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 from Amazon.com

Presenter: Mingran Peng
EECS 591 2020Fall
Content

• Dynamo Overview

• Detailed Design

• Experiences & Lessons Learned

• Example: DynamoDB
Dynamo Overview
System Model and Requirements

- Key-Value query model
  - Relational query is redundant
- ACID (of course)
  - Atomicity, Consistency, Isolation, Durability
- Efficient
  - 300ms latency
  - Measured at 99.9 percentile
- Other assumptions:
  - non-hostile environment
  - Scalable, of course
Why and What is Dynamo?

• Traditional Database is not a perfect solution
  • Complex query not needed
  • Typically choose consistency over availability

• Amazon wants a highly scalable, available, simple distributed storage system
SLA: Service Level Agreement

• A contract where a client and a service agree on several system-related characteristics

• Example:
• This service will provide a response within 300ms for 99.9% of its requests for a peak client load of 500 requests per second.
Continue: SLA

• Every service should obey its SLA:
  • A service call another services which call more services which call more …

• Why 99.9%?
  • Common metrics are average, median, expected variance
  • Customers!
Additional Design Considerations

• “always writeable”
  • i.e. Solve the conflict during read
  • Why? Customers!

• Sacrifice strong consistency for high availability
  • Why? Customers!

• Incremental scalability, Symmetry, Decentralization, Heterogeneity
  • Basically they mean easy to scale, proper load balance, high failure tolerance
Detailed Design
System Interface

- Get(Key)
- Put(Key, Object, Context)

- What is Context?
  - Context contains other important information
  - Such as version information
    - Remember “always writeable”, so there exists multiple versions of course
Partition Algorithm

- There are many keys and many nodes, Dynamo needs to distribute keys to nodes
- All keys are hashed, the hashed value form a ring
- Each node is assigned a random position
- Clockwise to find the node
Partition Algorithm

- Advantage: The arrival or departure of a node only affects neighbor
- Disadvantage: Non-uniform load balance
- Solution: virtual nodes. A node is assigned to multiple virtual nodes
Replication

• N replications: just clockwise go through N nodes.

• Example: N=3, blue arrow pointed key are stored in B,C,D
Data Versioning

- Remember “always writeable”
  - It will cause lots of different versions
  - Solution: vector clock strategy
  - Client share some reconciliation responsibility

- Problems: what if vector clock get too big?
  - Set a limit, if exceeds, drop the oldest write server information
Execution of Get and Put

- First, client needs to route to “coordinator”
  - Coordinator: the smallest ranked node that store the requested key
  - Load balancer routing or client library routing

- Coordinator will broadcast responses will wait for R responses for get() and W responses for put().
  - \( R + W > N \) to guarantee consistency
  - Coordinator will return all versions of Object
Handling Failures: Hinted Handoff

• To deal with temporal failure.

• Example: if B is failed, then the replica information of key K will be sent to E.

• When B recovers, E will handle information back to B
Handling permanent failures: Replica synchronization

• Use Merkle trees to detect the inconsistencies between

• Each node maintains a separate Merkle tree for each key range it hosts.

• 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.
Membership, Failure Detection, Adding/Removing nodes

- When new nodes are added, it chooses multiple tokens (position on hash ring) and knows the partition
  - Partition information reconciled regularly

- Neighbor nodes handle corresponding key range to new node

- Failure detection using gossip based protocol
Implementation

- Java

- Local persistence component allows for different storage engines to be plugged in:
  - Berkeley Database (BDB) Transactional Data Store: object of tens of kilobytes
  - MySQL: object of > tens of kilobytes
  - BDB Java Edition, etc.
EXPERIENCES & LESSONS LEARNED
Different configurations

• Different N, R, W value
  • Usually N,R,W = 3,2,2

• Reconciliation method
  • Timestamp based reconciliation
  • Business logic specific reconciliation
Balancing Performance and Durability

• Latencies follow a diurnal pattern similar to the request rate
  • Most time the client get Responses within 300ms
    • But there is still some data points over 300ms

(hourly plot of latencies during our peak session in Dec. 2006)
Balancing Performance and Durability

• Again, sacrifice consistency for latency

• Maintain a buffer, write only to buffer and periodically write back to storage

• 5 x speed up during peak
Partition algorithm Revisit

• Strategy 1: T random tokens per node and partition by token value:

• Key range handling is a lot work
• Merkle trees recalculation
• Not easy to archive
• Strategy 2 fix the key range partition by dividing the whole ring into $Q$ segments ($Q \gg S \times T$)

• Strategy 3 further align the Token with partition
• Strategy 2 served as an interim setup during the process of migrating Dynamo instances from using Strategy 1 to Strategy 3
Divergent Versions Revisit

- Track the number of versions returned to the shopping cart service for a period of 24 hours.
  - 99.94% of requests saw exactly one version;
  - 0.00057% of requests saw 2 versions
  - 0.00047% of requests saw 3 versions
  - 0.00009% of requests saw 4 versions.

- Divergent versions are created rarely.
Client-driven or Server-driven Coordination

- Recall previously said a client route to coordinator by client library or load-balancing

<table>
<thead>
<tr>
<th></th>
<th>99.9th percentile read latency (ms)</th>
<th>99.9th percentile write latency (ms)</th>
<th>Average read latency (ms)</th>
<th>Average write latency (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server-driven</td>
<td>68.9</td>
<td>68.5</td>
<td>3.9</td>
<td>4.02</td>
</tr>
<tr>
<td>Client-driven</td>
<td>30.4</td>
<td>30.4</td>
<td>1.55</td>
<td>1.9</td>
</tr>
</tbody>
</table>
Balancing background vs. foreground tasks

- background tasks like replica synchronization and data handoff triggered resource contention and affected the performance of the regular put and get operations (foreground tasks).

- Admission control mechanism: use controller to assign runtime slices of the resource (e.g. database) to background tasks
Example: DynamoDB
DynamoDB: Fast and flexible NoSQL service

- NoSQL ≠ NO SQL
- NoSQL means not only SQL
- It’s a database stored using key-value method
- It’s easier to scale than relational database
DynamoDB: Fast and flexible NoSQL service

• Advantages of DynamoDB:
  • Highly scalable
    • Auto scaling!
  • Low latency, consistent performance
    • Measured at 99.9%
  • Flexible
  • ...


DynamoDB: Fast and flexible NoSQL service

- DynamoDB can auto backup tables to other storage, like Amazon S3 bucket
- Remember we talked about partition method.

For strategy 2 and strategy 3, the partition of keys is fixed, each partition can be arranged into one file, which makes backup easier.
DynamoDB: Fast and flexible NoSQL service

• DynamoDB has a feature called **In-Memory Acceleration with DynamoDB Accelerator (DAX)**

• DAX provides lower latency while guarantee eventual consistency
DynamoDB: Fast and flexible NoSQL service

- DAX is more than presented in the paper
  - Users can set up clusters. All nodes in cluster served as cache using their memory
  - Client can specify its request to read/write from Cluster or from real DB
Questions?
Thanks for listening!