CS417: Distributed Systems

Large Scale Data Stores

Department of Computer Science The Johns Hopkins University

Key Value Stores:

key \rightarrow Value

Requirements: Scalability and Reliability

- Sharding – distribute keys/indexes
- Replication - same key/index on multiple machines

Example – DNS
Domain Name Service: maps URLs to IP addresses
Distributed Indexing

Distributed systems have two engineering aspects:

- A highly efficient sharding mechanism.
- A lookup mechanism that tracks down the node holding the object.

These can be used to implement a higher-level services.

Simple Sharding Mechanism

Let us consider doing it by simple hashing -

\[ \text{store} = \text{hash(key)} \% \text{stores.count} \]

Issues:
- What if stores.count changes?
- What if keys are non-uniformly distributed?
Consistent hashing

1. Imagine the hash space as a ring.

2. Take server/DB’s IP, hash it and place them on hash ring
Consistent hashing

3. Distribute keys – hash(key) and map it onto the ring
- The key resides on the first server in clockwise direction

Q1. What happens if stores.count changes?

O(K/N) keys remapped per server change
Consistent hashing

Q2. How to achieve uniform distribution?

Consistent hashing

Virtual Nodes:

Fall '19
Sahiti Bommareddy and Yair Amir
Lookup

A lookup mechanism that tracks down the node holding the object for the client efficiently.

Let us look at Chord's node join and startup mechanism

Naive Approach

Fall '19  Sahiti Bommareddy and Yair Amir  11

Naive Approach

Fall '19  Sahiti Bommareddy and Yair Amir  12
Chord’s Approach with Finger Tables

Finger Table: 8
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>N8+1</td>
<td>14</td>
</tr>
<tr>
<td>N8+2</td>
<td>14</td>
</tr>
<tr>
<td>N8+4</td>
<td>14</td>
</tr>
<tr>
<td>N8+8</td>
<td>21</td>
</tr>
<tr>
<td>N8+16</td>
<td>38</td>
</tr>
<tr>
<td>N8+32</td>
<td>42</td>
</tr>
</tbody>
</table>

\[
finger[k] = [n + 2^{k-1}] \mod 2^m \\
1 \leq k \leq m
\]

Fall '19  Sahiti Bommareddy and Yair Amir
Efficiency of Chord’s Lookup

Overall, log(n) hops for lookup in the worst case! – very good.

- What is a hop? Where are the nodes? Is log(n) really good?
- What about churn?
- Is it really log(n) worst case over time?
- How to maintain robustness?
Replication

How to fit replication into this?

The key is stored on first N servers in the ring.

Consensus Protocols.
Kelips

- Developed at Cornell (2003).
- Uses more storage ($\sqrt{n}$ instead of $\log(n)$) at each node. – Replicating each item at $\sqrt{n}$ nodes.
- Aims to achieve $O(1)$ for lookups.
- Copes with churn by imposing a constant communication overhead. – Although data quality may lag if updates occur too rapidly.
- How would you do that?

Kelips Lookup

- $N$ is approximate number for the number of nodes.
- Each node id is hashed into one of $\sqrt{N}$ affinity groups.
- Each key from $(\text{key}, \text{value})$ pair is hashed into one of the $\sqrt{N}$ groups.
- Approximately $\sqrt{N}$ replicas in each affinity group.
- Pointers are maintained to a small number of members of each affinity group. Lookup is $O(1)$.
- Weak consistency between the replicas is maintained using a reliable multicast protocol based on gossip.
Operations in Replicated DHTs

How is the most recent version determined?

Leader/Coordinators give each write update a timestamp, based on its local clock.
Clock-based Timestamps

- Each put is given a timestamp by the coordinator responsible for that update.
- If a replica receives an update with a lower timestamp than its current version, it ignores the update, but acknowledges that the write was successful.
- If the clocks on different coordinators drift, this can cause unexpected behavior.

Example: Losing an update

Let's say \( C_2 \) has a slower clock.
Resolution

- Applications need to handle
- How does your application want to handle these?

Operations in Replicated DHTs

Setting –
- A key is in N stores.
- When a request reaches one of them, that store becomes leader/co-ordinator for that operation.
Reads / Get(k)

• Coordinator requests the object from the relevant N nodes
• After R of those replicas respond, the coordinator returns the most recent version held by any of the replicas

Writes / Put(k,v)

• Coordinator forwards the update to the relevant N replicas.
• After W of those replicas have acknowledged the update, the coordinator can tell the client that the write was successful.
Quorum Flavors

Choice 1: R+W > N (Strong Consistency)
Every read quorum will contain a node with the latest write.


Choice 1: R+W < N (Weak Consistency)
Staleness

Crashes and recovery

To address this, nodes participate in anti-entropy with nodes that share a key-range by exchanging Merkle hash trees
Some existing distributed systems for mega services

To state some –

Amazon’s Dynamo
Facebook’s Cassandra
Google’s Slicer is alternative to Consistent hashing used by above two

Each with their own design considerations specific to their application

Hyperdex

Limited API for KV Stores – search based on only key

Want richer service –

High Performance, Scalable, Consistent, Fault-tolerant Data Stores

+ Supports efficient search on secondary attributes

Hyperdex
Hyperspace Hashing

Each server is responsible for a region of the hyperspace

- In addition to partitioning based on the key, each object is stored on additional servers based on its secondary attributes
- Combining the hashes of a set of secondary attributes forms a hyperspace which can be partitioned
- This enables efficient search by limiting the number of servers that need to be contacted
- This can be done for multiple sets of attributes
Hyperspace Hashing

Attribute values are hashed independently
Any hash function may be used

Phone Number

H("607-555-1024")

First Name

H("Neil")

Last Name

H("Armstrong")

Sahiti Bommareddy and Yair Amir

Fall '19

Hyperspace Hashing

Different objects reside at different coordinates

- Every object is placed on a server based on its primary key (like Cassandra and Chord)
- For each set of attributes (subspace) we would like to search by, we will place each object on an additional server
- For each object, its attributes are hashed into a point in the hyperspace, and object is placed on the server responsible for that point
Hyperspace Hashing

- By specifying more of the secondary attributes, we can reduce the number of servers that need to be searched.
- If all of the subspace attributes are specified, the search is equally efficient as searching by key.
- What if the secondary attributes are updated?

Answer: Value dependent Chaining

Value-Dependent Chaining

- Initially, let's assume there are no faults.
- To perform an update, all of the servers involved are organized in a chain:
  - The server responsible for the primary key is at the head of the chain.
  - Any server holding the current version of the object is in the chain.
  - Any server that will hold the updated version of the object is also in the chain.
- The update is ordered at the head and passed through the chain.
- Once it reaches the end of the chain, the tail server can commit the update, and pass an acknowledgment back through the chain.
- Updates are not committed until an acknowledgement is received from the next server in the chain.
Value-Dependent Chaining

Each put takes a forward pass down the chain and is committed during the backward pass.
Value-Dependent Chaining

When updating an object, the value-dependent chain includes the servers which hold the old and new versions of the object.

Consistency Guarantees

- Any operation that was committed before a search will be reflected in its results.
- In the presence of concurrent updates, either version may be returned, but at least one version of every object will be seen.
- Because an update can be reflected in a search before it is committed, search results may be inconsistent with get calls.
Fault Tolerance

- Each server in the chain can be replicated
- Hyperdex uses chain replication, but any consistent replication protocol could be used
- If every block of replicas remains available, the system remains available
Hyperdex: Conclusions

• Search can scale by partitioning on attributes other than the primary key
• What is the cost?

• What is the cost?
  • More servers
  • Higher latency
  • Lower resiliency
Geo-replicated Key Value Stores

Build it for the Globe!

So think of replication across data centers.

Some such systems-
• Google’s Spanner
• Consus, logical successor to HyperDex

References

• https://en.wikipedia.org/wiki/Consistent_hashing
• https://dl.acm.org/citation.cfm?id=1773922
• https://dl.acm.org/citation.cfm?id=1294281
• https://arxiv.org/abs/1612.03457