principles

- Optimistic replication techniques
  - Changes propagate to replicas in the background,
  - Server and network failures, Concurrent, disconnected work is tolerated
- *When* to perform the process of resolving update conflicts
  - Dynamo targets the design space of an “always writeable” data store
  - Rejecting customer updates could result in a poor customer experience
  - Push the complexity of conflict resolution to the reads
- *Who* performs the process of conflict resolution
  - Done by the data store or the application
  - Application is aware of the data schema, and can select best conflict resolution method

Design Principles

- Incremental scalability (weak scaling)
- Symmetry of nodes’ responsibilities
- Decentralization (P2P)
- Heterogeneity of node’s resources

Design Choices

- Sacrifice strong consistency for availability
  - "always writeable". No updates are rejected.
  - Conflict resolution is executed during ***read*** instead of ***write***, i.e. "always writeable".
- Incremental scalability & decentralization
  - Symmetry of responsibility
  - Heterogeneity in capacity
- All nodes are trusted

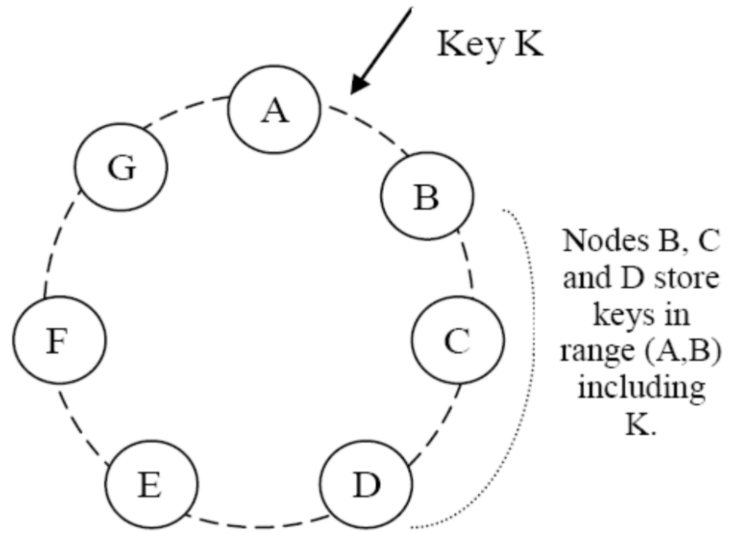
Techniques

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.

From external sources

Partitioning

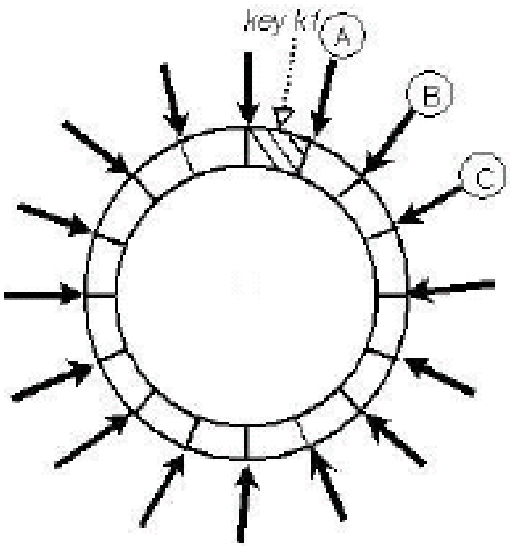
- Consistent hashing
- Output range of hash func. on key is a fixed "ring"
- Virtual node is responsible for a range of hash values (tokens)
  - Hash value for the key maps to a virtual node
- Each physical node responsible for multiple virtual nodes
  - Allows nodes to arrive and leave without having to change keys present in virtual nodes



From external sources

Partitioning and placement of key

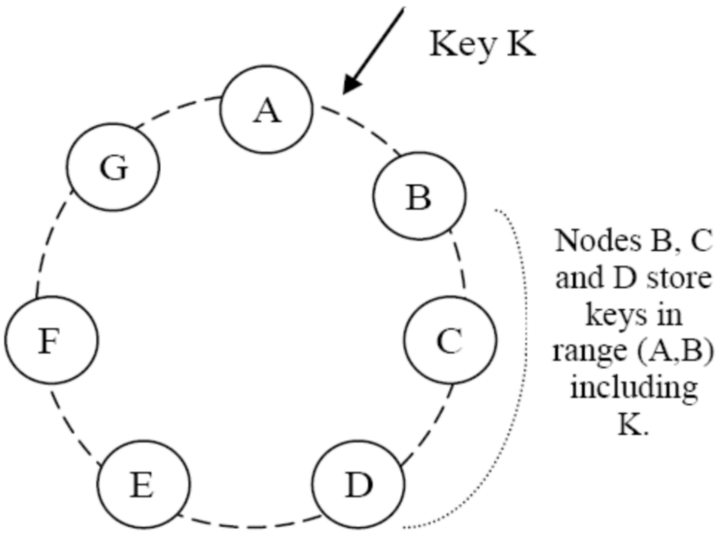
- Divide the hash space into Q equally sized partitions…virtual node or token
- Each physical node assigned Q/S tokens where S is the number of nodes in the system.
  - Can also assign variable tokens to physical node based on machine size
- Adapt to capacity of physical nodes
- Incrementally add/remove physical nodes
  - When a node leaves the system, its tokens (virtual nodes) are randomly & uniformly distributed to the remaining nodes to load balance
  - When a node joins the system it uniformaly "steals" tokens from nodes in the system to load balance



From external sources

Replication

- Each data item is replicated at N hosts.
  - "*preference list*": The list of nodes responsible for storing a particular key.
- *Coordinator node* (from hashing) stores first copy
  - Next copy stored in subsequent virtual nodes
  - Skip virtual nodes present on same physical node
- Gossip protocol
  - Propagates changes among nodes
  - Each node contacts a peer at random every second and the two nodes reconcile their membership change histories
  - Eventually consistent view of membership, mapping from tokens to nodes
  - "Seeds" to make propagation rapid, avoid partitioning: all peers know of the seeds



D stores (A, B], (B, C], (C, D]

- Permanent node adds and removes are done centrally and notified to peers
  - If a peer cannot reach a another, it must be a transient error

From external sources

Key Value Operations

- Add and update items both use **put(key, value)** operation
- **get(key)** returns the value
- Any node may receive the request
- Forwarded to the coordinator node for response
- put() and get() are sent to all N "healthy" replicas, but…
- **put()** may return to its client before the update is applied at all replicas
  - May leave replicas in inconsistent state
- **get()** may return many versions of same object

Sloppy Quorum

- Writes are successful if ‘w’ replicas out of N can be updated (w<N)
  - *Coordinator forwards requests to all N replicas*, and returns when ‘w’ respond
  - Vector clock generated at coordinator is forwarded
- Reads return all ‘r’ replica values (r<N)
  - *Coordinator sends requests to all N replicas*, and returns when ‘r’ respond
  - Coordinator returns *causally unrelated* copies
  - Clients need to decide how to use these copies
- Reads & writes dictated by *slowest replica*
  - Set **r+w > N**

From external sources

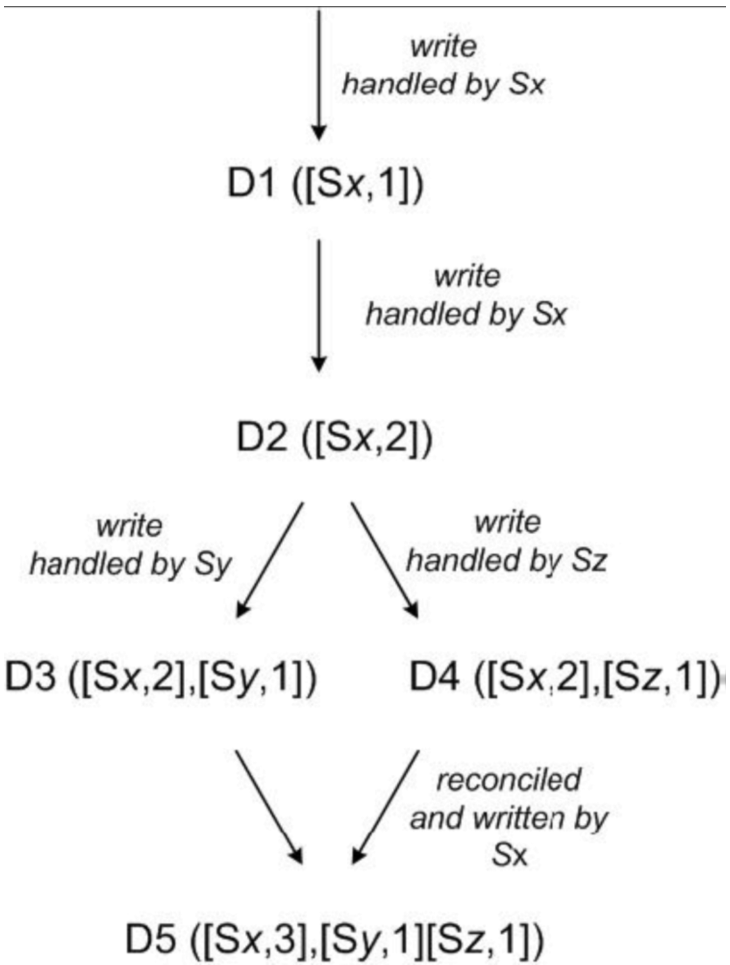
Data Versioning & Consistency

- put() is treated as *append* of the updated value
  - Immutable append to a *particular version* of the object
  - Multiple versions can coexist…but system will not internally "resolve" them
- Challenge
  - *Distinct version sub-histories need to be reconciled.*
- Solution
  - *Uses vector clocks to capture causality between different versions of the same object.*

From external sources

Consistency with Vector Clocks

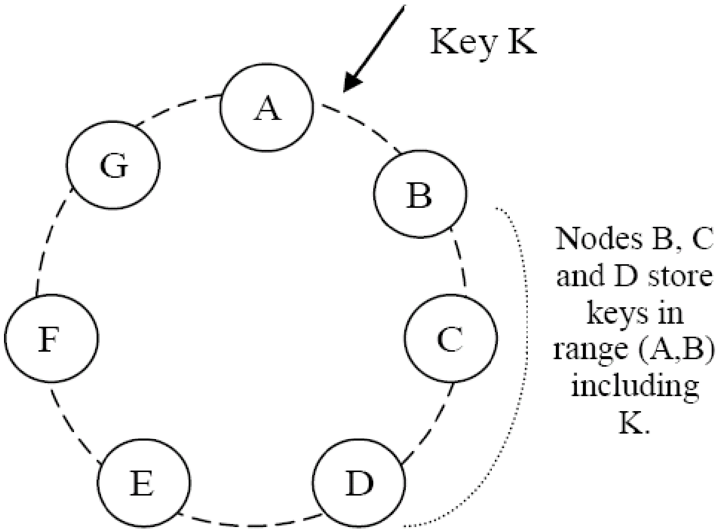
- Vector clock: (node, counter) pair
  - Every version of every object is associated with one vector clock.
  - If the counters on the first object's clock are <= all nodes in the second clock, then the first is an ancestor of the second and can be forgotten.
  - i.e. first object happened before second object
- If get() has multiple replica versions, return causally "unrelated" versions
  - *i.e. remove partial ordered & only return causally unordered versions for reconciliation*
- Client writes the reconciled version back
  - e.g. Sx resolves D3 and D4 into D5



From external sources

Hinted handoff

- Useful for transient failures
- Assume N = 3. When A is temporarily down or unreachable during a write, send replica to D.
- D is hinted thru metadata that the replica belongs to A (but A was down)
- D maintains write in a separate local DB. D will deliver writes to A when A is recovered.
- Again: “always writeable”



From external sources

Replica synchronization

- Handles permanent failures
  - Resolve replica inconsistency faster
  - Reduce data transfer
- 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.
- Advantage:
  - Each branch of the tree can be checked independently without requiring nodes to download the entire tree.
  - Help in reducing the amount of data that needs to be transferred while checking for inconsistencies among replicas.

From external sources

Replica synchronization

- Anti-entropy schemes when hinted hand-off does not work
  - Replicas have different subsets of writes
- Build a Merkle tree for common ranges of keys among replicas
  - N trees if there are N replicas
- Check root, then children if they don't match, etc.
  - Minimizes data transfer

From external sources