COMS 4995 Final Report

Linxiao Wu
lw2944@columbia.edu

Zixuan Zhang
zz2777@columbia.edu

Joseph Yang
zy2431@columbia.edu

Haoran Zhu
hz2712@columbia.edu

***Abstract*—For the final project, we have chosen the reading-based option. In this project, we dive into the topic of Consistent Hashing (CH) in depth. The paper begins with an introduction and a problem set up, which leads to early appearance of Consistent Hashing. Then we will separately talk about several key papers in this topic. And in the end, we will compare across these different Consistent Hashing algorithms and find the similarity/difference among them.**

***Index Terms*—Consistent Hashing, Rendezvous Hashing, Jump Hashing, Anchor Hash, Maglev Hashing, Load Balancing**

I. Introduction

Consistent Hashing was first addressed in Karger et al [1] in 1997. Since then, many different Consistent Hashing algorithms have been given for different specific needs in different use cases. We have chosen *Consistent Hashing and Random Trees: DistributedCaching Protocols for Relieving Hot Spots on the World Wide Web* [1], *A Name-Based Mapping Schemefor Rendezvous* [2], *AnchorHash: A Scalable Consistent Hash* [3], *A Fast, Minimal Memory, ConsistentHash Algorithm* [5], and *Maglev: A Fast and ReliableSoftware Network Load Balancer* [4] for our project. We will first dive in to the problem setup and historical origin of Consistent Hashing, and then discuss these papers one by one, and finally compare them in general.

II. First Appearance of Consistent Hashing

A. Problem Set-up

To begin with, we can recall the definition of Hashing:

Definition 1. Hashing:

Hashing is the process of converting a given key into another smaller value in a small and nearly constant time per retrieval.

This is done by taking the help of some function or algorithm which is called as hash function to map data to some encrypted or simplified representative value which is termed as "hash code" or "hash". This hash is then used as an index to narrow down search criteria to get data quickly.

Therefore, here is a very classic application of hashing: We have a set of keys and values as shown by Figure 18. We also have some servers for a key-value store. We want to distribute the keys across the servers so that we can find them again (retrieve).

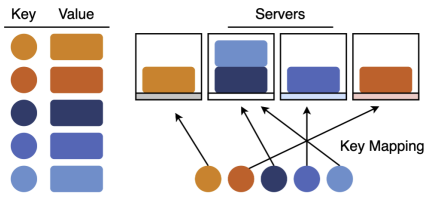

Fig. 1. Keys Stored in Servers [image source]

A very intuitive but naive solution to the above question may be the naive modulo-N solution [1]

$$server := serverList[hash(key)\%N] \tag{1}$$

Algorithm 1 is self-explanatory. If we think of the server as bins, and the keys are balls (we will keep consistent with this metaphor for the later section), it is just to use the modulo algorithms after the hashing algorithm to find the bin using the remainder. Potential hashing algorithms include SHA256 and MD5, or some other simpler ones.

However, there is a huge drawback of Algorithm 1: **what if the number of servers change? For example, servers are added (new one) or removed (server down)?** If we still use the Algorithm 1, almost all the keys need to be rehashed in order to retrieve the values. It is very unrealistic especially for the industries where we may have tens of thousands of servers around the globe, where server addition and removal is a common practice.

We define a hash family to have the $MinimalDisruption$ property:

Definition 2. Minimal Disruption

1) When adding or removing servers, only 1/nth of the keys should move
2) Don't move any keys that don't need to move.

We can be more specific with the above properties: For the first point: if we're moving from 9 servers to 10, then the new server should be filled with 1/10th of all the keys. And those keys should be evenly chosen from the 9 "old" servers. For the second point: the keys should only move to the new server, never between two old servers (if we use Algorithm 1, there will be movements between old servers). Similarly, if we need to remove a server (say, because it crashed), then

the keys should be evenly distributed across the remaining live servers.

This is the foundation of Consistent Hashing as a standard scaling technique.

B. Rendezvous Hashing (1996)

Rendezvous hashing is both more general than consistent hashing, and has a conceptually simpler algorithm. Rendezvous hashing may hence be preferable to consistent hashing in many applications.

Historically, rendezvous hashing [2] was first described in 1996, while the name of consistent hashing first appeared in 1997, and uses a different algorithm [1].

Rendezvous hashing is an algorithm to solve the distributed hash table problem - a common and general pattern in distributed systems to solve the problem shown in Figure 18. Given a key and a dynamic list of servers, the task is to map keys to servers while preserving:

- **Load Balancing**: Each server is responsible for (approximately) the same number of loads.
- **Scalability**: We can add and remove servers without too much computational effort.
- **Lookup Speed**: Given a key, we can quickly identify the correct server.

To overcome the drawback of Algorithm 1, rather than pick a single server, Rendezvous hashing makes each key generate a randomly sorted list of servers and chooses the first server from the list as shown in Figure 2.

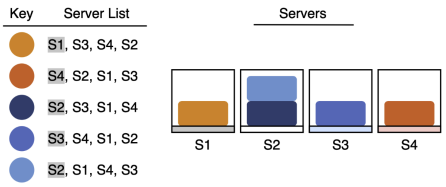

Fig. 2. Rendezvous Hashing Sorted Server List[image source]

If the first choice is removed, the algorithm simply move the key to the second server in the list. It is easy to see that we only need to move the keys that were previously managed by the server that went down shown in Figure 3.

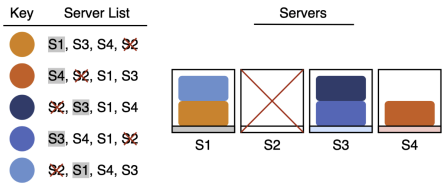

Fig. 3. Rendezvous Hashing: Server Down[image source]

A variant of Rendezvous hashing is to introduce weights. In some situations, we want to do biased load balancing rather than uniform random key assignment. For example, some servers might have larger capacity and should therefore be selected more often. Rendezvous hashing accommodates weighted servers in a very elegant way. Instead of sorting the servers based on their hash values, we rank them based on $-\frac{w_i}{\ln h_i(x)}$, where x is the key, w_i is the weight associated with server i, and $h_i(x)$ is the hash value. Therefore, Rendezvous hashing is also called highest random weight (HRW) hashing.

Rendezvous hashing gives excellent bounds and guarantees. According to Thaler et all [2], the server disruption bound achives its optimum of $1/N$ whenever a server is added or removed.

A concern is that the query time for Rendezvous hashing is $O(N)$, if we have N servers, because we have to examine all of the key-server combinations. Consistent hashing is $O(\log N)$ [1] and can be much faster when N is large enough.

C. Consistent Hashing (1997)

The name of ”Consistent Hashing” was brought up by Karger et al [1] in 1997, one year after Rendezvous hashing. This algorithm is the popular ring-based consistent hashing. We may have seen a “points-on-the-circle” diagram. Consistent Hashing is a distributed hashing scheme that operates independently of the number of servers or objects in a distributed hash table by assigning them a position on an abstract circle, or hash ring.

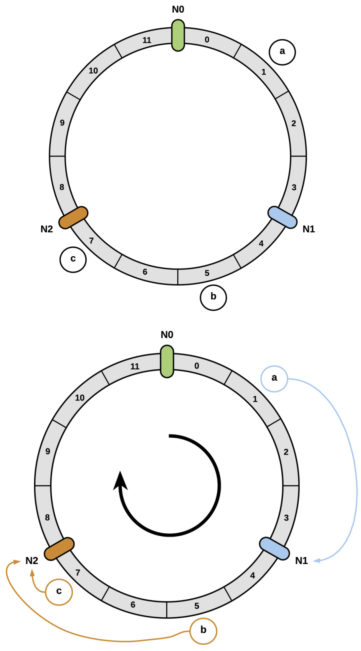

Fig. 4. Consistent Hashing: Hashing both Servers and Keys and Assigning Clockwise [image source]

The main idea is to use a hash function to randomly map both the objects and the servers to a unit circle, e.g. at an angle of $hash(o)$ (mod 360) as shown in Figure 4, where we first

hash the servers (N) evenly around the circle, and then hash the keys (objects), and then each object is then mapped to the next server that appears on the circle in clockwise order.

It is intuitive that if we have M keys and N servers, each server is expected to have m/n keys, and this is also the number of keys we expect to move when a server is added or removed [3].

However, there is a big problem of this simple consistent hashing ring as above. The maximum load is likely to be $\Theta(\log n)$ bigger than the average [3]. This has to do with a big variation in the coverage of the bin. To get a quick sense, Figure 5 shows the situation when we define another node N3 (node addition). There is no space for the overall distribution to be uniform anymore and we may also not reorganize the nodes, otherwise we would not be consistent anymore. What's more We must change its association and get a be associated to N3.

Although the above algorithm gets a good property of minimal disruption, where only $1/N = 1/3$ portion of the keys (0 to 3) needs to be remapped, but the server coverage is not uniform as shown in Figure 5.

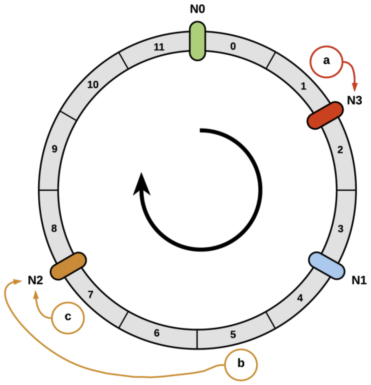

Fig. 5. Non-uniform Coverage Example[image source]

Formally speaking, we expect on the average, each server covers and interval of size $1/N$, but we expect some servers to cover intervals of size $\Theta(\frac{\log N}{N})$, and such bins are expected to get $\Theta(\frac{M \log N}{N})$ keys [1] [3].

To get a more uniform coverage, Karger et. al. [3] suggests the use of **virtual nodes**. The trick is that the key contents of $d = O(\log N)$ servers is unitedin a single super-server. The d servers making up a super server are called virtual nodes. We now have only $N' = N/d$ super servers and these super servers represent the real servers. A real server covers the intervals covered by its d virtual nodes. Karger et. al. [3] proves that with any constant $\epsilon > 0$, if we pick a large enough $d = O(\log N)$, then with high probability that each real server covers a fraction of $(1 \pm \epsilon)N'$ of the unit circle. The virtual node solution to Figure 5 is shown as Figure 14.

The following Table I shows the comparison of bounds between classic hash table and consistent hashing as described

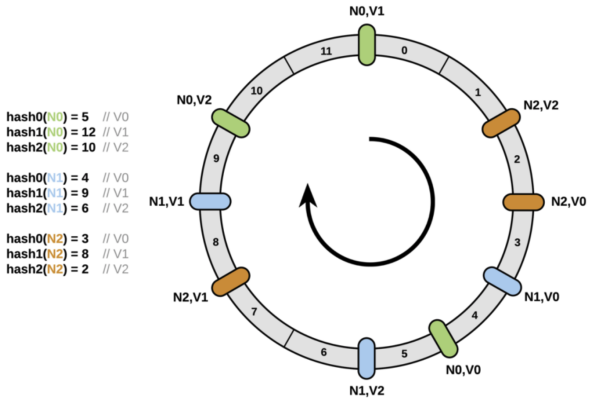

Fig. 6. Virtual Nodes Example[image source]

above. We will not provide a detailed proof here because it is not the math focus of this paper.

	Classic hash table	Consistent hashing
add a node	$O(K)$	$O(K/N + \log N)$
remove a node	$O(K)$	$O(K/N + \log N)$
add a key	$O(1)$	$O(\log N)$
remove a key	$O(1)$	$O(\log N)$

TABLE I
COMPARISON OF BOUNDS BETWEEN CLASSIC HASHING AND CONSISTENT HASHING FOR **K** KEYS

Note that The $O(K/N)$ is an average cost for redistribution of keys and the $O(\log N)$ complexity for consistent hashing comes from the fact that a binary search among nodes angles is required to find the next node on the ring.

D. Consistent Hashing with Bounded Loads

Classic Consistent Hashing introduced by Karget et. al. [1] maintains good Minimal Disruption property, but a problem raised by Mirrokni et. al. [3] is the **hard capacity constraint**: although we have expected $O(M/N)$ keys in each server, there is still possibility that many servers can be overloaded, and Mirrokni et. al. [3] gives an expected overload of $\Theta(\log n / \log \log n)$.

Thus, the main contribution of Mirrokni et. al. [3] is for M keys, N servers and a given user-specified balanced $c = 1 + \epsilon > 1$, they find a hashing scheme with no load above $\lceil cM/N \rceil$, referred to as the capacity of the servers.

In order to deal with the capacity constraint, they apply the idea of linear probing by forwarding the key on the circle to the first non-full server.

Their algorithm can be summarized as the follows (we refer to servers as "bins", and keys as "balls"):

- **General Insertion of a ball**: The newly inserted ball is first placed into the bin it is hashed to. If the bin becomes overfull (load exceeds the capacity), we forward any of the balls in the bin to the next bin.

- **Removing a ball**: Deleting a ball creates a new hole in the corresponding bin. Starting from the first hole in some bin b, we scan bins one by one. When we get to a bin b', if b' contains a ball q' which was hashed to or before b, we fill b with q'. Then we proceed from b'.
- **Removing a bin**: Removing a bin is the same as setting the capacity of the bin to 0, which makes the bin overfull. The we forward balls from the overfull bin until no bin is overfull.
- **Adding a bin**: When a bin is first created, it has the number of holes equal to its capacity. We can then fill the holes following the procedure in removing a ball.
- **Changing capacities for load balancing**: Let the lowest $\lceil cm \rceil - n\lfloor cm/n \rfloor$ bins have capacity $\lceil cm/n \lceil$ while the rest have capacity $\lfloor cm/n \rfloor$. Also, if $cm < n$, then we force all bins to have capacity 1.

They show that with this scheme, with a server added or removed, the expected number of keys that have to be moved is

1) $O(\frac{1}{\epsilon^2})$ of the optimum $(O(\frac{M}{N\epsilon^2}))$, if $\epsilon \leq 1$
2) $O(1 + \frac{\log c}{c})$ of the optimum $(O(1 + \frac{\log c}{c} \cdot \frac{M}{N}))$ if $\epsilon > 1$

As we get an overview of Consistent Hashing, we will list several concrete implementations and the proof of guarantees of consistent hashing in the following sections.

III. Jump Hashing

A. PRELIMINARIES

From the previous section, we saw that Karger et al. introduced the concept of consistent hashing (CH) and gave an algorithm to implement it. Initialliy, CH specifies a distribution of data among servers in such a way that servers can be added or removed without having to totally reorganize the data. It was originally proposed for web caching on the Internet, in order to address the problem that clients may not be aware of the entire set of cache servers.

Since then, CH has also seen wide use in data storage applications. In the paper "A Fast, Minimal Memory, Consistent Hash Algorithm" (arXiv:1406.2294 [cs.DS]) [5], the authors addressed the problem of splitting data into a set of shards, where each shard is typically managed by a single server (or a small set of replicas). As the total amount of data changes, increase or decrease of the number of shards is needed. This requires moving data in order to split the data evenly among the new set of shards, and the purpose of the JumpHash algorithm presented in the paper is to move as little data as possible while doing so.

B. JUMP HASH

Here's a complete implementation of the jump consistent hash in C++.

```
int32_t JumpConsistentHash(uint64_t key, int32_t num_buckets) {
  int64_t b = -1, j = 0;
  while (j < num_buckets) {
    b = j;
    key = key * 2862933555777941757ULL + 1;
    j = (b + 1) * (double(1LL << 31) / double((key >> 33) + 1));
  }
  return b;
}
```

Fig. 7. The complete Jump Hash C++ implementation [image source]

The basic idea behind the Jump consistent hash works by computing when its output changes as the number of buckets increases. Let $ch(key, num(buckets))$ be the consistent hash for the key when there are $num(buckets)$ buckets. Clearly, for any key k, $ch(k, 1) = 0$, since there is only the one bucket. In order for the consistent hash function to be balanced, $ch(k, 2)$ will have to stay at 0 for half the keys, while it will have to jump to 1 for the other half. In general, $ch(k, n + 1)$ has to stay the same as $ch(k, n)$ for $\frac{n}{n+1}$ fraction of all the keys, and jump to n for the other 1/(n+1) fraction.

From the above discussion, we see that a linear time algorithm can be defined by using the formula for the probability of $ch(key, j)$ jumping when j increases. Here's an implementation in C++.

```
int ch(int key, int num_buckets) {
    random.seed(key);
    int b = 0;  // This will track ch(key, j+1).
    for (int j = 1; j < num_buckets; j++) {
        if (random.next() < 1.0 / (j + 1)) b = j;
    }
    return b;
}
```

Fig. 8. The 1^{st} naive implementation [image source]

This linear algorithm works as follows: Given a key and number of buckets, the algorithm

1) Considers each successive bucket i, for $i = 1, 2, \ldots, number(buckets) - 1$.
2) Computes $ch(key, i + 1)$ using $ch(key, i)$.
3) At bucket $i + 1$, decides whether to keep $ch(k, i + 1) = ch(k, i)$, or to jump its value to i.
4) Uses a pseudo-random number generator with the key as its seed in order to jump for the right fraction of keys.
5) Generates a uniform random number in the range $[0.0, 1.0]$ and jumps if the value is less than $\frac{1}{i+1}$, when it jumps for $\frac{1}{i+1}$ fraction of the keys.

Furthermore, practically, $ch(key, i+1)$ is usually unchanged as i increases, only jumping occasionally. We may exploit this practical observation and convert the above **linear** algorithm to **logarithmic** time. The revised algorithm will only compute the destination ofs of jumps (i.e. the i's where $ch(key, i+1) \neq ch(key, i)$). For these i's, $ch(key, i + 1) = i$.

Suppose that the algorithm is tracking the bucket numbers of the jumps for a particular key, k. And suppose that b was the destination of the last jump, that is, $ch(k,b) \neq ch(k,b+1)$, and $ch(k,b+1) = b$. Now, we want to find the next jump, the smallest i such that $ch(k,i+1) \neq ch(k,b+1)$, or equivalently, the largest i such that $ch(k,i) = ch(k,b+1)$. Let X be the random variable of the value of such i. Then we notice that for any bucket number j, we would have $X \geq j$ if and only if the consistent hash value hasn't changed by j, or equivalently, $ch(k,j) = ch(k,b+1)$. We have reached

$$P(X) = P(ch(k,j) = ch(k,b+1)) \tag{2}$$

In general, if $n \geq m$, $P(ch(k,n) = ch(k,m)) = \frac{m}{n}$, so for any $j > b$,

$$P(X) = P(ch(k,j) = ch(k,b+1)) = \frac{b+1}{j} \tag{3}$$

Now we generate a pseudo-random variable r that is uniformly distributed in the range $[0.0, 1.0]$. Since we want $P(X \geq j) = \frac{b+1}{j}$, we set $P(X \geq j)$ if and only if $r \leq \frac{b+1}{j}$. The condition is equivalent to if and only if $j \leq \frac{b+1}{r}$. From previous definition of i and the definition of the floor function, we would have $i = floor(\frac{b+1}{r})$. And a C++ implementation of this revised **logarithmic** time algorithm would be

```
int ch(int key, int num_buckets) {
    random.seed(key);
    int b = -1;  // bucket number before the previous jump
    int j = 0; // bucket number before the current jump
    while (j < num_buckets) {
        b = j;
        r = random.next();
        j = floor((b + 1) / r);
    }
    return = b;
}
```

Fig. 9. The 2^{nd} naive implementation [image source]

The final step to the actual code given at the beginning is to implement random. It has to be fast and yet to also to have well distributed successive values. So 64-bit linear congruential generator is chosen.

C. PERFORMANCE ANALYSIS

First, compare this algorithm with Karger et al.'s algorithm in terms of **distributing the keys uniformly among buckets**. Ideally, all buckets should receive the same fraction of hash values. Notice that in the Karger et al.'s algorithm, this depends on the number of points chosen per bucket. Here's a result diagram.

Then compare them in terms of **space** requirements. Distributing the keys uniformly among buckets requires using many points per bucket in Karger et al.'s algorithm, but this increases memory requirements significantly. In the experiment,

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)

Fig. 10. Comparison of bucket size confidence [image source]

two variations of Karger et al.'s algorithm were implemented in addition to jump consistent hash. All implementations are in C++ and use the Standard Template Library. They were compiled on a 64-bit platform using Gnu C++ and measured on an Intel Xeon E5-1650 CPU with 32GB of memory.

The first implementation of Karger et al.'s algorithm ("version A") represents the point data as an STL map from a 64-bit hash value to a 32-bit bucket number. This is probably easiest and most natural way to implement the algorithm. Internally the map is represented as a balanced binary tree. This implementation uses 48 bytes per point per bucket.

The second implementation ("version B") represents the point data as a sorted vector of (hash value, bucket number) pairs, where the hash values are truncated to 32 bits to save space. The bucket corresponding to a given hash value is located using binary search. This implementation uses less space (8 bytes per point per bucket), but unlike the previous implementation, it does not support dynamic updates efficiently: in order to change the number of buckets, the entire data structure must be rebuilt.

And the following table presents the total data size for various number of buckets, assuming that 1k points per bucket are used

Number of Buckets	Space (Karger, Version A)	Space (Karger, Version B)
10	469 KB	78 KB
1000	46 MB	7.6 MB
100000	4.5 GB	0.75 GB

Fig. 11. Comparison of space storage used [image source]

Finally compare them by **execution time**. Below is the chart of execution time comparison in exponential number of buckets.

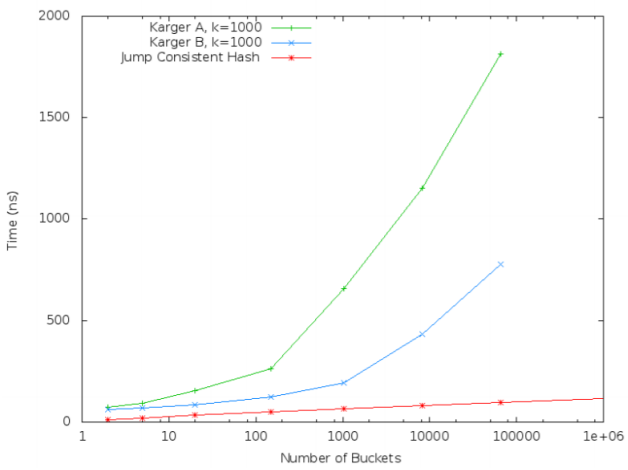

Fig. 12. Comparison of execution time [image source]

IV. AnchorHash

A. PRELIMINARIES

Consistent hashing(CH) is essential to many distributed software, like load balancing or web caching. The original design objective is to have a hash algorithm that minimizes rehash when buckets are added or removed. However, a good CH algorithm is now expected to have good time and space complexity, as the internet traffic has exploded since its invention.

AnchorHash [3] is another attempt to ensure full consistency under arbitrary changes with low memory footprint and fast lookup. Comparing to popular algorithms like Jump Hash, it does not rely on strict order in which buckets are added or dropped. Comparing to Maglev hash, another popular implementation, AnchorHash has much lower space complexity as it does not need to store large number of key-value pairs.

As a consistent algorithm, AnchorHash must meet two critical objectives: minimal disruption and balance. Define the set of all possible buckets $\mathcal{A}$. At any given moment, only some buckets $\mathcal{W} \subset \mathcal{A}$ are used in the system. Denote the set of buckets not in use as $\mathcal{A} \backslash \mathcal{W}$

Definition 3. *Minimal Disruption*

- Upon addition of a new bucket $b \in \mathcal{A} \backslash \mathcal{W}$ to $\mathcal{W}$, keys either maintain their mapping or are remapped to b
- Upon removal of an existing bucket $b \in \mathcal{W}$, only keys that were mapped to b are remapped. Other keys remain consistent.

Definition 4. *Balance*

Let $k \in \mathcal{U}$ be a key. A hash algorithm achieves balance iff k has equal probabilities of being mapped to any other bucket $b \in \mathcal{W}$

Definition 5. *Consistency*

An algorithm is consistent iff it achieves both *minimal disruption* and *balance*

B. ANCHOR HASH

Here's the heuristic of Anchorhash and why it achieves *consistency* when buckets are removed and added.

Bucket Removal: Define the hash function for current working buckets as $\mathcal{H}_w$. Suppose we want to remove a bucket $b \in \mathcal{W}$, we keep $\mathcal{H}_w$ in use if $\mathcal{H}_w(k) \neq b$. If a key is mapped to b, a another hash function $\mathcal{H}_{w_b}(k)$ is used for remapping.

Bucket Addition: For any set of buckets $\mathcal{W}$, it creates another subset $\mathcal{W}_b$ by removing bucket b. Therefore, whenever a bucket is added, we choose to add back b to roll back the current set $\mathcal{W}_b$ to its predecessor $\mathcal{W} = \mathcal{W}_b \cup \{b\}$

Consistency of *bucket removal*: When b is removed, only keys mapped to is is reshashed, thus guaranteeing minimum disruption. For keys mapped to b, they are re-distributed by another uniform hash function $\mathcal{H}_{w_b}$, thus achieving balance.

Consistency of *bucket removal*: when b is added, we simply store the resource to bucket b rather than doing another hash with $\mathcal{H}_{w_b}$. It is obvious that this step is consistent.

Algorithm 1 — AnchorHash

```
1:  function InitAnchor(A, W)
2:      R ← ∅
3:      for b ∈ A\W do
4:          R.push(b)
5:          W_b ← A\R
6:
7:  function GetBucket(k)
8:      b ← H_A(k)
9:      while b ∉ W do
10:         b ← H_{W_b}(k)
11:     return b
12:
13: function AddBucket( )
14:     b ← R.pop()
15:     delete W_b
16:     W ← W ∪ {b}
17:     return b
18:
19: function RemoveBucket(b)
20:     W ← W\ {b}
21:     W_b ← W
22:     R.push(b)
```

Initialization. We fill the stack $\mathcal{R}$ with the initially unused buckets. For each unused bucket, remember the working set $\mathcal{W}_b$ after its removal. Obviously, $\mathcal{W}_b = \mathcal{W}b \in \mathcal{R}$ in the initialization stage.

AddBucket. As mentioned, when adding a bucket, anchorhash must add the last removed bucket b back to $\mathcal{W}$. To maintain the history of removed buckets, one can simply use a stack $\mathcal{R}$ data structure. The runtime of this function is $O(1)$.

RemoveBucket. To remove a bucket b, calculate the remaining working set $\mathcal{W}_b$ and push b into the stack $\mathcal{R}$. This step has runtime complexity of $O(1)$.

GetBucket. When a key is provided, calculate its hash using $\mathcal{H}_A$. Obviously some keys are mapped to bucket $b \notin \mathcal{W}$. When that happens, find the working set $\mathcal{W}_b$ and calculate $\mathcal{H}_{\mathcal{W}_b}$. This

step can take at most $|A|$ iterations, but it can be shown that the amortized runtime is $O(1 + ln(\frac{|\mathcal{A}|}{|\mathcal{W}|}))$.

C. PROPERTIES

Theorem .1. *Computational Complexity*

Fix $\mathcal{A}, \mathcal{W}$ and $\mathcal{R}$ such that $|\mathcal{A}| = a$ and $|\mathcal{W}| = w$. For a key k, denote by τ the number of hash operations performed by *GetBucket(k)*. Then:

1) The average τ is upper-bounded by $1 + ln(\frac{a}{w})$
2) The standard deviation of τ is upper-bounded by $\sqrt{ln\frac{a}{w}}$

Proof. Consider a fixed sequence of removed buckets $\mathcal{R} = \{r_{a-w-1} \leftarrow ... \leftarrow r_0\}$; namely r_{a-w-1} is the first removed bucket and r_0 the last removed bucket (i.e., $\mathcal{W} = \mathcal{W}_{r_0}$).

Denote the number of iterations after the loop is entered with working set $\mathcal{W}_{r_i}$ by τ_i. Let $b_i = \mathcal{H}_{\mathcal{W}_{r_i}}(k)$. When $b_i \notin \mathcal{W}$, then $b_i = r_j$ for some $r_j \in \mathcal{R}$. Then by the uniform hashing assumption, τ_i has the same distribution as $1 + \tau_j$.

For $0 \leq i \leq a - w$, define the moment generating function of τ

$$\phi_i(s) = \mathbb{E}[e^{s\tau_i}] \tag{4}$$

Then by the law of total expectation, for $i > 0$,

$$\begin{aligned}\phi_i(s) &= \mathbb{P}(b_i \in \mathcal{W})\mathbb{E}[e^{s\tau_i}|b_i \in \mathcal{W}]+ \\ &\quad \sum_{j=0}^{i-1}\mathbb{P}(b_i = r_k)\mathbb{E}[e^{s\tau_i}|b_i = r_j] \\ &= \frac{w}{w+i}\mathbb{E}[e^{s\tau_i}|b_i \in \mathcal{W}]+ \\ &\quad \frac{1}{w+i}\sum_{j=0}^{i-1}\mathbb{E}[e^{s\tau_i}|b_i = r_j]\end{aligned} \tag{5}$$

When $b_i \in \mathcal{W}$, the loop terminates in one hash. Hence

$$\mathbb{E}[e^{s\tau_i}|b_i \in \mathcal{W}] = e^s \tag{6}$$

Recall that the distribution of τ_i conditioned on $b_i = r_j$ follows the same distribution as $1 + \tau_j$. Therefore,

$$\begin{aligned}\mathbb{E}[e^{s\tau_i}|b_i = r_j] &= \mathbb{E}[e^{s(1+\tau_i)}] \\ &= e^s\phi_j(s)\end{aligned} \tag{7}$$

Together, we have:

$$\phi_i(s) = \frac{we^s}{w+i} + \frac{e^s}{w+i}\sum_{j=0}^{i-1}\phi_j(s) \tag{8}$$

□

It can be shown that, with the recursive definition, we are able to calculate its closed-form expression.

$$\phi_i(s) = \phi_{i-1}(s) \cdot \frac{w+i-1+e^s}{w+i} \tag{9}$$

Use techniques demonstrated in the original paper, it can shown that

$$\mathbb{E}[\tau_i] = \phi_i'(0) = 1 + \sum_{j=1}^{i}\frac{1}{w+j} \tag{10}$$

and therefore

$$\begin{aligned}\mathbb{E}[\tau] &= \mathbb{E}[\tau_{a-w}] \\ &= 1 + \sum_{j=1}^{a-w}\frac{1}{w+j} \\ &\leq 1 + ln(\frac{a}{w})\end{aligned} \tag{11}$$

Similarly, it can be shown that the variance of τ

$$\begin{aligned}Var(\tau) &= \phi_{a-w}''(0) - (\phi_{a-w}'(0))^2 \\ &\leq ln(\frac{a}{w})\end{aligned} \tag{12}$$

D. EVALUATION

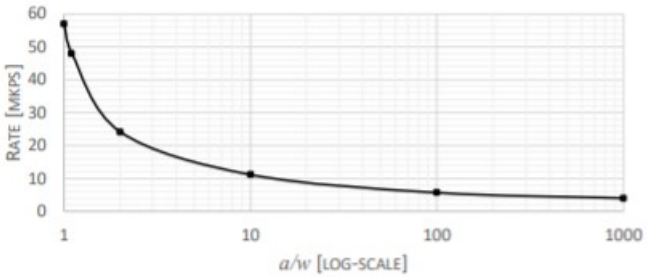

Fig. 13. Anchorhash query time [image source]

As figure 13 shows, AnchorHash achieves very high key lookup rate when the available working set is more than $\frac{1}{10}$ of the total buckets. When fewer buckets are available, the performance degradation is slow.

V. MAGLEV HASHING

A. Overview

In 2016, Google released Maglev, a large distributed software system that runs on commodity Linux servers [4]. Maglev is equipped with a variant of consistent hashing algorithm, Maglev hashing, which is used to minimize the negative impact of unexpected faults and failures on connection oriented protocols, such as TCP. The primary goal of Maglev hashing is good load balancing as compared to hashing methods such as Rendezvous hashing. Maglev hashing effectively produces a lookup table which allows lookup in constant time. The table is essentially a random permutation of nodes. During lookup, the key is hashed and the entry at that location is checked. Therefore, the resultant lookup time is simply $O(1)$ with a small constant (just the time to hash the key and do the checking).

However, despite its strength, Maglev hashing comes with its downsides. First of all, generating a new table upon node failure becomes very slow in Maglev hashing (although it is assumed that backend failure is very rare). This also limits the maximum number of backend nodes. Moreover, instead of aiming for the optimal solution, Maglev hashing only aims for minimal disruption when nodes are added or deleted. In other words, when the set of backend changes, a connection will likely (not guaranteed!) be sent to the same backend as it was before.

B. Context and Motivation of Maglev Hashing

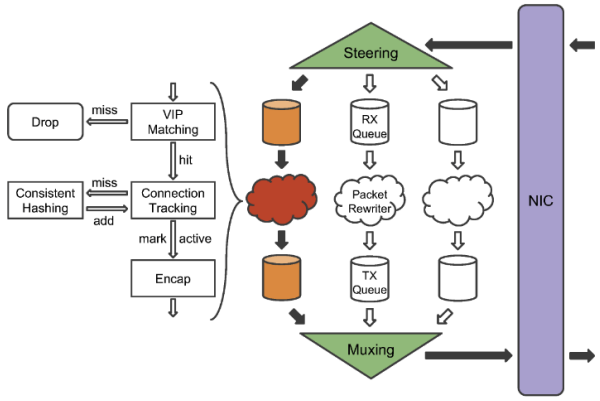

Fig. 14. Maglev Forwarder Structure (Adapted from [4])

To better comprehend Maglev hashing and how it works as a component in Maglev, we need a basic understanding of Maglev forwarder. Figure 13 illustrates the structure of Maglev forwarder. We will focus on the connection tracking table and Maglev hashing. For every incoming packet, a 5-tuple hash of the pack is computed and used to look up the hash value in the connection tracking table. The connection table stores backend selection results for recent connections. If a match is found and the selected backend is still healthy, then the result is simply reused. If the hash value of a packet does not exist in the table, Maglev will use Maglev hashing to assign a backend to the packet and store the assignment in the table.

While the connection tracking table is able to handle most requests, we need Maglev hashing to assign a backend to any unseen packet. Moreover, there will be cases such as adding or removing Maglev machines when we cannot rely on connection tracking to handle backend changes. Therefore, we need an additional way of ensuring reliable packet delivery under circumstances such as ones stated above. One possible approach is to share connection state among all Maglev machines. However, this would harm forwarding performance. In Maglev, connection states are not even shared among packet threads on the same Maglev machine to avoid contention. Hence, in such scenario, a better performing solution would be to use consistent hashing.

As we have seen in the original consistent hashing, the idea is to generate a large lookup table with each backend taking some entries in the table. To create a resilient selection process, two following properties are desired. First, *load balancing*: each backend will receive an almost equal number of connections. Second, *minimal disruption*: upon backend changes, a connection will likely be directed to the same backend as it was before.

It is critical for Maglev to balance load as evenly as possible among backends. Otherwise the backends must be aggressively overprovisioned in order to accommodate the peak traffic. On the contrary, while minimizing lookup table disruptions is important, it's okay for Maglev to have a small number of disruptions. With these considerations in mind, a new consistent hashing algorithm, Maglev hashing, was designed.

C. Maglev Hashing

In this section, we examine Maglev hashing in details. The basic idea of Maglev hashing is to assign a preference list of all lookup table positions to each backend. Then, all backends take turns filling their available most-preferred table positions until the lookup table is completely full. In this way, an almost equal share of the lookup table is assigned to each backend. If backends are heterogeneous, then we can always adjust their weights by modifying the relative frequency of the backends' turns. In following subsections, we will present Maglev hashing in steps as well as the correctness and runtime analysis of the algorithm.

1) Generating Preference List: In this subsection, we look at how the preference list for each backend is generated and populated. Let M be the size of the lookup table and $permutation$ be a list of lists where $permutation[i]$ corresponds to the preference list for backend i. Thus, $permutation[i]$ is essentially a random permutation of array $(0...M-1)$. For convenience of populating $permutation[i]$, we assign a unique name to each backend. Let $name[i]$ be the name assigned to backend i.

We first hash the backend name using two different hashing functions h_1, h_2 to generate two numbers $offset$ and $skip$. Then we generate $permutation[i]$ following the below procedure:

$$\begin{aligned} offset &\leftarrow h_1(name[i]) \mod M \\ skip &\leftarrow h_2(name[i]) \mod (M-1)+1 \\ permutation[i][j] &\leftarrow (offset + skip) \mod M \end{aligned}$$

Note that M has to be a prime number so that all possible values of $skip$ are relatively prime to M. In this way, we can generate the preference list for each backend efficiently.

2) Populating Lookup Table: In this subsection, we look at how the lookup table is populated in Maglev hashing algorithm after we obtain the preference list for each backend. Let N be the total number of backends we need to assign. The algorithm is stated in Pseudocode 1 (adapted from [4]).

Pseudocode 1 Populate Maglev hashing lookup table.

```
1:  function POPULATE
2:      for each i < N do next[i] ← 0 end for
3:      for each j < M do entry[j] ← −1 end for
4:      n ← 0
5:      while true do
6:          for each i < N do
7:              c ← permutation[i][next[i]]
8:              while entry[c] ≥ 0 do
9:                  next[i] ← next[i] + 1
10:                 c ← permutation[i][next[i]]
11:             end while
12:             entry[c] ← i
13:             next[i] ← next[i] + 1
14:             n ← n + 1
15:             if n = M then return end if
16:         end for
17:     end while
18: end function
```

We use $next[i]$ to track the next index in the preference list to be considered for backend i, The final lookup table is stored in the list $entry$. In the body of the outer $while$ loop, we iterate through all backends. For each backend i and its preference list $permutation[i]$, we first find a candidate index c from $permutation[i]$ such that $entry[c]$ is still empty. We then fill $entry[c]$ with i, which means that index c of the lookup table is no longer empty and corresponds to backend i. Eventually, as housekeeping items, we increment $next[i]$ and mark that one more entry in the lookup table has been filled. The loop keeps going until all entries in the table has been filled. Following the algorithm, we are able to populate the lookup table.

3) Analysis: In this subsection, we look at the correctness and runtime analysis of Maglev hashing algorithm. It is rather straightforward to observe that the Pseudocode 1 is guaranteed to terminate. Since for every complete iteration of the for loop, n increases by N, we do not have to worry about the outer $while$ loop (line 5-17) running forever. Regarding the inner $while$ loop (line 8-11), since $permutation[i]$ is of the same length as $entry$, we are guaranteed to find an empty spot in $entry$ given that we are already in the loop (which means $n < M$ and we haven't filled up the lookup table yet). Finally, we only increase n after successfully assigning a backend to an entry in the lookup table. Thus, when $n = M$, we are guaranteed that the lookup table is full.

In the worst case of Maglev hashing algorithm, we would have the same number of backends and lookup table entries and all backends were assigned the same preference list. The runtime complexity would then become $O(M^2)$ because we have to iterate the inner $while$ loop for $1 + 2 + ... + M - 1 = O(M^2)$ times. To avoid the worst case from happening, we always choose M such that $M \gg N$. The average time complexity is $O(M \log M)$ because at step n, we expect the algorithm to take $\frac{M}{M-n}$ tries to find an empty candidate index. Thus the total number of expected steps is $\sum_{n=1}^{M} \frac{M}{n}$.

D. Discussion and Comparison

In this section, we discuss the pros and cons of Maglev hashing. Since Maglev hashing take turns assigning backends to lookup table entries, each backend will take either $\lfloor \frac{M}{N} \rfloor$ or $\lceil \frac{M}{N} \rceil$ entries in the lookup table. Therefore, the number of entries occupied by different backends will differ by at most 1. In practice, M is usually chosen to be larger than $100 \times N$ so that at most a 1% difference in hash space is assigned to backends. The developers of Maglev hashing compares its load balancing efficiency with two other hashing methods – Karger hashing [1] and Rendezvous hashing [2]. One lookup table is populated using each method and the number of tables entries assigned to each backend is count. The total number of backends is 1,000, and the look up table size is 65,537 and 655,373. Figure 14 presents the maximum and minimum percent of entries per backend for each method and table size. M, K and R stand for Maglev, Karger and Rendezvous, respectively. Lookup table size is 65,537 for *small* and 655,373 for *large*.

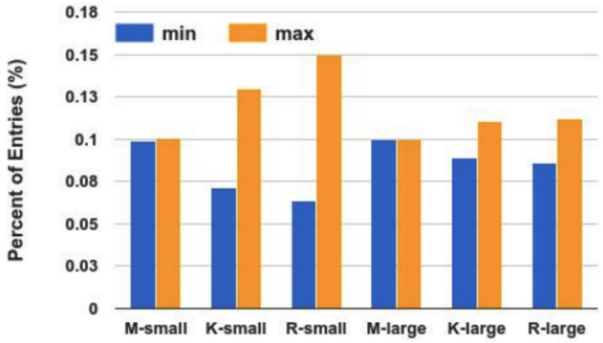

Fig. 15. Load Balancing Efficiency for Different Hashing Methods (Adapted from [4])

From the comparison we can see that Maglev hashing provides almost perfect load balancing for both hash table sizes. When table size is 65537, Karger and Rendezvous require backends to be overprovisioned by 29.7% and 49.5% respectively to accommodate the imbalanced traffic. The numbers drop to 10.3% and 12.3% as the table size grows to 655373. Thus, we conclude that Karger and Rendezvous are not suitable for Maglev's load balancing needs. What's more, with the final lookup table, finding a node in Maglev hashing becomes $O(1)$ with a small constant (just the time to hash the key and do the checking).

With that being said, Maglev hashing has its drawbacks. Developers of Maglev hashing evaluate its resilience to backend changes. Figure 15 presents the percent of changed table entries as a function of the percent of concurrent backend failures. As we can observe from the curve of the function,

Maglev hashing is rather sensitive to backend failures. On the contrary, both Karger and Rendezvous hashing guarantee that when some backends fail, the entries for the remaining backends will not be affected. However, we shall also notice that Maglev hashing is more resilient to backend changes when the table size is larger. In practice, large lookup tables are used to reduce the amount of changes upon backend failures. Also, as claimed by the developers of Maglev hashing, backend failures to be rare. Hence, the effect of the sensitivity of Maglev hashing to backend failures is not a big problem in practical usage.

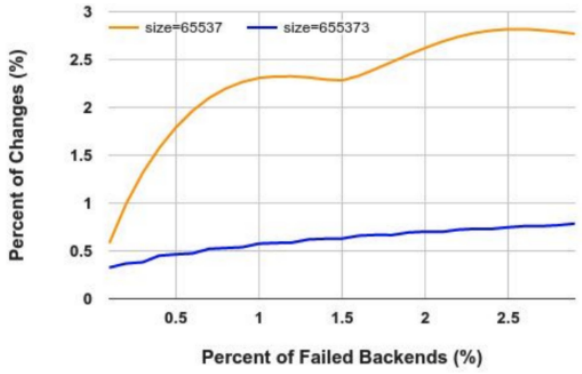

Fig. 16. Resilience of Maglev Hashing to Backend Changes (Adapted from [4])

VI. Comparison

There is no "perfect" Consistent Hashing algorithm. Each one of them has certain trade-off between its pros and cons. In this section, we present some simple practical facts of the algorithms we have discussed in this report.

A. Rendezvous 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]
}
```

Fig. 17. An implementation of the lookup function of Rendezvous hashing) [image source]

In the Rendezvous hashing algorithm, $O(n)$ lookups have to be performed in order to find an element. However, as the inner loop isn't very expensive, in practice, the lookup step of Rendezvous hashing could actually be "fast enough".

B. Jump Hash

Jump hash addresses two disadvantages of the original ring hash (Karger et. al.):

1) Jump hash does not have memory overhead
2) Jump hash has a virtually perfect key distribution

The first point is obvious from the C++ implementation of Jump hash in the previous section. And recall from the diagram we presented in the previous section, the standard deviation of bucket distribution is 0.000000764%, which is almost perfect key distribution.

Also, Jump hash runs much faster than the traditional ring hash. The loop in Jump hash algorithm executes $O(\ln n)$ times, which makes it highly preferable over several other hash algorithms.

And the main limitation of Jump hash is that it only returns an integer in the range $0 \ldots num\_buckets - 1$. It does not support arbitrary bucket names, which makes it not so applicable in certain use cases.

C. Anchor Hash

Anchor hash is mostly used as a central building block in many networking applications. Its most impressing feature (which most state-of-the-art Consistent Hashing algorithm don't have) is its endured full consistency under arbitrary changes and scaling while maintaining reasonable memory footprints and update times. It also achieves high key lookup rates.

D. Maglev Hash

Comparing to Rendezvous hash and ring hash, one of the most important goals of Maglev hash is higher lookup speed and lower memory usage. Maglev hash effectively produces a lookup table with which a single lookup could be done in constant time.

There are also two cons of Maglev hash. First, as the original paper assumes backend failures to be rare, if the algorithm encounters a node failure, generating a new table could be very slow, and also limits the maximum number of backend nodes supported. Second, due to a focus on minimal disruption when performing node addition and removal, the operations themselves are not optimal, which could lower the general performance to some extent. We could imagine that Maglev hash might not be so suitable for use cases where tons of node additions and removals are performed very frequently.

E. Single Lookup Time Comparison

In the last section, we present benchmarks for a single lookup with different node counts. Note that, the following diagram was referenced from the source page listed below the diagram. In that original paper, more hashing algorithms (not included in our study) were tested. Here, we only pay attention

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	

Fig. 18. Single lookup time (unit: nanoseconds) [image source]

to the test results of the algorithms we have discussed in the report.

VII. Conclusion

In this report, we introduced several consistent hashing methods in details. We started off by setting up the general hashing problem and listing the criteria for assessing hashing algorithms. We also presented the original consistent hashing algorithm and its correctness and runtime analysis. We then dived deep into three variants of consistent hashing that aim to improve on at least one criteria based on the original consistent hashing. Namely, they are Jump hashing, Anchor hashing, and Maglev hashing. In general, for each algorithm, we examined the context and motivation, the algorithm itself, and the correctness and runtime analysis of the algorithm. Moreover, we tried to compare these consistent hashing algorithms and discuss their pros and cons. This report is intended to be served as a survey of consistent hashing, which guides beginners to quickly understand consistent hashing and some of its variants. However, a considerable amount of details and generalization are omitted from the original papers which proposed these algorithms. Therefore, for better and more thorough understanding, it's advised to read the original papers listed as references below. Also, the consistent hashing algorithms listed here is far from being a comprehensive list of all algorithms based on consistent hashing. In the future, we plan to include more consistent hashing algorithms into this report and thus expand our project in both width and depth.

References

[1] Karger, D.; Lehman, E.; Leighton, T.; Panigrahy, R.; Levine, M.; Lewin, D. (1997). Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing. ACM Press New York, NY, USA. pp. 654–663. doi:10.1145/258533.258660

[2] Thaler, David; Chinya Ravishankar. "A Name-Based Mapping Scheme for Rendezvous". University of Michigan Technical Report CSE-TR-316-96. Retrieved 2013-09-15.

[3] Gal Mendelson, Shay Vargaftik, Katherine Barabash, Dean Lorenz, Isaac Keslassy, Ariel Orda. AnchorHash: A Scalable Consistent Hash. arXiv:1812.09674 [cs.DS].

[4] Eisenbud, D. E., Yi, C., Contavalli, C., Smith, C., Kononov, R., Mann-Hielscher, E., … Hosein, J. D. (2016). Maglev: A Fast and Reliable Software Network Load Balancer. 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16), 523–535.

[5] John Lamping, Eric Veach (2014). A Fast, Minimal Memory, Consistent Hash Algorithm. arXiv:1406.2294 [cs.DS]