Large-Scale Key-Value Stores
Eventual Consistency

Marco Serafini
Consistent Hashing
Consistent Hashing

- Each node has a membership set \( M \)
- When a node needs to access a key
  - It hashes the IDs of the nodes in \( M \) to a ring \((\text{mod } n)\)
  - It hashes the key to the same ring \((\text{mod } n)\)
  - Access goes to the next “successor” node in ring

In this example:
- \( n = 8 \)
- Hash function is identity for simplicity
Membership Changes

- Node joins: take <k,v> pairs from successor
- Node leaves: give <k,v> pairs to successor
- **Local changes**, no global reconfiguration → good for churn
Theoretical Results

Theorem 1. For any set of $N$ nodes and $K$ keys, with high probability:

1. Each node is responsible for at most $(1 + \epsilon)K/N$ keys

2. When an $(N + 1)^{st}$ node joins or leaves the network, responsibility for $O(K/N)$ keys changes hands (and only to or from the joining or leaving node).

• Q: What do these results tell us?
• $\epsilon$ is arbitrarily small with $O(\log N)$ virtual nodes
  • Virtual nodes: multiple keys associated to the same physical node
Dynamo: Amazon’s Highly Available Key-value Store

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

Amazon.com

ABSTRACT
Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in the world; even the slightest outage has significant financial consequences and impacts customer trust. The Amazon.com platform, which provides services for many web sites worldwide, is implemented on top of an infrastructure of tens of thousands of servers and network components located in many datacenters around the world. At this scale, small and large components fail continuously and the way persistent state is managed in the face of these failures drives the reliability and scalability of the software systems.

This paper presents the design and implementation of Dynamo, a highly available key-value storage system that some of Amazon’s cloud services depend on. Dynamo’s design is driven by the need to support hundreds of services of varying sizes and the need to be able to scale to meet the varying demand of the services. Dynamo is a highly available, consistent, fault-tolerant, scalable and simple key-value store.

One of the lessons our organization has learned from operating Amazon’s platform is that the reliability and scalability of a system is dependent on how its application state is managed. Amazon uses a highly decentralized, loosely coupled, service oriented architecture consisting of hundreds of services. In this environment there is a particular need for storage technologies that are always available. For example, customers should be able to view and add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornados. Therefore, the service responsible for managing shopping carts requires that it can always write to and read from its data store, and that its data needs to be available across multiple data centers.

Dealing with failures in an infrastructure comprised of millions of servers is a complex and ongoing challenge. There are many challenges such as managing large distributed systems and building fault-tolerant, consistent, scalable, high-throughput software systems. Building such systems is like developing a new kind of supercomputer and thus requires new thinking about parallelism and distributed systems. There is no prior experience and existing algorithms and theory are insufficient. We are in the process of building something unprecedented. This paper presents Dynamo and the thinking that went into it.
Goals of Key-Value Stores

• Export simple API
  • put(key, value)
  • get(key)

• Simpler and faster than a DBMS
  • Less complexity, faster execution

• Varied forms of consistency
  • Typically no support for transactions (multi-key)
  • Sometimes even updates to the same key are not consistent
NoSQL

• Key-value stores are a typical “NoSQL” system
• Properties of NoSQL
  • Do not require relational schema
  • Do not use SQL
  • Weak consistency
CAP: Three Properties

- Consider a distributed data store for key-value pairs
- Data is replicated for fault tolerance and latency
- Three properties are desirable
  - **Consistency**: system behaves as if non-replicated
  - **Availability**: every client request is served
  - **Partition tolerance**: system can withstand network partitions
**CAP “Theorem”**

- C, A, P: pick two
- Examples
  - A+C: Strongly consistent system, no P
  - A+P: Weakly consistent system, no C
  - C+P: Trivial (no A required, system does nothing)
- DBMS are typically A+C systems
  - Replication is good for fault tolerance, bad for latency
- NoSQL stores are typically A+P
  - Replication is good for latency, bad for consistency
Eventual Consistency

• Each storage node commits locally
• Commits are pushed to other nodes asynchronously
• Conflicts are merged with deterministic criteria
Dynamo

- Large scale key-value store
- Partitioned, fault tolerant
- Strict Service-Level Agreement (SLA)
  - Upper bound on 99.9% percentile low latency
  - This is called *tail latency*
Replication and Eventual Consistency

- Each key is replicated in a *preference list* of nodes
- Eventually consistent
  - Updates go to first $W$ healthy nodes in preference list
  - Read and write quorums might not intersect
  - Later reconciliation in presence of inconsistency
- If a node in preference list is not reachable, skip and try to recontact later (hinted handoff)
Quorums

• Sequential consistency: $W+R > N$
• Weak consistency: $W+R \leq N$
• Q: How to set $W$ and $R$ to achieve persistency with $f$ crashes AND weak consistency?
Versioning: Vector Clocks

- One entry per node
- Node increments its entry when updates
- $v_1 > v_2$ if every entry of $v_1$ is $\geq$ than the one of $v_2$ and at least one is $>$
- If two vectors cannot be ordered, conflict