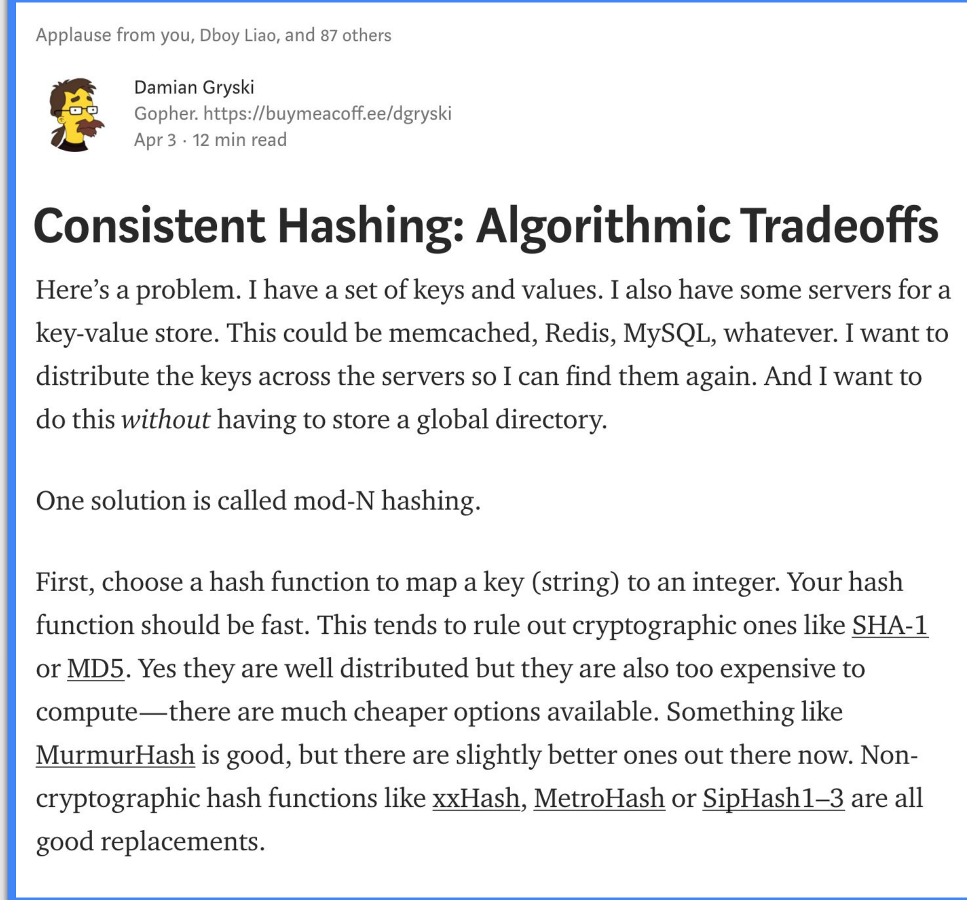
Consistent Hashing: Algorithmic Tradeoffs

Evan Lin

This slide is inspired by

[Consistent Hashing: Algorithmic Tradeoffs by Damian Gryski]



Applause from you, Dboy Liao, and 87 others

Damian Gryski
Gopher. https://buymeacoff.ee/dgryski
Apr 3 · 12 min read

Consistent Hashing: Algorithmic Tradeoffs

Here's a problem. I have a set of keys and values. I also have some servers for a key-value store. This could be memcached, Redis, MySQL, whatever. I want to distribute the keys across the servers so I can find them again. And I want to do this *without* having to store a global directory.

One solution is called mod-N hashing.

First, choose a hash function to map a key (string) to an integer. Your hash function should be fast. This tends to rule out cryptographic ones like SHA-1 or MD5. Yes they are well distributed but they are also too expensive to compute—there are much cheaper options available. Something like MurmurHash is good, but there are slightly better ones out there now. Non-cryptographic hash functions like xxHash, MetroHash or SipHash1–3 are all good replacements.

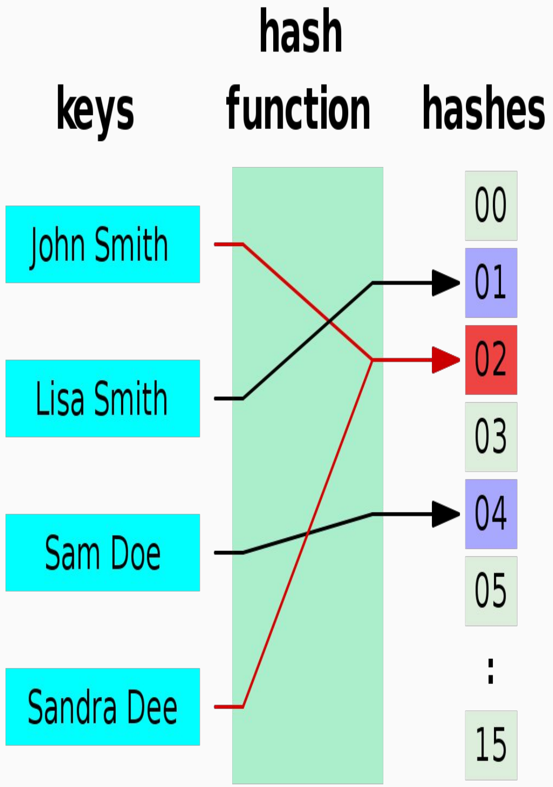
Agenda

- Original problems
- Mod-N Hashing
- Consistent Hashing
- Jump Hashing
- Maglev Hashing
- Consistent Hashing with bounded load
- Benchmarks
- Conclusion

Problems

- Some keys and values
- Need a way to store those key-value
- Distributed caching usage
- Don’t want a global directory

Mod-N Hashing



- **Pros:**
  - Quick (O(n) for read)
  - Easy to implement
- **Cons:**
  - Hard not add/remove value might need reconstruct whole hashing table

```
server := serverList[hash(key) % N]
```

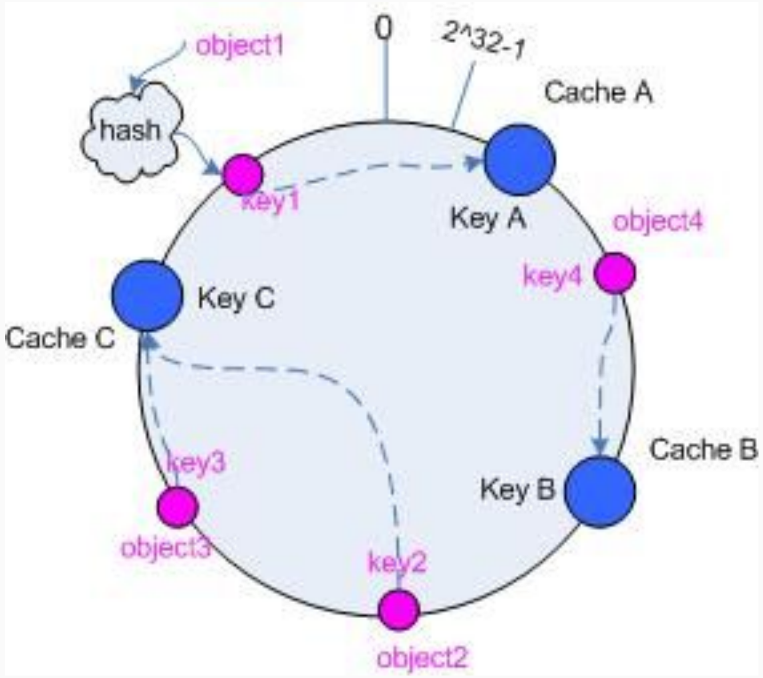
Mod-N Hashing: Problems

- Hard to add new server:
    a. e.g. N = 9 → N = 10
- Also, hard to remove original server
    a. e.g. N= 9 → N = 8
- We need a better way to add/remove servers and make it change as less as possible..

```
server := serverList[hash(key) % N]
```

Consistent Hashing

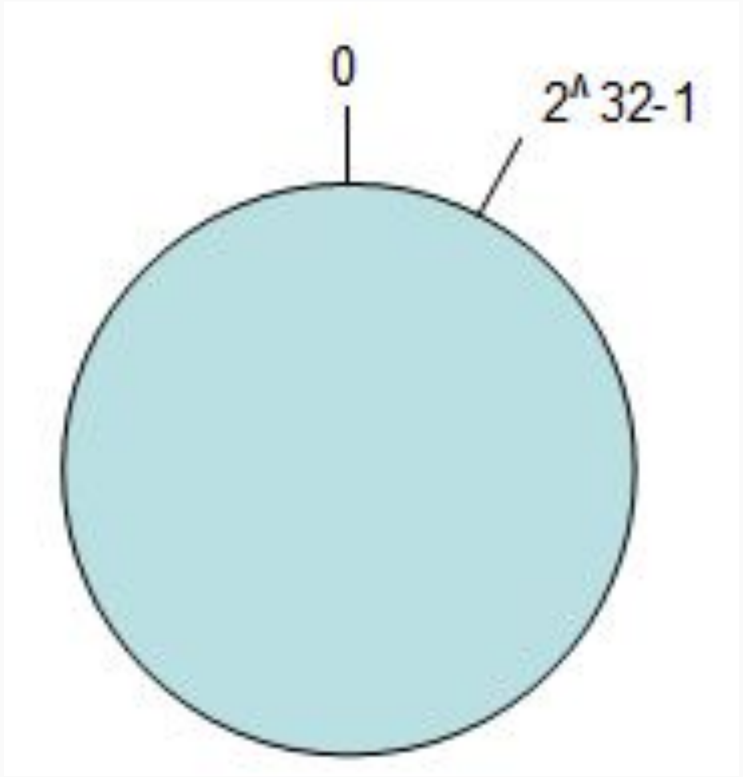
- A ring hashing table
- “N” not represent servers, it represent all hasing table slots.
- Two hash functions: one for data another one for servers
- Support add/remove servers and only impact necessary servers



Paper: Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web

Consistent Hashing Introduce (1)

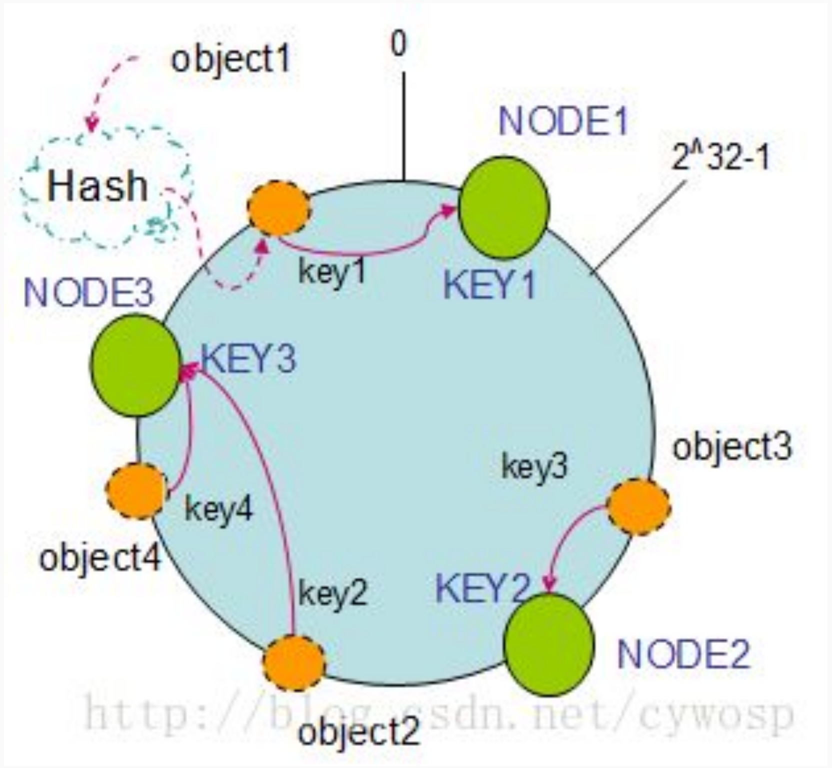
A ring hashing table



Ref: 每天进步一点点——五分钟理解一致性哈希算法(consistent hashing)

Consistent Hashing Introduce (2)

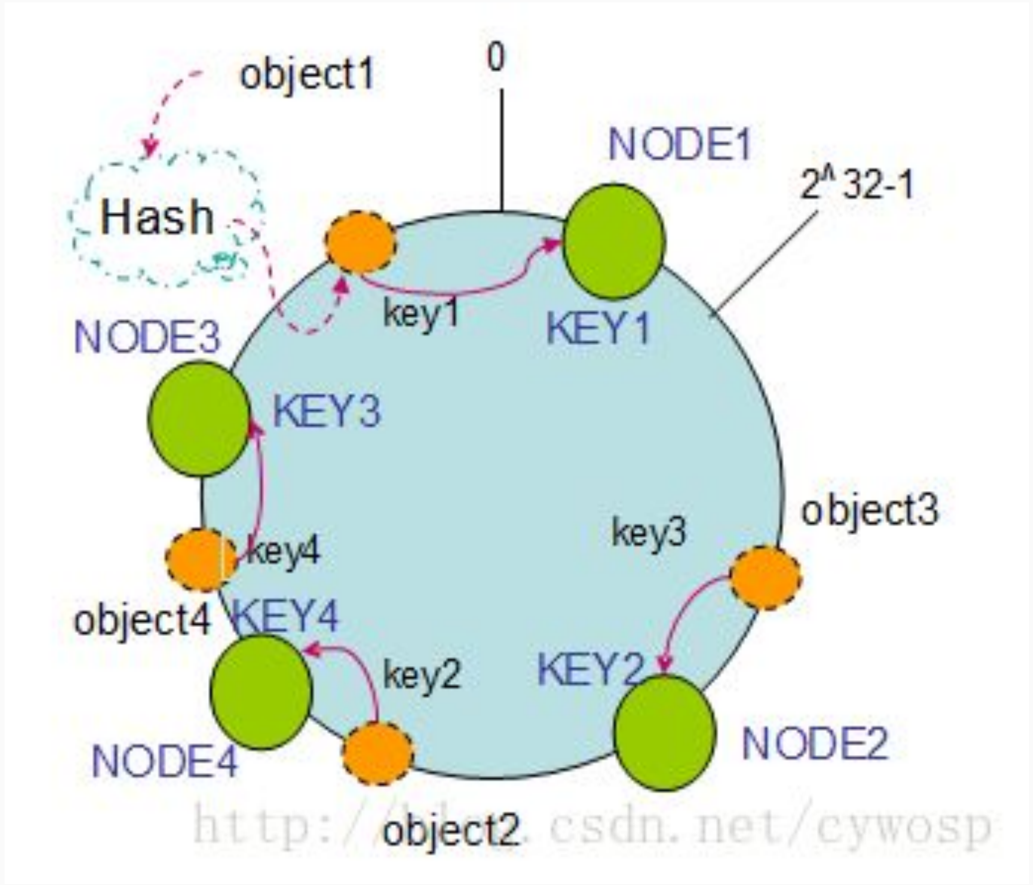
- Add servers in another hash function into hashing table
- Mapping data into servers clockwise



Ref: 每天进步一点点——五分钟理解一致性哈希算法(consistent hashing)

Consistent Hashing Introduce (3)

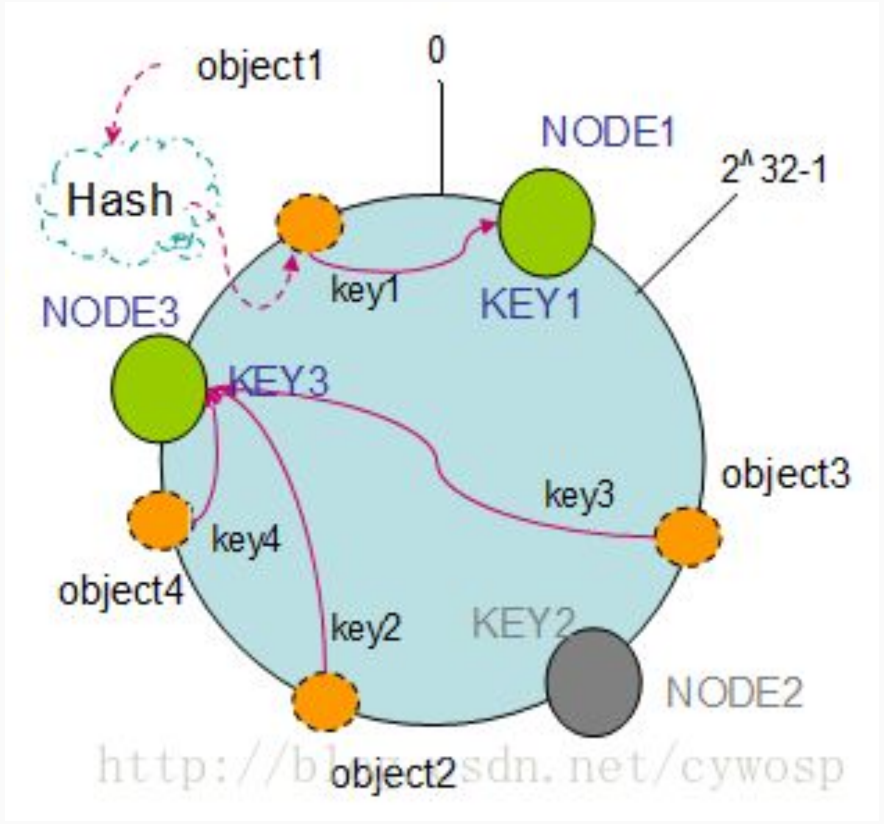
Add servers



Ref: 每天进步一点点——五分钟理解一致性哈希算法(consistent hashing)

Consistent Hashing Introduce (4)

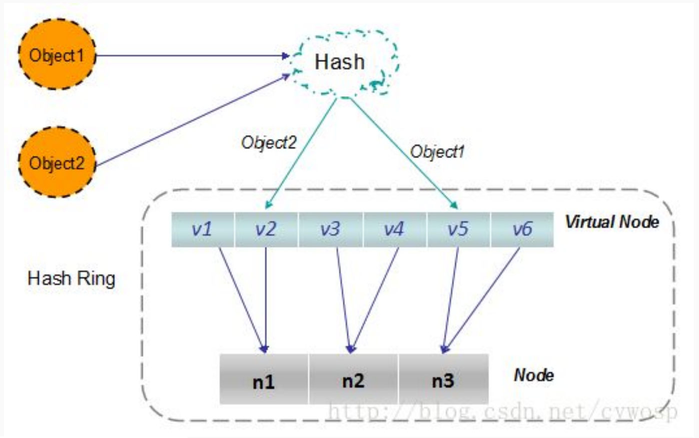
Delete servers

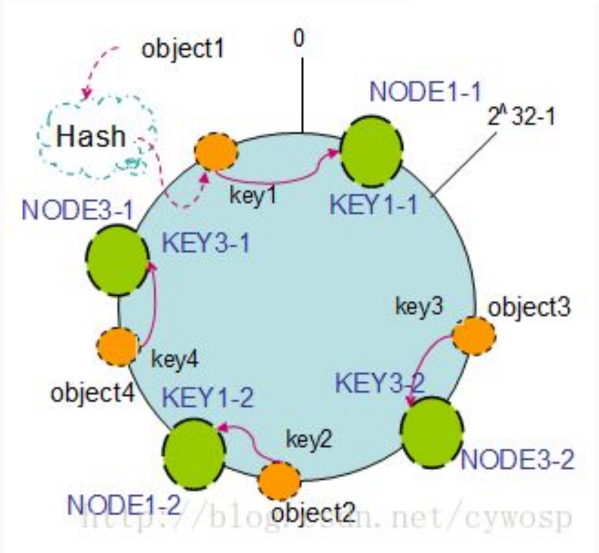


Ref: 每天进步一点点——五分钟理解一致性哈希算法(consistent hashing)

Consistent Hashing Introduce (5)

- Virtual Nodes
  - Make more server replica address in hashing talbe
  - Usually use “server1-1”, “server1-2” …
  - More virtual nodes means load balance but waste address
- This reduces the load variance among servers





Ref: 每天进步一点点——五分钟理解一致性哈希算法(consistent hashing)

Examples of use

- Data partitioning in Apache Cassandra
- GlusterFS, a network-attached storage file system
- Maglev: A Fast and Reliable Software Network Load Balancer
- Partitioning component of Amazon's storage system Dynamo

GlusterFS

cassandra

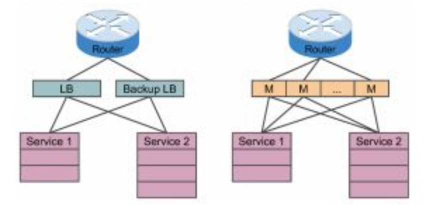


Figure 1: Hardware load balancer and Maglev.

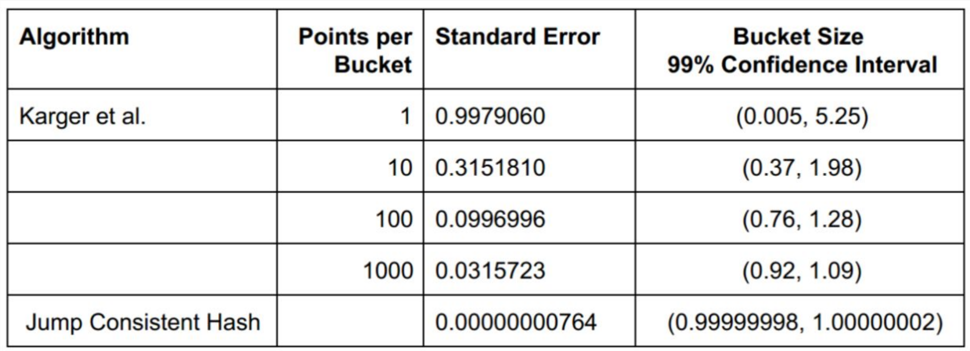
Ref: Wiki

Are we done?

- Load Distribution across the nodes can still be uneven
  - With 100 replicas (“vnodes”) per server, the standard deviation of load is about 10%.
  - The 99% confidence interval for bucket sizes is 0.76 to 1.28 of the average load (i.e., total keys / number of servers).
- Space Cost
  - For 1000 nodes, this is 4MB of data, with O(log n) searches (for n=1e6) all of which are processor cache misses even with nothing else competing for the cache.

Jump Hashing

- Paper publish by Google
- Better load distribution:
- Great space usage:
  - Consistent Hashing: *O(n)*
  - Jump Hashing: *O(1)*
- Good performance:
  - Time complexity: *O(log n)*
- Easy to implement...



Algorithm	**Points per Bucket**	**Standard Error**	**Bucket Size 99% Confidence Interval**
Karger et al.	1	0.9979060	(0.005, 5.25)
	10	0.3151810	(0.37, 1.98)
	100	0.0996996	(0.76, 1.28)
	1000	0.0315723	(0.92, 1.09)
Jump Consistent Hash		0.00000000764	(0.99999998, 1.00000002)

Paper: A Fast, Minimal Memory, Consistent Hash Algorithm

Jump Hashing implementation code

```
func Hash(key uint64, numBuckets int) int32 {
  var b int64 = -1
  var j int64
  for j < int64(numBuckets) {
    b = j
    key = key*2862933555777941757 + 1
    j = int64(float64(b+1) *
      (float64(int64(1)<<31) / float64((key>>33)+1)))
  }
  return int32(b)
}
```

magic number

The time complexity of the algorithm is determined by the number of iterations of the while loop. [...] So the expected number of iterations is less than ln(n) + 1.

Ref: Consistent Hashing: Algorithmic Tradeoffs Google 提出的 Jump Consistent Hash

Limitation of Jump Hashing

- Not support server name, only provide you the “shard number”.
- Don’t support arbitrary node removal

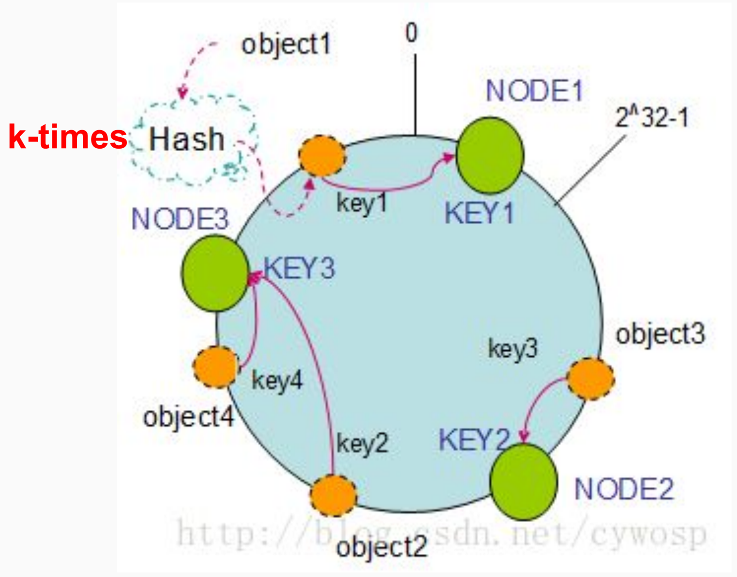
```
func Hash(key uint64, numBuckets int) int32 {
  var b int64 = -1
  var j int64
  for j < int64(numBuckets) {
    b = j
    key = key*2862933555777941757 + 1
    j = int64(float64(b+1) *
      (float64(int64(1)<<31) / float64((key>>33)+1)))
  }
  return int32(b)
}
```

Why Is This Still a Research Topic?

- Ring Hashing:
  - Arbitrary bucket addition and removal
  - High memory usage
- Jump Hashing:
  - Eeffectively perfect load splitting
  - Reduced flexibility

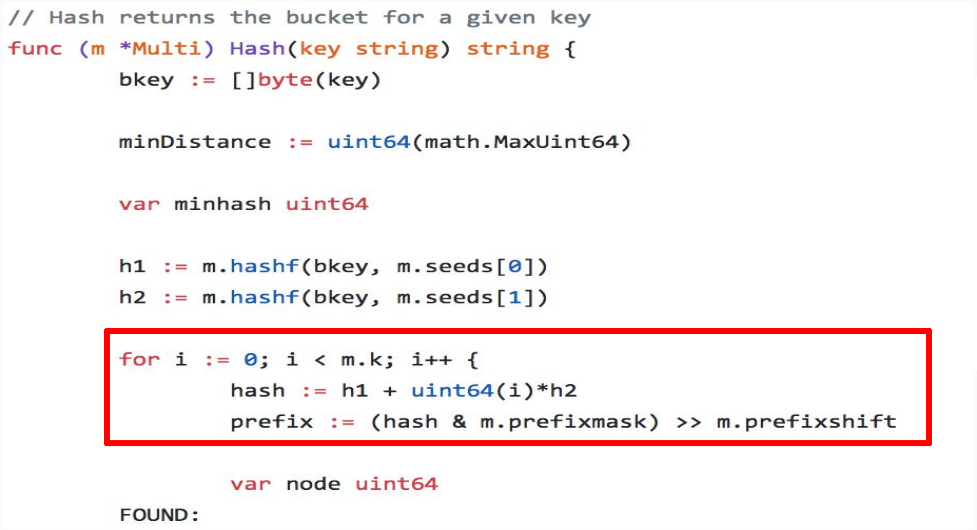
Multiple-probe consistent hashing

- Equal with consistent hashing
  - *O(n)* space (one entry per node),
  - *O(1)* addition and removal of nodes.
- Add *k-times hashing* when you try to get key. → look up will slower than consistent hashing
- The *k* value determine the peak-to-average



Paper: [Multi-probe consistent hashing]()

Multiple-probe consistent hashing (cont.)



```
// Hash returns the bucket for a given key
func (m *Multi) Hash(key string) string {
        bkey := []byte(key)

        minDistance := uint64(math.MaxUint64)

        var minhash uint64

        h1 := m.hashf(bkey, m.seeds[0])
        h2 := m.hashf(bkey, m.seeds[1])

        for i := 0; i < m.k; i++ {
                hash := h1 + uint64(i)*h2
                prefix := (hash & m.prefixmask) >> m.prefixshift

                var node uint64
        FOUND:
```

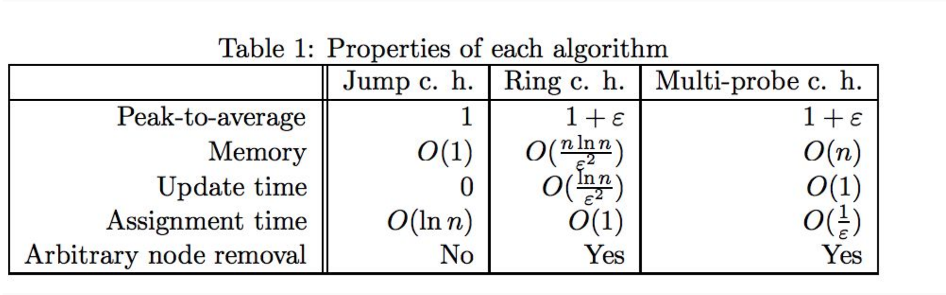


Table 1: Properties of each algorithm

	Jump c. h.	Ring c. h.	Multi-probe c. h.
Peak-to-average	1	$1+\varepsilon$	$1+\varepsilon$
Memory	$O(1)$	$O(\frac{n \ln n}{\varepsilon^2})$	$O(n)$
Update time	0	$O(\frac{\ln n}{\varepsilon^2})$	$O(1)$
Assignment time	$O(\ln n)$	$O(1)$	$O(\frac{1}{\varepsilon})$
Arbitrary node removal	No	Yes	Yes

Another approach: Rendezvous Hashing

- Hash the node and the key together and use the node that provides the highest hash value.
- Lookup cost will raise to *O(n)*
- Because the inner loop doesn't cost a lot, so if the number of node is not big we could consider using this hashing.

```
func (r *Rendezvous) Lookup(k string) string {
        khash := r.hash(k)

        var midx int
        var mhash = xorshiftMult64(khash ^ r.nhash[0])

        for i, nhash := range r.nhash[1:] {
                if h := xorshiftMult64(khash ^ nhash); h > mhash {
                        midx = i + 1
                        mhash = h
                }
        }

        return r.nstr[midx]
}
```

Paper: Rendezvous Hashing

Google:
Maglev Hashing

- Google software load balancer publish in 2016
- One of the primary goals was lookup speed and low memory usage as compared with ring hashing or rendezvous hashing. The algorithm effectively produces a lookup table that allows finding a node in constant time.

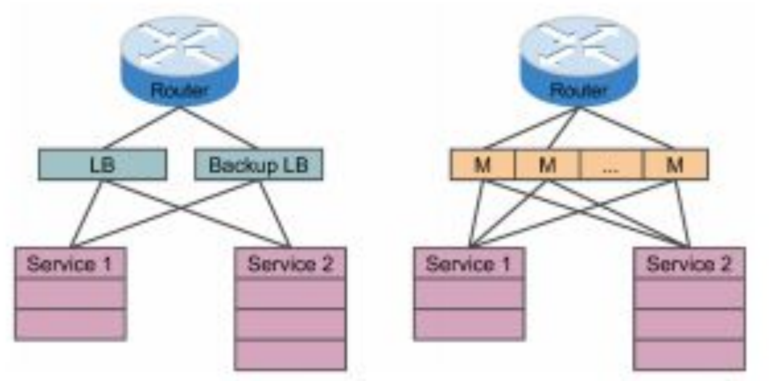


Figure 1: Hardware load balancer and Maglev.

Paper: Maglev: A Fast and Reliable Software Network Load Balancer

Google:
Maglev Hashing (downsides)

- Generating a new table on node failure is slow (the paper assumes backend failure is rare)
- This also effectively limits the maximum number of backend nodes.

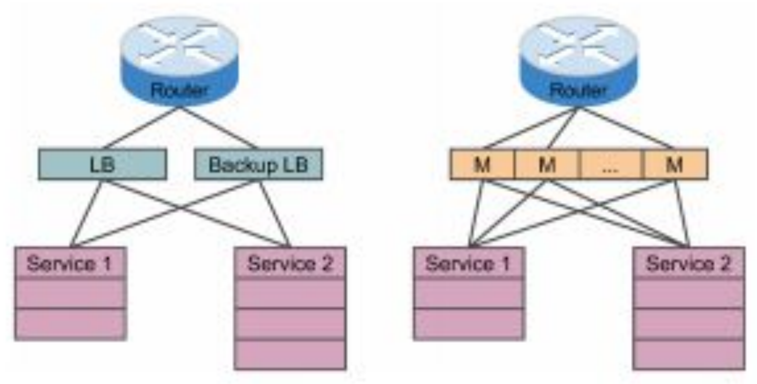


Figure 1: Hardware load balancer and Maglev.

Paper: Maglev: A Fast and Reliable Software Network Load Balancer

Google:
Maglev Hashing

	Consistent Hashing	***Maglev Hashing***
Preparing	● Put each server in hashing ring ● Add virtual node for load distribution	● Prepare Permutation Table ● Generate lookup table
Loopup	Hash value and lookup server	Using hashing value to lookup table to find server

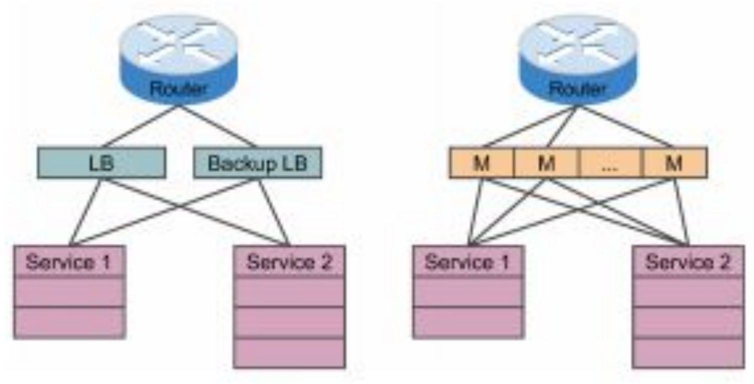
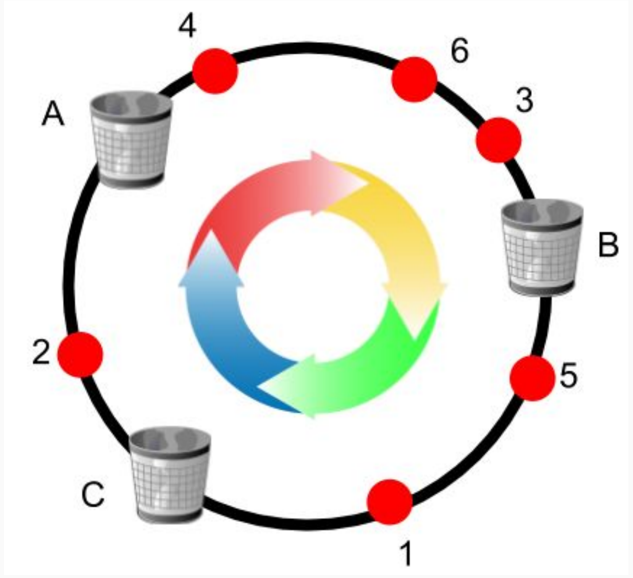


Figure 1: Hardware load balancer and Maglev.

Paper: [Maglev: A Fast and Reliable Software Network Load Balancer]

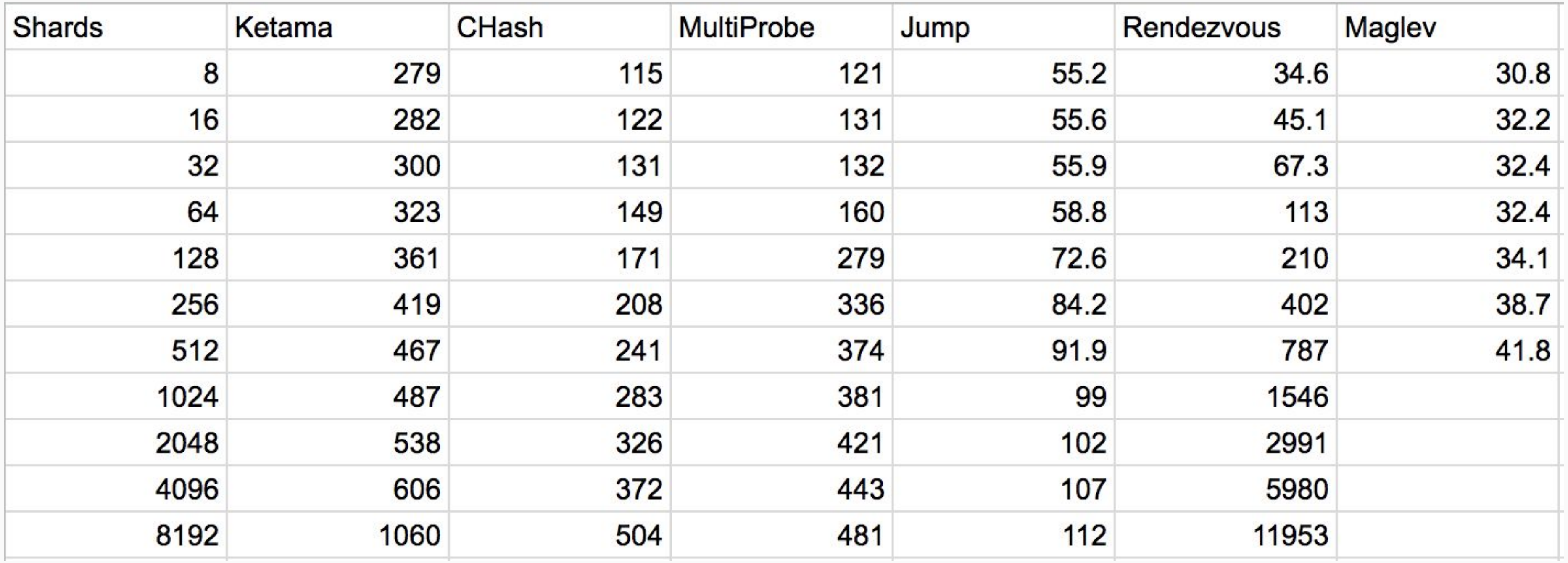
Consistent Hashing with bounded load

- Papaer publish by Google in 2016, already use in Google pubsub service for long time.
- Using consistent hashing as load balance
- Using bound load to check if using such server.
- Vimeo implement this in HAProxy and post in this blog commits



Paper: Consistent Hashing with Bounded Loads

Benchmarking



Shards	Ketama	CHash	MultiProbe	Jump	Rendezvous	Maglev
8	279	115	121	55.2	34.6	30.8
16	282	122	131	55.6	45.1	32.2
32	300	131	132	55.9	67.3	32.4
64	323	149	160	58.8	113	32.4
128	361	171	279	72.6	210	34.1
256	419	208	336	84.2	402	38.7
512	467	241	374	91.9	787	41.8
1024	487	283	381	99	1546	
2048	538	326	421	102	2991	
4096	606	372	443	107	5980	
8192	1060	504	481	112	11953	

Conclusion

- No perfect consistent hashing algorithm
- They have their tradeoffs
  - balance distribution,
  - memory usage,
  - lookup time
  - construction time (including node addition and removal cost).

Q&A