# A survey and comparison of consistent hashing algorithms

Massimo Coluzzi*, Amos Brocco*, Patrizio Contu†, Tiziano Leidi*

*Information Systems and Networking Institute, Department of Innovative Technologies,
University of Applied Sciences and Arts of Southern Switzerland, Lugano, Switzerland
Email: {massimo.coluzzi,amos.brocco,tiziano.leidi}@supsi.ch

†Natzka SA, Lugano, Switzerland
Email: patrizio.contu@natzka.com

***Abstract*—When a cluster is scaled, a well-known hashing technique called consistent hashing permits only a small number of resources to be remapped. In a variety of settings, including distributed databases, cloud infrastructures, and peer-to-peer networks, it performs key functions as a data router and load balancer. The literature does not offer a full review and comparative assessment of consistent hashing algorithms, despite the fact that research on these algorithms in relation to various usage scenarios has been conducted. Consequently, in a rather neutral setting, this study surveys and empirically compares the most widely used consistent hashing algorithms for distributed databases and cloud infrastructures, published from 1997 to 2021. All algorithms have been implemented in Java and benchmarked on common hardware to perform the comparison. 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 the number of memory accesses.**

***Index Terms*—Consistent hashing, load balancing.**

## I. Introduction

With an arbitrary amount of data as input, hashing algorithms are deterministic operations that generate a fixed-length result known as a *hash value* or *digest*. Hashing techniques can be used in distributed systems to distribute objects among nodes so that all nodes evenly divide the load, and clients can quickly identify which node is in charge of a particular object. By using the key to determine the index of the *bucket* where the data is stored, hashing techniques can be used to build associative arrays that enable data to be accessed using any key rather than a numeric index. In a Distributed Hash Table (DHT), each bucket might be located on a different node in a computer network. Due to the necessity to move data to a new node, dynamically enlarging a hash table like this usually comes at a high cost and might result in unbalanced data placement. Consistent hashing solutions have been created to solve these problems and exhibit low reallocation costs (in terms of data that need to be transferred to a different node). The goal of this paper is to analyze the state of the art to determine the most effective consistent hashing algorithm. For this study, the most relevant consistent hashing algorithms proposed 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:

- **Ring**: by D. Karger et al. (1997) [2][3].
- **Rendezvous**: by Thaler and Ravishankar (1996) [1] [4].
- **Jump**: by Lamping and Veach (2014) [5].
- **Multi-probe**: by Appleton and O'Reilly (2015) [6].
- **Maglev** by D. E. Eisenbud (2016) [7].
- **Anchor** by Gal Mendelson et al. (2020) [8].
- **Dx** by Chaos Dong and Fang Wang (2021) [9].

An early survey of consistent hashing focusing on lookup time was published online in 2018 [10]. Our goal is to consider additional metrics, namely: **memory usage**, **initialization time**, **lookup time**, **resize time** (when scaling the cluster), **balance** (how well keys are spread among nodes), **balance after resize**, and **monotonicity** (only the keys involved in the resizing should be moved). Due to space constraints, this paper only details the results about the lookup time and memory usage. Moreover, for a detailed description of the inner-working of each algorithm, the reader should refer to the corresponding literature. The source code (Java) for all the considered algorithms is publicly available [11].

### A. 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. 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.

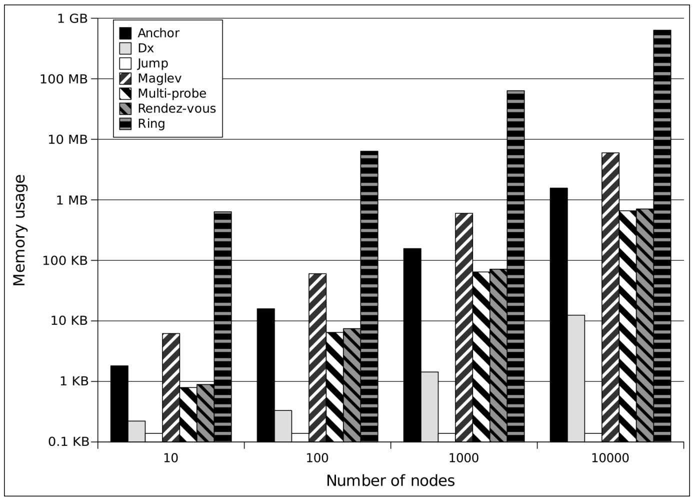

Fig. 1. Memory usage

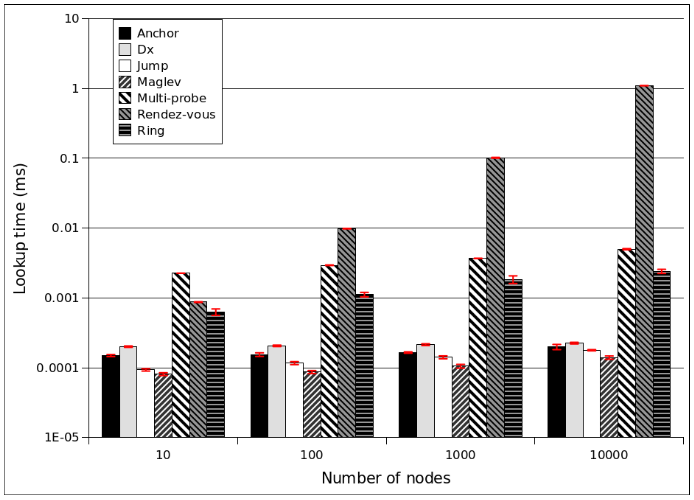

Fig. 2. Lookup time

## II. 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 [12]. We considered several functions (*XX* [13], *MD5* [14], and *MURMUR3* [15]), but since there are no significant differences [16], only the results concerning *XX* will be presented. We repeated each test for clusters with 10, 100, 1000 and 10000 nodes. The asymptotic complexity of each algorithm is reported in Table I ($A$ denotes the number of overall nodes in a cluster, whereas $W$ denotes the number of working nodes; $V$ is the number of virtual nodes for each physical node in *Ring*, while $M$ and $P$ are the number of positions in the lookup table for each node in *Maglev*, and of probes in *Multi-probe* respectively).

TABLE I
Asymptotic complexity

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

### *Memory usage*

Most of the algorithms (except *Jump*) keep an internal data structure. For clusters with a considerable amount of nodes, memory consumption can become a critical factor. As expected, *Ring* and *Maglev* are the algorithms with the most significant memory usage (Figure 1).

### *Lookup time*

Concerning lookup time (Figure 2) *Rendezvous* is the slowest in this metric because it checks the key against every node to find the best match. Multi-probe and *Ring* confirm their asymptotic complexity. Although *Jump* should be slower than *Anchor* and *Dx*, it performs faster because of less memory accesses. In a worst-case scenario, using a dynamic cluster where between 10% and 90% of the nodes are removed (starting from 10000 nodes), the lookup time of *Dx* grows quite rapidly, while *Anchor* is less affected (Figure 3).

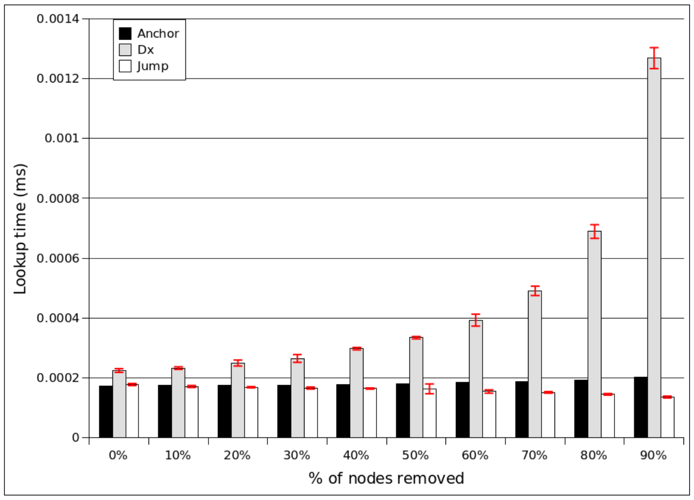

Fig. 3. Lookup time (worst-case)

## III. Conclusions

This paper surveyed and compared the most relevant consistent hashing algorithms for distributed databases and cloud infrastructures, published between 1997 and 2021. Our results match the asymptotic curves, with *Jump*, *Anchor*, and *Dx* producing the best results. Some asymptotically faster algorithms have been shown to be slower in practice due to the number of memory accesses. Jump is the best-performing stateless algorithm, but it is unable to handle random failures. Therefore, only Anchor and Dx might be suitable for real-life environments.

## References

[1] D. Thaler and 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, and 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, and Y. Yerushalmi, "Web caching with consistent hashing," *Comput. Netw.*, vol. 31, no. 11–16, p. 1203–1213, may 1999. [Online]. Available: https://doi.org/10.1016/S1389-1286(99)00055-9

[4] D. G. Thaler and C. V. Ravishankar, "Using name-based mappings to increase hit rates," *IEEE/ACM Transactions on networking*, vol. 6, no. 1, pp. 1–14, 1998.

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

[6] B. Appleton and 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, and 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, and A. Orda, "Anchorhash: A scalable consistent hash," *IEEE/ACM Transactions on Networking*, vol. 29, no. 2, pp. 517–528, 2020.

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

[10] D. Gryski. Consistent hashing: Algorithmic tradeoffs. [Online]. Available: https://dgryski.medium.com/consistent-hashing-algorithmic-tradeoffs-ef6b8e2fcae8

[11] Institute of Information Systems and Networking at SUPSI. java-consistent-hashing-algorithms. [Online]. Available: https://github.com/SUPSI-DTI-ISIN/java-consistent-hashing-algorithms

[12] C. Estébanez, Y. Saez, G. Recio, and P. Isasi, "Performance of the most common non-cryptographic hash functions," *Software: Practice and Experience*, vol. 44, no. 6, pp. 681–698, 2014.

[13] "xxhash," https://github.com/Cyan4973/xxHash, accessed: 2022-05-02.

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

[15] A. Appleby, "Murmurhash3," https://github.com/aappleby/smhasher, accessed: 2022-05-02.

[16] S. Priyamvada, "Analysis of various hash function," *International Journal of Innovative Science and Research Technology*, vol. 3, 2018.