Practical Lab

Cloud Systems Engineering

(cloud-lab)

Chair of Decentralized Systems Engineering
https://dse.in.tum.de/

Task #2: Distributed KVS

Sharding

Learning goals

In this task you will:

- Learn to improve performance by means of sharding across multiple servers
- Learn about key sharding and sharding strategies
- Build your first distributed KV store

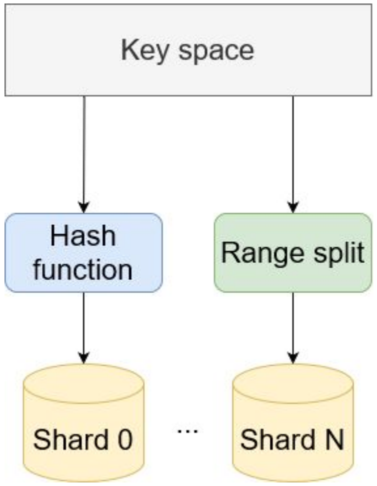
Background

Why distribute a single-node KVS?

- Fault-tolerance
  - A single node is a single point of failure
  - If this node fails, the system become unavailable
- Performance
  - A single node serves all PUTS/GETS for all keys
  - The system’s throughput = the node’s throughput
  - A single node is a bottleneck by definition
- Solutions:
  - Replication (next task !)
  - Sharding (this task)

Sharding (½)

- Is partitioning the key-space over a set of shards
  - Shards are mapped to groups of nodes, one-to-one
- Equivalently, a shard contains a subset of key-value pairs
  - And serves PUT/GET requests for this subset only
- Ideally, the partitioning should be uniform
  - They keys are uniformly distributed across the shards



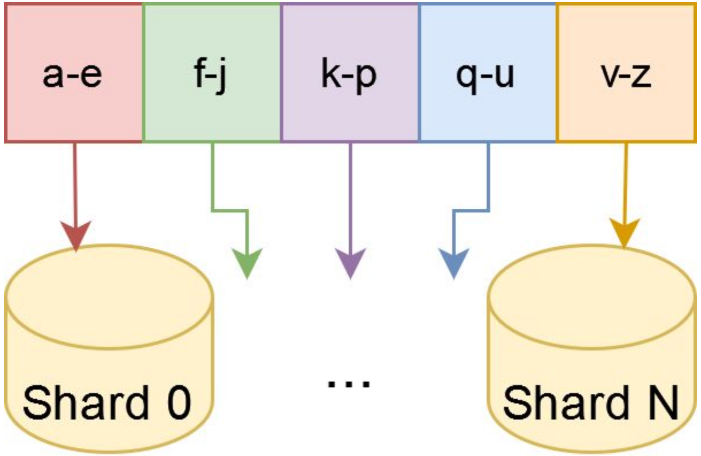
Sharding (2/2)

- Sharding directly enables *horizontal* scaling
  - More shards -> more throughput (linear scaling ideally)
  - Load-balancing across shards -> lower operation latency
  - Storage space scales linearly with the number of shards
- Also provides *partial* fault-tolerance
  - If some shards fail, the rest can still serve requests
  - A subset of the key-space remains available
- The sharding strategy should support flexible reconfiguration
  - Adding/removing shards affects a few shards
  - The key-subspace for the active shards should be automatically re-configured

Sharding strategies (½)

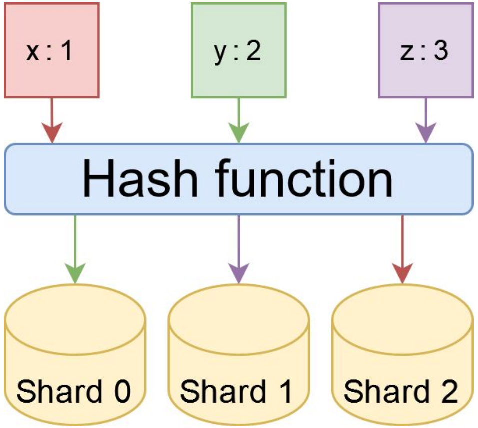
There are mainly two popular sharding strategies: Range and Hash based

- Range strategy:
    - The key-space is divided into equal ranges in a sequential manner
    - Each range is then mapped to a shard
    - Ideal for range-queries (e.g. fetch all values with keys in (*a-j*))
    - Suffers from key-hotspots (e.g. keys *k-p* are always accessed while keys *v-z* never)



Sharding strategies (2/2)

- Hash strategy
  - Maps keys to shards by applying a hash function on them
  - Simplest hash function: *shard = key % number_of_shards + 1*
  - Solves the hotspot problem, if the hash function provides uniform distribution
  - Inefficient for range-queries (sequential keys are scattered in different shards)
- In this lab we focus on simple PUT/GET operations and scalability, thus we choose a hash-based sharding strategy



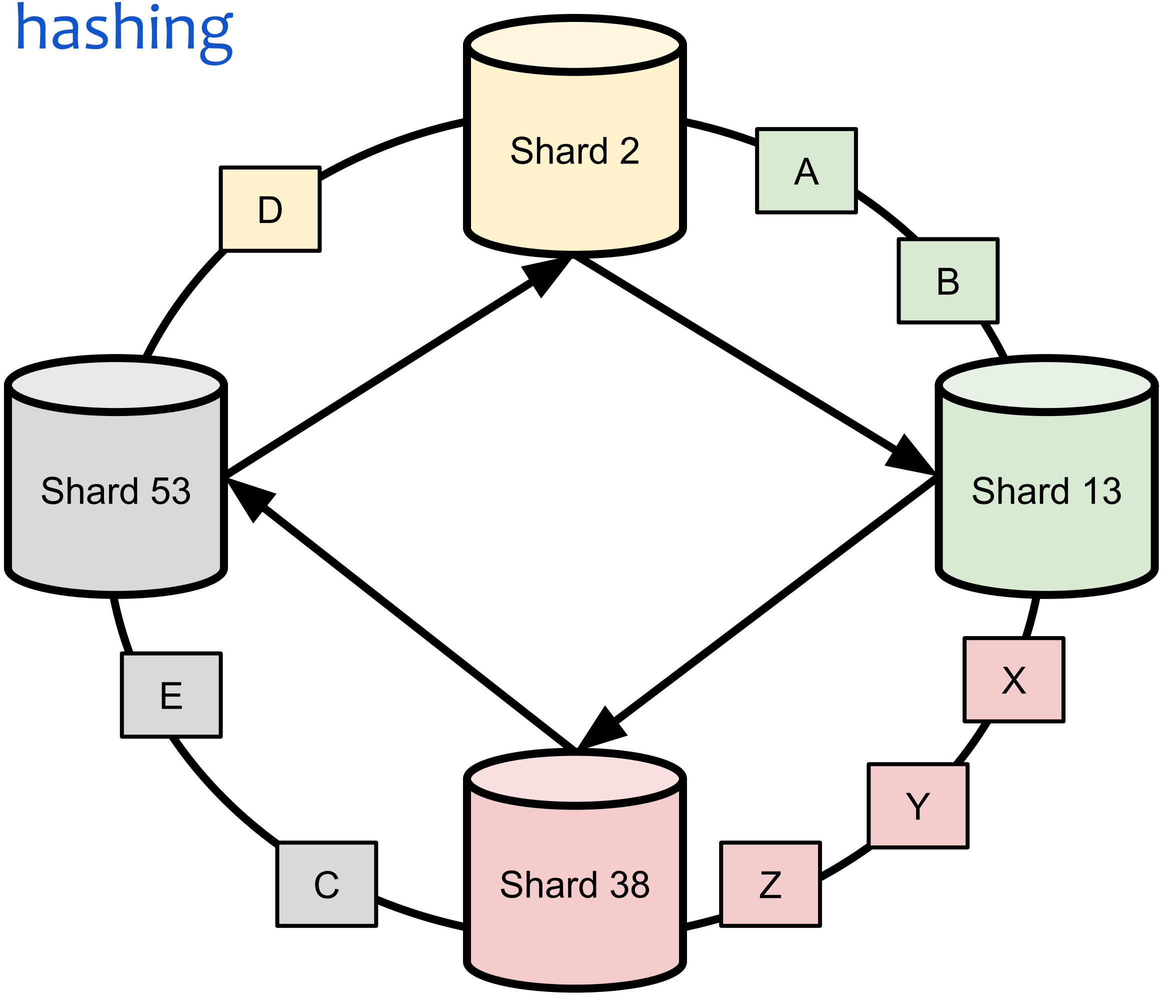
Consistent hashing

- Shard = (key % num_servers) + 1
  - Change in number of shards means keys have to be remapped to new shards
- Solution: Consistent hashing
  - Reduces number of remappings
- Advantages
  - Independent of number of shards or the keys
  - Allows system to scale easily without affecting existing shards

Consistent hashing

- The shards are placed on the edges of a circle
  - The shards are placed based on their unique IDs, ordered clockwise
- The keys are hashed to some values which now lie on the circle as well
  - Some keys may be placed on the same spot with shards, others not
- Both the keys and shards are assigned an m-bit ID
- For each key that doesn’t overlap with a shard, assign it to the next shard in the circle (clockwise)

Consistent hashing



Finger tables

If a shard only knew its successor node in the circle, a key lookup would cost O(n) network hops for *n* shards.

Solution: Finger tables

- Hold the IDs of *m* shards (identifier: 0 to 2^m - 1)
- The *j*-th entry of node *i* contains the ID of successor $(i + 2^{j-1}) \bmod 2^m$
  - Where *successor* is a function that returns the closest node clockwise

S2 + 1	13	S2 + 8	38
S2 + 2	13	S2 + 16	53
S2 + 4	13	S2 + 32	53

Finger table for Shard 2

Key placement

The keys are assigned to shards based on the trivial hash function:

- Shard = successor(*key.hashCode()* **mod** 2^m)

Where *hashCode*() is the function:

- hash(s) = s[0]*31^(m - 1) + s[1]*31^(m - 2) + ... + s[m - 1]

  Where s[*i*] is the i^{th} character of the string and *m* its length

Key lookup

Lookup algorithm for key *k*:

1. Query any random shard
2. If shard *ID* == successor(*k*), then this shard is responsible for *k*
3. Else, look in the finger table for the closest successor *s* of *k*
    3.1. Of course this may wrap around in the circle
4. Forward request to *s*

Steps 2-4 may be repeated O(log *n*) times, in each contacted shard in the path to the key, n is the number of shards

Key lookup example (1/2)

Say we want to lookup for a key with hashCode = 37

1. We pick Shard 2, randomly
2. 37 mod 64 = 37 (and successor(37) != 2)
3. The closest successor in the finger table is shard 38
4. We forward the lookup request to shard 38

S2 + 1	13	S2 + 8	38
S2 + 2	13	S2 + 16	53
S2 + 4	13	S2 + 32	53

Finger table for Shard 2

Key lookup example (2/2)

Now we repeat for shard 38

1. 37 mod 64 = 37 (and successor(37) = 38)
2. Shard 38 contains the key for ID 37 and it completes the request

Reconfiguration

1. The newly added node is placed between nodes n_1 and n_2
2. The keys lying on the arc between n_1 and n_2 are split between the new node and n2
3. n_1 and n_2 are informed for this

Similarly, when a node leaves (either gracefully or by crash), a coordination service (e.g. ZooKeeper) will detect this and inform the node's predecessor and successor

Chord

- CH: https://drive.google.com/file/d/1W2ikH7vXfoa7LBUnq9GzaX6xwL8vlGbW/view?usp=sharing
- DHTs: https://drive.google.com/file/d/1t6wlczVNBBeSiktnUFurZK7-eE1EBDrX/view?usp=sharing
- Chord: https://ieeexplore.ieee.org/abstract/document/1180543
- Go: https://github.com/arriqaaq/chord
- C++: https://github.com/sit/dht
- Paper: https://pdos.csail.mit.edu/papers/ton:chord/paper-ton.pdf
- Simplified explanation: https://resources.mpi-inf.mpg.de/d5/teaching/ws03_04/p2p-data/11-18-writeup1.pdf

Task #2

Building on top of the single-node Key-Value store of task #1:

- Distribute it across a number of nodes by the way of sharding
- The sharding will be based on a hash function to map keys to their shards
  - Static sharding (shards not added once the client performs a few PUTs)
  - Dynamic sharding (shards added at any time)
- Implement a failure-detection system
  - Detect if a server has crashed

Thank you for listening!

See you in the Q&A session