Consistently Faster: A Survey and Fair Comparison of Consistent Hashing Algorithms

Massimo Coluzzi[1], Amos Brocco[2] and Tiziano Leidi[3]

Information Systems and Networking Institute, Department of Innovative Technologies,
University of Applied Sciences and Arts of Southern Switzerland, Lugano, Switzerland

Abstract

Consistent hashing is a well-known hashing technique that allows for a minimal number of resources to be remapped when a cluster is scaled. It plays a fundamental role as a data router and a load balancer in various fields, such as distributed databases, cloud infrastructures, and peer-to-peer networks. Although studies of consistent hashing algorithms relating to different usage scenarios have been carried out, the literature does not provide a thorough evaluation and comparative assessment. Therefore, this paper surveys and empirically compares the most prominent consistent hashing algorithms for distributed databases and cloud infrastructures, published from 1997 to 2021, in a fair agnostic context. Comparison has been performed by implementing all algorithms in Java and benchmarking them on commodity hardware. The metrics involved in the comparison are memory usage, initialization, resize and lookup time, balance, and monotonicity. We found *Jump*, *Anchor*, and *Dx* to outmatch the other algorithms on all the considered metrics. The measured values match the asymptotic curves. Although, some asymptotically faster algorithms have been shown to be slower in practice due to memory accesses.

Keywords

Consistent hashing, load balancing, distributed systems, survey

1. Introduction

Hashing algorithms are deterministic functions which take an arbitrary amount of data as input and produce a fixed length output called *hash value* or *digest*. In distributed database clusters, tables sharding can be used to achieve horizontal scalability and improved performance. Hashing algorithms can be used to distribute records among all shards, while the system can efficiently determine which shard is responsible for a specific record.

Hashing algorithms can be used to construct associative arrays that allow for referencing data through an arbitrary key instead of a numerical index: the key is consumed by the hashing algorithm to determine the index of the *bucket* where data resides. With Distributed Hash Tables (*DHTs*) each bucket might reside on a different node in a computer network. Dynamically resizing an hash table to change the number of buckets typically involves a considerable cost, as the elements of the original table need to be inserted into the new table. On a DHT such an operation would be required each time a node joins or leaves the network, and it incurs a considerable transmission cost, as data would need to be transferred across the network should

SEBD 2023: 31st Symposium on Advanced Database System, July 02–05, 2023, Galzignano Terme, Padua, Italy

massimo.coluzzi@supsi.ch (M. Coluzzi); amos.brocco@supsi.ch (A. Brocco); tiziano.leidi@supsi.ch (T. Leidi)
0000-0002-0877-7063 (M. Coluzzi); 0000-0002-0877-7063 (A. Brocco); 0000-0002-0877-7063 (T. Leidi)

the corresponding bucket change. To overcome this issue, consistent hashing solutions have been developed. Consistent hashing is a class of distributed hashing algorithms that strives to achieve balanced data placement and minimal reallocation costs. The goal of this paper is to analyze state of the art to determine the most effective consistent hashing algorithm to use to distribute data among the nodes of a cluster. Accordingly, the most prominent consistent hashing algorithms published since the first papers on this matter published by Thaler and Ravishankar in 1996 [1] and by David Karger et al. in 1997 [2] were considered, namely:

- **Consistent Hashing Ring**: published by D. Karger et al. in 1997 [2][3] and recognized as the first one utilizing the term *consistent hashing* to describe a solution to the aforementioned problem.
- **Rendezvous**: published by Thaler and Ravishankar in 1996 [1] [4].
- **Jump**: published by Lamping and Veach in 2014 [5].
- **Multi-probe**: published by Appleton and O'Reilly in 2015 [6].
- **Maglev**: published by D. E. Eisenbud in 2016 [7].
- **Anchor**: published by Gal Mendelson et al. in 2020 [8].
- **Dx**: published by Chaos Dong and Fang Wang in 2021 [9].

Some of them (like *Ring* and *Maglev*) are widely used in popular tools and massively distributed infrastructures like Amazon and Google. In contrast, *Anchor* and *Dx* have not yet seen a significant adoption as their publication is relatively recent. Other research efforts led to the development of consistent hashing algorithms with bounded loads [10, 11]. While relevant in the field of distributed load balancing, such a type of algorithms is out of the scope of this research work. Early surveys of consistent hashing were published in 2018 [12] and 2022 [13], but they focus mainly on lookup time. More specifically, in this paper, we aim at finding the best fitting algorithm to be used as the foundation of a smart partition scheme capable of: uniform data distribution among nodes (to avoid ”hotspots”), elastic re-balancing when the compute cluster scales, minimizing data reassignment during cluster scaling, identifying the partitions where data is stored, optimal performance with respect to initialization time, resize time, and memory usage. We implemented all the algorithms in Java to provide uniform and comparable implementations publicly available on GitHub [14] together with the benchmarking tools. The measured values match the asymptotic curves. Although, some asymptotically faster algorithms have been shown to be slower in practice due to the number of memory accesses. The considered metrics will be thoroughly discussed in Section 3.1.

2. Description of the algorithms

In this section, we will summarize the problem addressed by consistent hashing algorithms and then describe the solutions. It should be noted that the paper is not covering consistent hashing algorithms designed to work in peer-to-peer environments, and all the algorithms analyzed in this paper are designed to distribute resources among a set of known nodes. This problem is common to many applications, including distributed storage systems, database and data warehouse clusters. Every distributed system used to store and retrieve data where the underlying hardware can manage to fail can take advantage of the proposed solutions. It is left to the reader to choose which solution best fits a specific use case. This work aims to compare the solutions in a fair agnostic context.

Notation The following notations have been used throughout this paper:

- **K**: the number of looked-up keys;
- **W**: the number of working nodes in a cluster;
- **A**: the number of overall nodes in a cluster (both working and not working);
- **V**: the number of virtual nodes for each physical node in Ring;
- **M**: the number of positions in the lookup table for each node in Maglev;
- **P**: the number of probes in Multi-probe;
- **N**: a generic number of buckets (i.e., values between 0 and $N-1$);

Problem statement Given a set of keys of size K and a set of buckets of size N, we aim to distribute all the keys evenly among the available buckets. We can map each bucket to an integer between 0 and $N-1$. Then we can use a traditional hashing algorithm to map each key to an integer value. Finally, we can map such a value to an integer between 0 and $N-1$ with modular arithmetic. A good hashing function will distribute the keys evenly so that each node will get about K/N keys. Unfortunately, simplistic approaches are not suitable in a distributed scenario, because adding or removing a bucket would require a remapping of almost all the keys. In particular, suppose we identify the buckets with the nodes of a distributed system and the keys with the resources stored in the system: it is not desirable to redistribute all the resources when scaling up or down the cluster. Ideally, only the keys stored in the buckets involved in the resizing operation should be moved. Consistent hashing algorithms aim to address this situation and distribute the keys among the buckets so that adding or removing a bucket will cause only K/N keys to move.

Consistent Hashing Ring This algorithm was first introduced in 1997 by David Karger, who addressed the problem of distributing keys such that hot spots are decreased or eliminated, there is no need for constant communication between caches, and the initial key-to-bucket assignment is minimally affected as buckets change [2]. The proposed solution was aimed at improving caching in distributed networks without the need for complete information about their state. The original paper introduced the term *consistent hashing*, and described three key properties of consistent hashing, namely *smoothness*, *spread* and *load*, which are related to the balance and monotonicity metrics used in our analysis. In particular, smoothness measures the expected fraction of keys that must be moved to a new bucket in order to maintain balance, spread concerns the total number of buckets to which a object is assigned, whereas load is the number of distinct objects assigned to a particular bucket. As the name implies, this algorithm is based on a (unit) ring structure. A traditional *base* hash function maps each key to an numerical value on the unit interval $[0-1]$ and, therefore, to a position in the ring. Buckets as well are hashed and mapped to a position in the ring using a traditional hash function. The position of a key, as determined by the *Ring* algorithm, is chosen as being the first bucket found moving clockwise from the key point in the ring (this bucket is referred to as the *successor* to the key).

When a bucket is removed, all of its keys are remapped to the next bucket, traversing the ring clockwise. As this situation will lead to an unbalance in the key distribution (a bucket adjacent to the one being removed might get twice the amount keys), the basic algorithm can be improved by mapping each bucket multiple times in different positions using virtual nodes, as proposed by Dynamo [15]. A major drawback of this algorithm is related to the memory required for storing the topology of the ring and the association between nodes and buckets.

Rendezvous This algorithm, also known as *highest random weight* (HRW), was initially published in 1996 by Thaler and Ravishankar [1][4]. Even though it predates the term *consistent hashing*, the proposed approach aims at the same goals of achieving balanced data distribution and minimal disruption. In contrast to *Ring*, there is no need for virtual nodes in order to address unbalance, dropping the requirement for a map between nodes and virtual nodes (which would increase memory usage and would need to be kept continuously updated): in this regard, *Rendezvous* can be considered as a stateless algorithm, which trades minimal memory requirements for a computational time proportional to the number of buckets. The key and the identifier of each node are hashed together and the node generating the smallest hash for the given key is chosen. This algorithm uses a smaller amount of memory than Ring, but finding the correct node for a given key takes $O(W)$. It can be a reasonable solution for systems with a few nodes, but might not scale well as the number of nodes grows.

Jump *Jump* [5] works by computing when its output changes as the number of buckets increases. In contrast to the previously mentioned algorithms, it is not possible to assign arbitrary identifiers to a node. Instead, *Jump* assumes that buckets are numbered sequentially, therefore new buckets can only be added after the existing ones. Furthermore, removal of arbitrary buckets is not supported. The algorithm takes an integer key and the number of buckets N as an input, and computes a bucket number in the range $[0, N-1]$. *Jump* uses 64-bit linear congruential generator [16] (a pseudo-random generator) with a seed corresponding to the key to be hashed. It computes *pseudo-random jumps* among the buckets until it finds a position at or past the number of buckets (the output of the algorithms corresponds to the previous computed bucket). As previously mentioned, since there is no internal data structure to keep track of the working nodes, this algorithm assumes all buckets in the range $[0, N-1]$ referring to working nodes. During the scale down of the cluster, only the bucket $N-1$ can be removed. Furthermore, it cannot handle the failure of a random node. These might be considered substantial limitations preventing the potential use of this algorithm in real-life environments. *Jump's* authors claim that such limitations make the algorithm more suitable for data storage applications than for distributed web caching. Although, storage hardware can fail as any other hardware. Google describes a test where they sorted 1PB of data on 48000 hard drives [17], and at each run, at least one disk managed to break. Therefore, *Jump* seems not to be a good fit even for data storage applications.

Multi-probe Multi-probe [6] aims at addressing the main limitation of *Jump*, namely the impossibility of removing arbitrary nodes. However, in order to achieve a good balance it requires hashing the key multiple times, negatively affecting its lookup performance. *Multi-*

probe puts nodes on a hash ring as done by the *Ring* algorithm, but instead of creating many virtual nodes to improve the balance, the key is rehashed multiple times and the node with the minimal distance is subsequently chosen. This algorithm uses less memory than *Ring* (with virtual nodes), however, as the authors suggest, rehashing each key 21 times is required to optimize the trade-off between performance and balance.

Maglev *Maglev* is the name of a network load balancer and its underlying consistent hashing algorithm [7]. A network load balancer typically comprises multiple devices logically located between routers and service endpoints (generally TCP or UDP servers), and is responsible for matching each packet to its corresponding service and forwarding it to one of that service's endpoints. *Maglev* creates a lookup table where each node is listed M times ($M = 128$ in our benchmarks). Each node gets a preference list of all the lookup table positions. This list is constructed by having all the nodes take turns filling their most-preferred table positions that are still empty until the lookup table is completely filled in. This procedure gives to each node an almost equal share of the lookup table. Mapping a key to its destination node is very fast, since finding the entry in the table takes $O(1)$. On the other hand, the table needs to be recreated when nodes are added or removed. Furthermore, the authors suggest replicating each node at least 100 times, causing the table size to become significant.

Anchor The *Anchor* algorithm keeps track of the currently working nodes, and maps each key to a bucket between 0 and $A-1$, where A is the number of available nodes (working and not working). If the corresponding node is not working, the algorithm will rehash the key to hit another bucket. The lookup ends when the algorithm selects a bucket related to a working node. The algorithm uses an intelligent approach to avoid selecting the same bucket more than once. Given W the number of working nodes, the algorithm keeps a set of working nodes called working set (*WS*). Initially, the working set contains all the buckets between 0 and $W-1$. If a bucket x is removed, the working set becomes $WS\setminus\{x\}$, and *Anchor* manages to remap the keys such that all the keys initially mapped to a bucket in $WS\setminus\{x\}$ will still be mapped to the same bucket, while the keys mapped to the bucket x will be evenly spread among the remaining buckets. The authors describe two implementations. The first creates a new copy of the working set for every removed bucket using $\Theta(A + W^2)$ memory and performing the lookup in $O(ln(\frac{A}{W}))$. The second is an in-place version that uses four arrays of integers to keep track of the state of the cluster. The in-place version uses $\Theta(A)$ memory and takes $O(ln(\frac{A}{W})^2)$ for the lookup. For our comparison, we implemented the in-place version.

Dx *Dx* [9] combines ideas from several of the aforementioned algorithms. For example, it keeps track of all the available nodes (like *Anchor*) using a bit-array to mark whether nodes are working or not, and leverages a pseudo-random generator like *Jump* to determine the target bucket. In order to compute a position in the range $[0, A-1]$, *Dx* uses a pseudo-random function $R()$ initialized with the key as the seed. Accordingly, a sequence of buckets in the form $R(k), R(R(k)), R(R(R(k)))$... can be generated, and the first working bucket is chosen.

3. Benchmarks

All benchmarks have been performed on the same hardware, using an Intel® Core™ i7-1065G7 CPU with 4 cores and 8 threads, as well as 32GB of main memory. Each analyzed algorithm leverages a collision-resistant non-cryptographic hash function for mapping a key to a bucket [18]. We repeated each test for the following hash functions: *XX* [19], *MD5* [20], and *MURMUR3* [21]. Since there are no significant differences between the tested functions [22], only the results concerning *XX* will be presented. Repeating tests for every possible combination of parameters would generate a combinatorial explosion of results that would make the article verbose and difficult to interpret. For this reason, we decided to configure each algorithm with the parameters experimentally found to perform best. We used 1000 virtual nodes for each physical node in *Ring* as suggested by Dynamo [15] and a lookup table with a size greater than 100 times the number of working nodes in *Maglev* as suggested by the authors. More precisely, we used 128 (i.e., 2^7) entries per node. The authors of *Anchor* and *Dx* do not suggest a proper amount for the cluster capacity (A). We found it reasonable to set the overall capacity of the cluster to be at least 10 times the number of initial nodes.

Datasets The distribution of the keys can influence the balance of the algorithms. If the keys follow a uniform distribution, it is reasonable to assume the distribution's result to be balanced. We expect the algorithms also to be balanced for clustered distributions. We tested each algorithm with three different distributions, namely a uniform distribution, a normal or Gaussian distribution, and a clustered distribution extracted from a real-life dataset [23]. Since all the algorithms result in a good balance for the first two distributions, we will show only the worst-case scenario results represented by the clustered distribution.

3.1. Performance Metrics

We tested how the performance of the considered algorithms scales as the number of nodes grows, by repeating each test for clusters with 10, 100, 1000 and 10000 nodes. In the following, we will briefly describe every analyzed metric.

Memory usage Most of the algorithms keep an internal data structure. For clusters with a considerable amount of nodes, memory consumption is undoubtedly a critical factor, therefore we deem important to evaluate this metric in relation to the size of the network.

Time We analyzed the time taken by each algorithm to initialize its internal structures, as well as while adding or removing buckets, and when performing a lookup.

Balance With the term *balance*, we mean the ability to spread the keys evenly among the buckets. Given K the number of keys and N the number of buckets, ideally, we expect each bucket to get $\frac{K}{N}$ keys. Let $nodeMinK \leq \frac{K}{N}$ be the minimum number of keys in a bucket, and $nodeMaxK \geq \frac{K}{N}$ be the maximum number of keys in a bucket; we define $nodeMin\% = nodeMinK * \frac{N}{K}$ as the minimum amount of keys in a bucket related to the expected $\frac{K}{N}$, and $nodeMax\% = nodeMaxK * \frac{N}{K}$ as the maximum amount of keys in a bucket related to the expected $\frac{K}{N}$. The balance is defined as the interval $[nodeMin\%, nodeMax\%]$. We expect a well-balanced algorithm to have such an interval close to $[1, 1]$.

Resize balance We expect the keys to be evenly distributed also after adding or removing buckets. Given a system with K keys evenly distributed among N buckets, after adding a new bucket, we expect each bucket to have $\frac{K}{(N+1)}$ keys, whereas after removing a bucket, we expect each remaining bucket to have $\frac{K}{(N-1)}$ keys. We measured the distribution of the keys after adding and removing buckets a certain amount of times using the aforementioned $[nodeMin\%, nodeMax\%]$ interval.

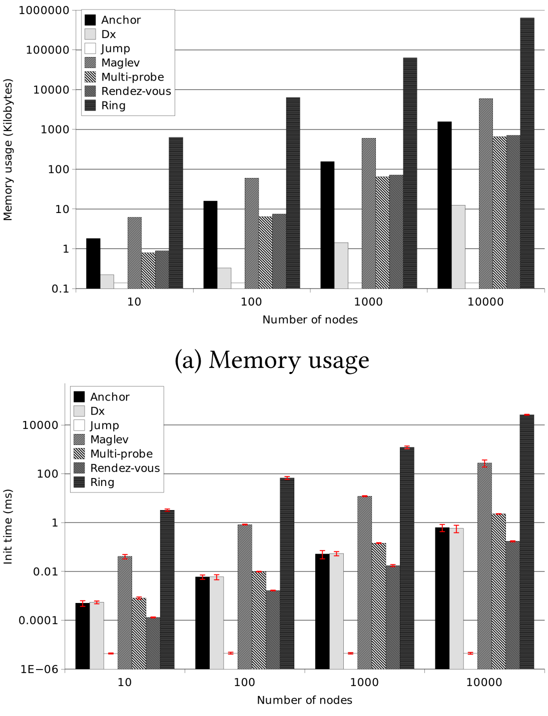
Monotonicity We expect that only the keys related to the nodes involved in the resize will move during the resize of a cluster. More precisely, only $\frac{K}{(N+1)}$ keys should move to the new node when a new node is added, and only the keys belonging to such node should move when a node is removed. We can measure this property by tracking the position of each key before and after the resize, accounting for the number of keys that are not behaving as expected.

Table 1
Asymptotic complexity

	Memory usage	Lookup time	Init time	Resize time
Ring	$\Theta(VW)$	$O(log_2(VW))$	$O(VWlog_2(VW))$	$O(Vlog_2(VW))$
Rendezvous	$\Theta(W)$	$\Theta(W)$	$\Theta(W)$	$\Theta(1)$
Jump	$\Theta(1)$	$O(ln(W))$	$\Theta(1)$	$\Theta(1)$
Multi-probe	$\Theta(W)$	$O(Plog_2(W))$	$O(Wlog_2(W))$	$O(log_2(W))$
Maglev	$\Theta(MW)$	$\Theta(1)$	$\Theta(MW)$	$\Theta(MW)$
Anchor	$\Theta(A)$	$O(ln(\frac{A}{W})^2)$	$\Theta(A)$	$\Theta(1)$
Dx	$\Theta(A)$	$O(\frac{A}{W})$	$\Theta(A)$	$\Theta(1)$

Asymptotic Results

Table 1 shows the asymptotic complexity in space and time of each algorithm. The value represented by each variable has been summarized in Section 2. Concerning memory usage, *Jump* is expected to be the best performer, because no data structure is required to run the algorithm. Conversely, *Maglev* and *Ring* shall require memory space proportional to the number of repetitions M of each node in the lookup table, and the number V of virtual nodes for each physical node respectively. Regarding the lookup time, *Maglev* should be able to perform this operation in constant time, whereas the other algorithms should scale roughly logarithmically with respect to the size of the cluster. With respect to the initialization time, *Jump* is clearly favored by not depending on any data structure. Similarly, concerning resize time, *Anchor*, *Dx*, *Jump*, and *Rendezvous* should scale without issues.



(a) Memory usage

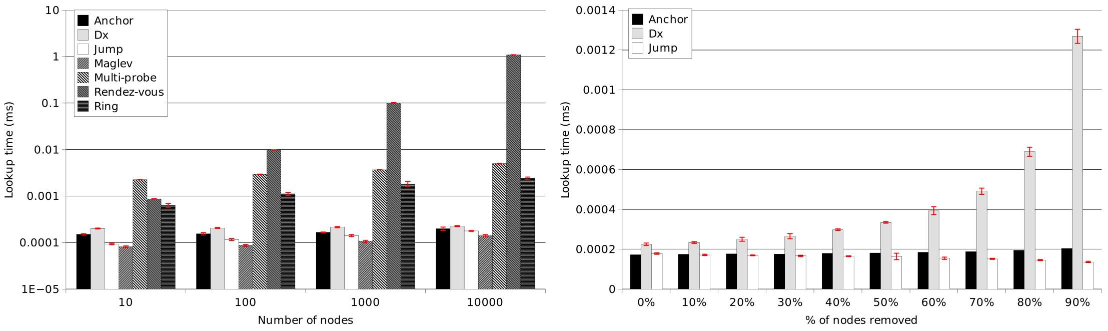
(b) Initialization time

Figure 1: Memory usage and Initialization time

Empirical Results

This section will describe the results of the benchmarks for each metric. As stated before, the results are pretty similar for different hash functions and distributions, therefore we will show only the results for the clustered distribution and the *XX* hash function. In the following sections, we will discuss each metric showing the related benchmarks in detail. The data reported in the graphs refers to an average over 10 runs: error bars will be used to represent the variability of the reported measurement across all runs.

Memory usage As expected, *Ring* and *Maglev* are the algorithms with the most significant memory usage (Figure 1a). The reason for *Ring* is the creation of 1000 virtual nodes for each physical node, while the reason for *Maglev* is the creation of a lookup table with size more than 100 times the number of nodes. Even if *Maglev* is the second worst after *Ring* in memory usage, it still uses 100 times less memory than *Ring* because we implemented the lookup table of *Maglev* using an array while the internal structure of *Ring* leverages a tree-map which causes many wrapping objects to be created. The implementation of *Ring* can be changed to optimize memory usage, but the asymptotic complexity will not change. *Dx* and *Jump* use the least amount of memory, while *Multi-probe*, *Rendezvous*, and *Anchor* are in the middle of the pack. For all the algorithms except *Jump*, the memory usage is proportional to the number of buckets. As shown in the last chart, *Jump* uses a constant amount of memory, while *Dx* uses an amount



(a) Best case

(b) Worst case

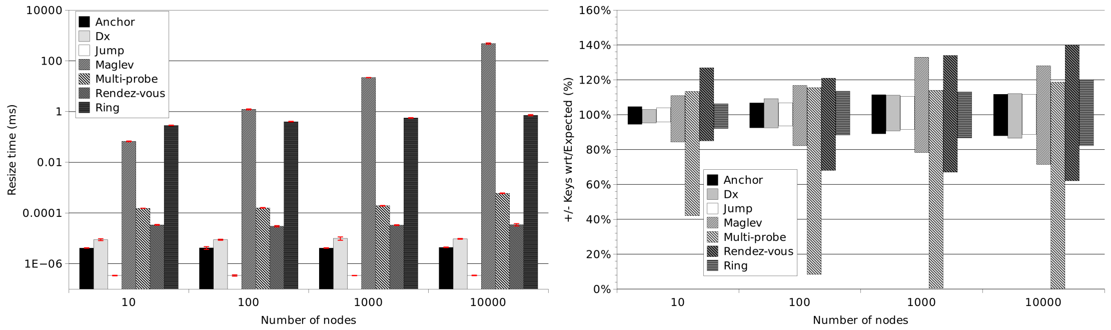
Figure 2: Lookup time

of memory proportional to the number of buckets, though, Dx's internal data structure is a bit-array, and the actual consumption of memory is limited.

Initialization time

The initialization time (Figure 1b) is the time needed by the algorithm to initialize its internal data structure. As expected, *Ring* is the slowest algorithm because it needs to create a sorted list of $V * W$ virtual nodes, with a suggested value of V equal to 1000, which takes time $O(VWlog_2(VW))$. The second slowest is *Maglev*, the reason is the population of a lookup table with more than 100 times the number of buckets. Every node will choose a position in the table. If the chosen position is already taken, another one will be evaluated. The process continues until every entry in the table is filled. *Anchor*, *Dx*, and *Multi-probe* use an average amount of time for creating the list of nodes. *Multi-probe* uses a sorted list whose population takes $O(Wlog_2(W))$. The initialization time for Anchor and *Dx* is proportional to the cluster's capacity (overall number of nodes), in our benchmark we assumed the capacity to be ten times the number of the initial working nodes (W). The best performing algorithms during initialization are *Rendezvous* and *Jump*. *Rendezvous* uses a set of nodes that can be populated in $\Theta(W)$. On the other hand, *Jump* does not have any data structure to initialize; therefore, it initializes, as expected by the asymptotic analysis, in constant time.

Lookup time The lookup time is the time needed to find the node a given key belongs to, assuming the cluster is stable (no nodes added or removed). In the best case, as shown in Figure 2a, *Rendezvous* is the slowest because it checks the key against every node to find the best match. It causes the lookup to be linear in the number of working nodes $\Theta(W)$; Multi-probe and *Ring* confirm a complexity of $O(log_2(N))$. As we can see in the chart, *Multi-probe* is slower than *Ring* by a constant factor. In the case of *Multi-probe*, we used 21 probes as suggested in the paper, therefore the complexity is $O(21log_2(W))$. The complexity of *Ring* is $O(log_2(VW))$. We used $V = 1000$ as suggested by the authors, therefore, the complexity is $O(log_2(1000W)) \leq O(10 + log_2(W))$. The remaining algorithms are very fast, in the order of nanoseconds. From a theoretical point of view, *Jump* should be slower than *Anchor* and *Dx*, but



(a) Resize time

(b) Balance of key distribution

Figure 3: Resize time and balance

practically it shows to be faster because it does not access the memory, and therefore it works at CPU speed. On the other hand, *Maglev* should perform the lookup in constant time ($\Theta(1)$), but using a big lookup table causes many memory accesses that slow down the algorithm. The asymptotic complexity for *Anchor* is $O(ln(\frac{A}{W}))$ and the complexity for *Dx* is $O(\frac{A}{W})$; therefore, the worst-case scenario for these two algorithms is when W is much smaller than A (i.e., $\frac{A}{W}$ is big). We tested the worst-case scenario using a cluster of 10000 nodes and collecting the lookup time after removing the nodes from 10% to 90%. As shown in Figure 2b for *Dx*, the lookup time grows quite rapidly while the lookup time of *Anchor* is less affected by the removal of the nodes. As expected, *Jump* is getting faster because its complexity is proportional to the number of working nodes. Even in the worst-case scenario, these algorithms are faster in lookup than all the others (except for *Maglev*).

Resize time Resize time is the time needed by the algorithm to update the data structure when adding or removing buckets. We tested different situations by adding and removing buckets from 10% to 50% of the initial cluster size. We will show the average time for adding or removing one bucket, all algorithms but *Maglev* do not allow multiple removals at the same time. As expected, *Ring* and *Maglev* are the slowest due to the size of their internal data structures. In particular, *Maglev* is very slow because every change in the cluster size causes the rebuild of the lookup table. Therefore, even adding or removing one single bucket will have a complexity of $\Theta(100 * W)$. This cost could be amortized should multiple removals happen concurrently (as the lookup table could be rebuilt only once). Nonetheless, as shown in Figure 3a, the difference compared to other algorithms is so significant that such an optimization is unlikely to make *Maglev* fair well in this metric. The second slowest resize time is obtained by *Ring* due to adding or removing a certain number of virtual nodes (e.g., 1000) for each physical node. In particular, the virtual nodes need to be added to a sorted list, therefore the complexity to add or remove a node is $O(Vlog_2(VW))$. *Jump* is the fastest because it does not have any data structure to update. *Anchor*, *Dx*, and *Rendezvous* are slower but still resized in constant time. *Multi-probe* keeps a sorted list of nodes, and adding or removing a node will take $O(log_2(W))$.

Balance The balance is conditioned by the way keys are distributed. As mentioned before, we tested this metric using three different distributions: *uniform* (best-case scenario where keys are already uniformly distributed), *normal* (average-case scenario where keys are distributed following a Gaussian curve), and *clustered* (worst-case scenario where keys are concentrated in some intervals leaving other areas empty). In the best-case and average-case scenarios, almost all algorithms are well balanced except for *Multi-probe*, which seems to ignore some nodes completely. In the worst-case scenario, represented by a clustered distribution of the keys, there are algorithms able to keep a satisfying balance while others lose balance completely. To represent the balance, we measured each node's percentage of keys collected. Then, we took the node with the lower percentage and the one with the highest. The chart in Figure 3b shows a list of intervals. A well-balanced algorithm shows an interval close to 100%. The larger is the interval, and the worst is the balance. We can see that *Anchor*, *Dx*, and *Jump* are pretty well balanced throughout our benchmark. On the contrary, *Rendezvous*, *Ring*, and *Maglev* lose their balance as the cluster grows. In particular, *Rendezvous* is not balanced even on small clusters. Finally, *Multi-probe* is the worst performing in balance. When the cluster grows in size, *Multi-probe* ends by completely ignoring some nodes, which is an undesired behavior.

Balance after resizing We performed the same test after resizing the cluster. We added and removed random nodes several times before executing the balance test. The results of this test prove that the resize of the cluster does not affect the balance of the selected algorithms.

Table 2
Performance results

	Memory Usage	Init Time	Resize Time	Lookup Time	Balance	Resize Balance	Monotonicity
Anchor	$1.5\ MB$	$< 650\ \mu s$	$< 5\ \mu s$	$< 200\ ns$	equal	equal	equal
Dx	$12.5\ KB$	$< 600\ \mu s$	$< 10\ \mu s$	$< 230\ ns$	equal	equal	equal
Jump	$150\ Bytes$	$< 5\ ns$	$< 0.5\ ns$	$< 180\ ns$	equal	equal	equal

Monotonicity Finally, we tested the property of monotonicity. This property states that only the keys related to the nodes involved in the resize should move during the resize of a cluster. We tested this property by tracking the position of each key before and after the resize and by counting the number of keys that are not behaving as expected. We sized the number of keys in order to have 1000 keys for every node. Subsequently, starting with a stable cluster, we stored the destination of each key. Next, we removed up to 50% of the nodes and stored the destination of each key after the removal. Next, we restored the removed nodes and stored the destination of each key after the restoration. Finally, we compared the stored values. Monotonicity proved to hold for every algorithm, and when we add or remove up to the 50% of the nodes, only the keys involved in the resize will move from one node to another. The nodes not involved in the resize will not be affected.

Ranking

After analyzing all the benchmarks, we graded each algorithm for each metric and grouped them into three categories (best performing, average performing, and worst performing), as shown in Table 3. The algorithms showing the best performance for all the analyzed metrics are *Anchor*, *Dx*, and *Jump*. We compared in more detail those three algorithms and summarized the results in Table 2. The comparison is based on a cluster of 10000 working nodes, assuming an overall capacity (A) of ten times the number of working nodes. It is also based on the *XX* hash function and the clustered distribution. These three algorithms are equal in balance, balance after resizing, and monotonicity. *Jump* does not use any internal data structure; therefore, it uses minimal memory and is the fastest in any metric. On the other hand, *Jump* has the remarkable limitation that only the last inserted node can be removed. It means that *Jump* is not able to handle the failure of a random node in the cluster, making it unsuitable for production environments and jeopardizing its excellent performance. *Anchor* and *Dx* are addressing this limitation by using an internal data structure to keep track of the cluster's nodes (both working and removed) which causes the algorithms to be slower during initialization and resize, but they reach similar performance in lookup operations. Nevertheless, *Anchor* and *Dx* are the fastest after *Jump*, performing lookups in the range of nanoseconds and avoiding Jump's limitations which makes them the best choice to handle consistent hashing in non-peer-to-peer environments.

Table 3
Performance ranking

Performance	Memory usage	Init time	Resize time	Lookup time	Balance	Resize balance	Monotonicity
Best	**Anchor** **Dx** **Jump** Multi-probe Rendezvous	**Anchor** **Dx** **Jump** Rendezvous	**Anchor** **Dx** **Jump** Multi-probe Rendezvous	**Anchor** **Dx** **Jump** Maglev	**Anchor** **Dx** **Jump**	**Anchor** **Dx** **Jump**	**Anchor** **Dx** **Jump** Maglev Multi-probe Rendezvous Ring
Average	Maglev	Multi-probe	Ring	Multi-probe Ring	Maglev Rendezvous Ring	Maglev Rendezvous Ring	
Worst	Ring	Ring Maglev	Maglev	Rendezvous	Multi-probe	Multi-probe	

4. Conclusions

This paper surveyed and compared the most prominent consistent hashing algorithms for distributed databases and cloud infrastructures, published from 1997 to 2021. We analyzed Ring, the first algorithm of this kind, published in 1997 but still adopted by many distributed systems. We analyzed Rendezvous, published in 1996, which can be considered its principal competitor. We compared them with Jump, Multi-probe, and Maglev, published by Google between 2014 and 2016. Maglev is the algorithm used by Google's network load balancers. Finally, we analyzed Anchor and Dx published in 2020 and 2021. These last two algorithms have not yet seen a significant adoption, but they are very promising and have interesting asymptotic curves. The metrics involved in the comparison were memory usage, initialization, resize and lookup time, balance, and monotonicity. We performed several benchmarks using different key distributions and hashing algorithms. Our results match the asymptotic curves and show that *Ring* suffers from higher than average memory usage due to virtual nodes and, similar to *Maglev*, is penalized due to its algorithmic complexity during initialization and resizing. Moreover, *Multi-probe* is the worst performer when it comes to balance and resize balance. Overall, we found *Jump*, *Anchor*, and *Dx* to perform better than the other algorithms on all the considered metrics. In particular, Jump is the best-performing algorithm because it does not use any internal data structure, which allows it to work at CPU speed. On the other hand, using no internal data structure makes Jump a stateless algorithm unable to handle random failures. This limit makes Jump unsuitable for real-life environments despite its excellent performance, therefore in the majority of use cases, only *Anchor* and *Dx* are worth considering.

References

[1] D. Thaler, C. V. Ravishankar, A name-based mapping scheme for rendezvous, in: Technical Report CSE-TR-316-96, University of Michigan, University of Michigan, 1996.

[2] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, D. Lewin, Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web, in: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing, 1997, pp. 654–663.

[3] D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, Y. Yerushalmi, Web caching with consistent hashing, Comput. Netw. 31 (1999) 1203–1213. URL: https://doi.org/10.1016/S1389-1286(99)00055-9. doi:10.1016/S1389-1286(99)00055-9.

[4] D. G. Thaler, C. V. Ravishankar, Using name-based mappings to increase hit rates, IEEE/ACM Transactions on networking 6 (1998) 1–14.

[5] J. Lamping, E. Veach, A fast, minimal memory, consistent hash algorithm, arXiv preprint arXiv:1406.2294 (2014).

[6] B. Appleton, M. O'Reilly, Multi-probe consistent hashing, arXiv preprint arXiv:1505.00062 (2015).

[7] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, J. D. Hosein, Maglev: A fast and reliable software network load balancer, in: 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16), 2016, pp. 523–535.

[8] G. Mendelson, S. Vargaftik, K. Barabash, D. H. Lorenz, I. Keslassy, A. Orda, Anchorhash: A scalable consistent hash, IEEE/ACM Transactions on Networking 29 (2020) 517–528.

[9] C. Dong, F. Wang, Dxhash: A scalable consistent hash based on the pseudo-random sequence, arXiv preprint arXiv:2107.07930 (2021).

[10] V. Mirrokni, M. Thorup, M. Zadimoghaddam, Consistent hashing with bounded loads, 2016. URL: https://arxiv.org/abs/1608.01350. doi:10.48550/ARXIV.1608.01350.

[11] J. Chen, B. Coleman, A. Shrivastava, Revisiting consistent hashing with bounded loads, in: AAAI, 2021.

[12] D. Gryski, Consistent hashing: Algorithmic tradeoffs, 2018. URL: https://dgryski.medium.com/consistent-hashing-algorithmic-tradeoffs-ef6b8e2fcae8.

[13] A. Slesarev, M. Mikhailov, G. Chernishev, Benchmarking hashing algorithms for load balancing in a distributed database environment, in: P. Fournier-Viger, A. Hassan, L. Bellatreche, A. Awad, A. Ait Wakrime, Y. Ouhammou, I. Ait Sadoune (Eds.), Advances in Model and Data Engineering in the Digitalization Era, Springer Nature Switzerland, Cham, 2022, pp. 105–118.

[14] Institute of Information Systems and Networking at SUPSI, java-consistent-hashing-algorithms, 2021. URL: https://github.com/SUPSI-DTI-ISIN/java-consistent-hashing-algorithms.

[15] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, W. Vogels, Dynamo: Amazon's highly available key-value store, SIGOPS Oper. Syst. Rev. 41 (2007) 205–220. doi:http://doi.acm.org/10.1145/1323293.1294281.

[16] P. L'Ecuyer, Tables of linear congruential generators of different sizes and good lattice structure, Math. Comput. 68 (1999) 249–260. doi:10.1090/S0025-5718-99-00996-5.

[17] G. Czajkowski, Sorting 1pb with mapreduce, 2008. URL: https://googleblog.blogspot.com/2008/11/sorting-1pb-with-mapreduce.html.

[18] C. Estébanez, Y. Saez, G. Recio, P. Isasi, Performance of the most common non-cryptographic hash functions, Software: Practice and Experience 44 (2014) 681–698.

[19] xxhash, https://github.com/Cyan4973/xxHash, 2014. Accessed: 2022-05-02.

[20] R. L. Rivest, The MD5 message digest algorithm, RFC 1321, 1992. ftp://ftp.rfc-editor.org/in-notes/rfc1321.txt.

[21] A. Appleby, Murmurhash3, https://github.com/aappleby/smhasher, 2015. Accessed: 2022-05-02.

[22] S. Priyamvada, Analysis of various hash function, International Journal of Innovative Science and Research Technology 3 (2018).

[23] M. Coluzzi, A. Brocco, P. Contu, T. Leidi, Clustered distribution for consistent hashing algorithms, 2022. URL: https://dx.doi.org/10.21227/r6ps-dz05. doi:10.21227/r6ps-dz05.