Dynamo

Zeyuan Shang
Agenda

01 Introduction
02 System Architecture
03 Limitation & Comparison
04 DynamoDB Demo
05 Q & A
Introduction

- Background
- Motivation
- Assumptions & Requirements
- Design Consideration
1. Background

• Amazon’s e-commerce platform is getting big
• But performance is getting worse…
• A Shopping Cart
The platform is implemented on top of an infrastructure of **tens of thousands of** servers and network components located in **many datacenters** around the world.

Persistent state is managed in the face of these failures - drives the **reliability** and **scalability** of the software systems.

Guarantee **Service Level Agreements** (SLA): In Amazon’s platform, services have stringent latency requirements which are in general measured at the 99.9th percentile of the distribution.
3. Assumptions & Requirements

- **Query Model**: Key/value for storing objects with small size (less than 1MB)
- **Consistency**: ACID is bad according to experiences at Amazon, thus Dynamo provides weaker C (eventual consistency)
- **SLA**: 99.9% Latency must be bounded, while average is not enough
- **Security**: No security-related problems such as authentication and authorization
4. Design Consideration

01 Highly Available
Sacrifice strong consistency for availability

02 Always Writable
Conflict resolution is executed during read instead of write

03 Scalability
able to scale out easily, with minimal impact

04 Symmetry & Heterogeneity
Same responsibilities and decentralization; adding new nodes with higher capacity without having to upgrade all hosts at once
System Architecture

- Partitioning
- Replication
- Failure Recovery
- Membership & Failure Detection
- Execution of Put and Get
- Implementation
- Summary
1. Partitioning – Consistent Hashing

- Consistent Hashing: the output range of a hash function is treated as a fixed circular space or “ring”; for example, $K \mod N$
- Each node becomes responsible for the region in the ring between it and its predecessor node on the ring
- A node handling a read or write operation is known as the coordinator. (typically the first among the top $N$ nodes in the preference list)
- Preference List: nodes storing corresponding key
- Arrival/Departure of nodes only affects neighbors

![Diagram showing nodes A, B, C, D, E, F, G on a circular ring.](example-diagram)
1. Partitioning – Virtual Nodes

- Virtual Nodes: each physical node can be responsible for more than one virtual node.
  - When a node becomes unavailable...
  - When a node becomes available again
  - Accounting for heterogeneity
Eventual Consistency: if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value.
A `put()` call may return to its caller before the update has been applied at all the replicas.
A `get()` call may return many versions of the same object.
Challenge: reconcile different versions.
Example: Shopping Cart.
Solution: using **vector clock** to capture causality between different versions of the same object.
A vector clock is a list of (node, counter) pairs, for example, [(A, 1), (B, 2)]
Every version of every object is associated with one vector clock
If the counters on the first object’s clock are less-than-or-equal to all of the nodes in the second clock, then the first is an ancestor of the second and can be forgotten, ELSE they are conflict, for example, [(A, 1), (B, 1)] < [(A, 2), (B, 1), (C, 1)]
Example
2. Replication – Vector Clock Example

```
write
handled by Sx

D1 ([Sx,1])

write
handled by Sx

D2 ([Sx,2])

write
handled by Sy
write
handled by Sz

D3 ([Sx,2],[Sy,1])  D4 ([Sx,2],[Sz,1])

reconciled
and written by Sx

D5 ([Sx,3],[Sy,1][Sz,1])
```
2. Replication – Vector Clock

- In case of network partitions or multiple server failures, the size of vector clock may grow.
- Along with each (node, counter) pair, Dynamo uses timestamp to remove oldest one when the size is beyond the threshold.
- However, this problem has not surfaced in production.
• Assume N = 3. When A is temporarily down or unreachable during a write, send replica to D
• D is hinted that the replica belongs to A and it will deliver to A when A is recovered
• Again, **Always Writable**
• 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.
• How Dynamo uses Merkle Tree
3. Failure Recovery – Merkle Tree Example

The diagram illustrates a Merkle Tree where each node represents a hash of its child nodes. The root hash (Top Hash) is the hash of the hashes of the leaf nodes (L1, L2, L3, L4).

- **Hash 0**: hash(Hash 0-0  Hash 0-1)
- **Hash 0-0**: hash(L1)
- **Hash 0-1**: hash(L2)

- **Hash 0**: hash(Hash 0-0  Hash 0-1)
- **Hash 0-0**: hash(L1)
- **Hash 0-1**: hash(L2)

- **Hash 1**: hash(Hash 1-0  Hash 1-1)
- **Hash 1-0**: hash(L3)
- **Hash 1-1**: hash(L4)

The data blocks (L1, L2, L3, L4) are at the bottom of the tree, and their hashes are propagated to the root to verify integrity.
• Gossip-based membership protocol and failure detection
• Won’t cover this in today’s talk
5. Execution of Put and Get

- Select a node
  - Route the request through a generic load balancer
  - A partition-aware client library that routes requests **directly** to the appropriate coordinator nodes
- Consistency Protocol: Sloppy Quorum
  - R/W is the minimum number of nodes that must participate in a successful read/write operation
  - Setting R + W > N yields a quorum-like system
  - In this model, the latency of a get (or put) operation is dictated by the slowest of the R (or W) replicas (thus usually less than N)
  - Common (N, R, W) setting is (3, 2, 2)
- Versions in Put/Get
5. Execution of Put and Get

- **Put**
  - Generate new version
  - Send it to other replicas
  - Wait for W - 1 nodes

- **Get**
  - Request all existing versions
  - Wait for R nodes
  - Return all versions that are not causally related
  - Reconciled by the application then written back
Implementation

- Local Persistent and Pluggable Store
  - Berkeley Database (BDB): object of tens of kilobytes
  - MySQL: object of > tens of kilobytes
- All in Java
- Improvement
  - Buffer for read/write
  - Durable write
Highly Available
Always Writable
Scalability
Symmetry & Heterogeneity

Replication, Vector Clock
Partition, Hinted Handoff, Sloppy Quorum, Failure Recovery
Partition, Membership and Failure Detection

01 Design Consideration

01 Highly Available
02 Always Writable
03 Scalability
04 Symmetry & Heterogeneity

Replication, Vector Clock with reconciliations during reading
<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>
Comparison & Limitations

- Comparison
- Limitations
<table>
<thead>
<tr>
<th>Name</th>
<th>DynamoDB</th>
<th>Cassandra</th>
<th>HBase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Hosted, scalable database service by Amazon with the data stored in Amazons cloud</td>
<td>Wide-column store based on ideas of BigTable and DynamoDB</td>
<td>Wide-column store based on Apache Hadoop and on concepts of BigTable</td>
</tr>
<tr>
<td>Database model</td>
<td>Document Store</td>
<td>Wide Column Store</td>
<td>Wide Column Store</td>
</tr>
<tr>
<td>Cloud-based</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Implementation language</td>
<td>Java</td>
<td>Java</td>
<td>Java</td>
</tr>
<tr>
<td>Data scheme</td>
<td>Schema-free</td>
<td>Schema-free</td>
<td>Schema-free</td>
</tr>
<tr>
<td>Typing</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Secondary indexes</td>
<td>yes</td>
<td>Restricted (equality queries)</td>
<td>no</td>
</tr>
<tr>
<td>SQL</td>
<td>no</td>
<td>Restricted (CQL)</td>
<td>no</td>
</tr>
<tr>
<td>APIs and other access methods</td>
<td>RESTful</td>
<td>CQL, Thrift</td>
<td>Java, RESTful, Thrift</td>
</tr>
<tr>
<td>Server-side scripts</td>
<td>no</td>
<td>no</td>
<td>Yes (coprocessor)</td>
</tr>
<tr>
<td>Triggers</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Partitioning methods</td>
<td>Sharding (consistent hashing)</td>
<td>Sharding (consistent hashing)</td>
<td>Sharding (range partition)</td>
</tr>
<tr>
<td>MapReduce</td>
<td>No (only EMR)</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Consistency concepts</td>
<td>Eventual Consistency</td>
<td>Eventual Consistency</td>
<td>Immediate Consistency</td>
</tr>
<tr>
<td>Foreign keys</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Transaction concepts</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>
Limitations

- Object size can not be large
- No cross-region(data center) replication
- Backup process is not very friendly
DynamoDB Demo
• Demo
• DynamoDB Portal
Thank You!

THANK YOU FOR WATCHING