Dynamo: Amazon’s Highly Available Key-value Store

Presentation by: Steven van Beelen
Outline

- Introduction
  - So, what’s the problem?
  - What is Dynamo?
- How does Dynamo meet requirements
  - Partitioning
  - Replication
  - Versioning
  - Membership
  - Failure handling
- Results
- Conclusion
Introduction

- Amazon: Big e-commerce platform
  - Many customers -> need to be satisfied
  - Big requirements

- Requires:
  - Availability/Reliability
  - Scalability
So, what’s the problem?

- Dealing with requirements in a:
  - Decentralized-
  - loosely coupled-
  - and service oriented-
  - System

- Asks for always available and scalable storage technologies
What is Dynamo?

- Simple:
  - A datastore

- More precisely:
  - Primary-key (PK) only datastore
  - PK interface makes it quicker
  - Tailored for high availability

- Usages:
  - Different services
  - PK access only services
How does Dynamo meet requirements

- Many problems in such a system
  - Partitioning
  - Replication
  - Versioning
  - Membership
  - Failure Handling

<table>
<thead>
<tr>
<th>Problem</th>
<th>Technique</th>
<th>Advantage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partitioning</td>
<td>Consistent Hashing</td>
<td>Incremental Scalability</td>
</tr>
<tr>
<td>High Availability for writes</td>
<td>Vector clocks with reconciliation during reads</td>
<td>Version size is decoupled from update rates.</td>
</tr>
<tr>
<td>Handling temporary failures</td>
<td>Sloppy Quorum and hinted handoff</td>
<td>Provides high availability and durability guarantee when some of the replicas are not available.</td>
</tr>
<tr>
<td>Recovering from permanent failures</td>
<td>Anti-entropy using Merkle trees</td>
<td>Synchronizes divergent replicas in the background.</td>
</tr>
<tr>
<td>Membership and failure detection</td>
<td>Gossip-based membership protocol and failure detection.</td>
<td>Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.</td>
</tr>
</tbody>
</table>

- All solved with their own techniques
Partitioning

- Scaling Incrementally
  - Dynamic Data Partitioning
  - Done by: Consistent Hashing technique

- Original Consistent Hashing:
  - Output range of hash = ring structure
  - Nodes get position in ring
  - Data items assigned to position
Partitioning

- Advantage
  - Ease of removal/arrival nodes

- Challenge
  - Random position: Non-uniform load distribution

- Dynamo Consistent Hashing:
  - Each node has multiple positions
Replication

- **Data item**
  - Replicated at N hosts
  - Coordinator does replication

- **Node responsibility:**
  - From itself to Nth predecessor
Versioning

- Eventual consistency
  - Asynchronous updates
  - Can lead to different versions

- Vector clocks used per object
  - Capture causality between versions
  - To later solve inconsistent versions
Versioning

Example:

D1 ([Sx,1])

write
handled by Sx

D2 ([Sx,2])

write
handled by Sx

D3 ([Sx,2],[Sy,1])

write
handled by Sy

D4 ([Sx,2],[Sz,1])

write
handled by Sz

D5 ([Sx,3],[Sy,1][Sz,1])

reconciled
and written by
Sx
Membership

- **Two Variants**
  - Ring Membership & Failure Detection

- **Ring Membership**
  - Membership in ring done manually
  - Propogated through gossip-based protocol

- **Failure Detection**
  - Local view of failure sufficient
  - Node detects failure only when that other node is needed
  - After detection, periodical recovery check
Failure Handling

- Two versions:
  - Temporary and Permanent failures

- Temporary failures
  - Use of ‘Sloppy Quorum’ and ‘Hinted Handoff’

- Assume:
  - Node A fails during write
  - Write goes to replica D
  - Replica has ‘hint’ that the write is not his
  - Periodical scans to check recovery
  - If recovered, write delivered
Failure Handling

- Permanent Failures
  - Use of anti-entropy protocol with Merkle Tree

- Merkle Tree:
  - Tree consisting of hashes
  - Leaves: hashes of individual keys
  - Parents: hashes of their children
  - Advantage: parts can be checked without need of whole tree
  - Example: two root hashes equal -> trees are equal

- Each node has Merkle Tree for its key-range
Results: Durability/Availability

- First: all tests run in real production environment
- Graph: Average and 99.9 percentile latencies
- As seen: upperbound is good, average is real good (availability)
Results: Performance

- Graph: Comparison between buffered and un-buffered writes
- As seen: Buffered writes boost performance
Results: Load Distribution

- Graph: Node imbalance and request load
- As seen: the higher the request, the better the balance
Conclusion

- Different results show off different possibilities
  - Dynamo can be tailored towards specific services

- Techniques described give scalability and availability

- T.H.M.:
  Dynamo is a great, scalable, highly available data store.
  But, it is just for Amazon
Questions?
System Assumptions and Requirements

- **Query Model:**
  - Simple read/write operations
  - Binary objects identified by unique key

- **ACID properties:**
  - Only C(onsistency) of less importance

- **Efficiency:**
  - Important for customer experience
  - Measured on 99.9th percentile

- **Other:**
  - Used in internal services, hence non-hostile environment
  - Each service own Dynamo instance
Design Considerations

- Incremental Scalability
- Symmetry
- Decentralization
- Heterogeneity
What’s in it for Science!?

- Showing how:
  - Different techniques together to get a highly available system
  - Showing that eventually consistent storage works in production with demanding applications
  - Insight into tuning these requirements