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

SOSP 2007

Intro

- Reliability and scalability are key operational requirements.
- Data, such as shopping carts, must be always available.
  - Even in the presence of hardware failure, communication failure, or natural disaster.
- The storage technology behind the shopping cart must ensure data can always been read/written and data needs to be available across data centers.
- Failure is the norm.
  - Shouldn't impact performance.

Intro

- Dynamo: a highly available and scalable distributed data store.
- Relational databases are too heavyweight for simple primary key access.
- Allows tradeoff between availability, consistency, cost-effectiveness and performance.

Intro

- Key features:
  - data is partitioned and replicated using consistent hashing
  - consistency is facilitated by object versioning
  - consistency among replicas during updates is maintained by a quorum-like technique and a decentralized replica synchronization protocol.
  - gossip based distributed failure detection and membership protocol
  - completely decentralized system with minimal need for manual administration
  - storage nodes can be added and removed from Dynamo without requiring any manual partitioning or redistribution

Background

- Complex query management provided by RDBMS requires expensive hardware and operational overhead.
- Traditional DBs favor consistency over availability.

System Assumptions

- Query model: support read/write on data identified by a key.
- ACID Properties: trade consistency for availability.
- Efficiency: runs on commodity hardware. Provides throughput and latency guarantees.
- Assumptions: environment is non-hostile; scale up to hundreds of nodes.

Design Considerations

- Use eventual consistency to provide high availability.
  - All updates eventually reach all replicas.
  - Dynamo is always writeable, so conflicts are resolved during reads.
  - The application provides conflict resolution.

Other Design Choices

- Incremental scalability
- Symmetry
- Decentralization
- Heterogeneity

Interface

- Two operations: get(key), put(key, context, object)
  - context includes metadata such as the version of the object.
  - An MD5 hash on the key determines where object should be stored.

Partitioning

- Consistent hashing generates a circular ID space (ring)
- Each node is assigned a random position on the ring
- Data's key is hashed and data is stored at first node with position larger than the item's position.
- Virtual nodes help to deal with heterogeneity. Each physical node is assigned several positions (tokens).

Replication

- Each data item is stored on a coordinator chosen as described on the last slide.
- The coordinator replicates the data on N-1 successors on the ring.
- The list of nodes storing a piece of data is the preference list.

Data Versioning

- No updates should be lost.
- Vector clocks maintain versioning information.
- For a put, the client specifies the version it is updating.
- A get request may result in multiple versions returned.

Execution of Operations

- Any get or put can be handled by any node.
- A client can select a node using a load balancer or a partition-aware client library.
- A node (coordinator) in the top N in the preference list handles the read/write. Requests sent to other nodes may be forwarded.
- Parameters R and W specify the number of nodes that must participate in a successful read/write.
- put - coordinator generates the new vector clock, writes the version locally, and sends to W-1 other nodes from the preference list. If W-1 respond, the write is successful.
- get - the coordinator gets R-1 versions of the data from nodes in the perference list and possibly returns multiple replicas.

Hinted Handoff

- Reads/writes are performed on the first N healthy nodes found by the coordinator.
- If a node is down, data will be sent to the next node in the ring.
- This node will keep track of the intended recipient and send later.
- Replicas are stored at multiple data centers.
- Merkle trees are used to keep replicas synchronized without significant overhead.

Membership

- An adminstrator adds and removes nodes from the ring.
- Every second, each node contacts another to exchange membership information.
- At startup, a node chooses a random set of tokens and writes the selection to disk. This information is exchanged during gossiping.
- Some nodes are are seeds. All nodes know about seeds and eventually send them their membership info.
- When a node finds out about a new node that should store some of the data it currently stores, it offers the data to the new node.

Lessons Learned

- Typical values for N, R, W - (3, 2, 2)
- Experiments consider several hundred nodes on multiple data centers.

Results

- 99.9 percentile latencies for reads/writes over 30 days
  - Patterns are diurnal because requests are diurnal
  - Writes slower than reads because of disk access