Load Balancing for Consistent-Hashing based Partitioning Problem

Kun Wu | kunwu@kth.se
Xiaoying Sun | xsu@kth.se

Oct 28, 2021

Contents

Abstract

When designing horizontally scalable systems, one requirement is to dynamically partition the requests over a set of nodes (i.e., storage) in the system. For example, many databases rely on consistent hashing to distribute the load across available storage hosts. However, consistent hashing may fail to handle the hotspot issue (i.e., some content is more frequently accessed due to skewed workload). As popular content requests will be routed to the same subset of servers, these requests may show long response latency. This paper analyzes the impact of skewed workload on a key-value database that adopts different consistent hashing implementations. Our research verifies impacts with respect to latency, throughput, and loads on the different nodes. This project provides a chance for comparison of different algorithms.

Keywords: Consistent Hashing, Partitioning, Database, Workload Skew, Horizontal Scaling.

1 Introduction

Modern cloud services need to handle billions of requests every second and require meeting strict performance objectives[1]. These objectives are often defined as a percentile of the latency distribution (such as the 99th[2]). To ensure the quality of service, many of them rely on the in-memory key-value store(KVS) to cache frequently accessed data. These systems often rely on horizontal partitioning to divide disjoint sets of rows[3].

Among the different partitioning strategies of KVS, one common technique is Consistent hashing[4]. It provides a way of distributing requests among a changing population of servers and needs minimum data movement[4] associated with changes in the server topology[5]. In this algorithm, each server is mapped to a point on a circle with a hash function. To determine which nodes store which data, one can visualize the mapping of the nodes and data onto a ring (where they do not overlap) and circling it clockwise, each node is responsible for storing the data between itself and the preceding node.

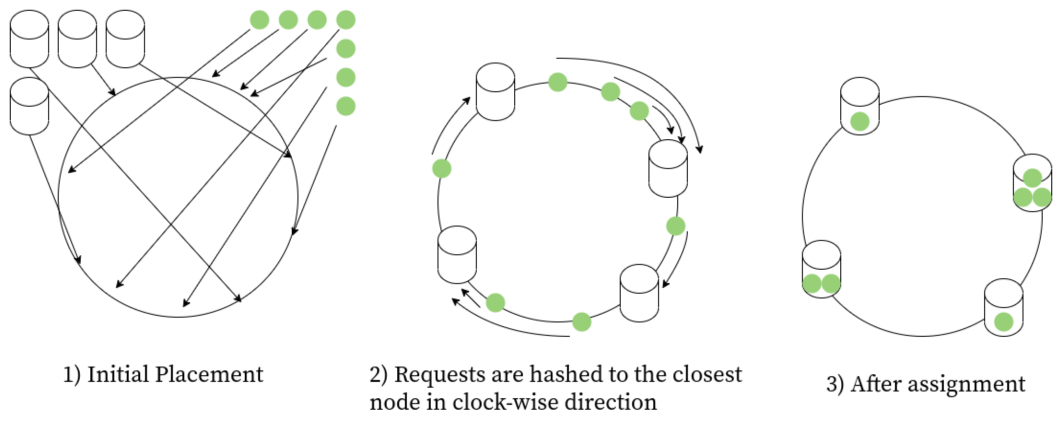
1) Initial Placement
2) Requests are hashed to the closest node in clock-wise direction
3) After assignment

Figure 1: Consistent Hashing request and node assignment. Requests are Green.

Despite its popularity[2], consistent hashing may fail to guarantee even distributions of requests. For equally distributed requests, it balances such loads as well as choosing a random server. However, for some popular content[6], [7] (as usual for the internet), it will route all the requests to the same subset of servers. Overloaded servers need to handle more requests than their counterparts and this leads to uneven response time[8]. As pointed out[9], [10], the 99th-ile latency can be more than an order of magnitude higher than the median.

2 Related Work

Consistent hashing has been applied in many large-scale projects, such as Riak, Cassandra and Dynamo[2], which need to distribute the data among available servers. However, most of them do not describe how to handle workload skews.

Hot Spots[11] and Load Spikes are two common types of workload skews. The first one can be illustrated with the case of celebrities with several million followers[12] causing the system to slow down because of the large amount of traffic generated from their updates. While the load spikes occur when the number of requests increases rapidly over a short period.

To mitigate such load imbalance issues[13], lots of research has been done and various algorithms and techniques have been proposed. Thaler and Ravishankar proposed the rendezvous algorithm[14]. It takes the request's key and for each candidate node computes a hash function value h. It then returns the index of the node with the highest value. The computing complexity is proportional to the number of nodes. Lamping and Veach from Google proposed a fast algorithm called jump consistent hash[15]. In comparison to the rendezvous algorithm, it requires no storage of nodes and can evenly divide the keyspace among the nodes, although the nodes here must be numbered sequentially. Similarly, Appleton designed multi-probe consistent hashing[16] which performs multiple lookups per key in a hash table of nodes and return the subsequent node which is closest to a key's hash. It needs no additional storage except for a hash table and gets a peak-to-average load ratio of $1 + \varepsilon$ with just $1 + 1/\varepsilon$ lookups per key. More recently, Consistent Hashing with Bounded Loads (CH-BL)[17], [18] has been introduced to account for server capacity. In CH-BL, if an object is about to be assigned to a full bin, it overflows or cascades into the nearest available bin in the clockwise direction.

Additionally, lots of other variants have been proposed, such as Hot Ring[19] and Random Jump Consistent Hashing[18]. However, a comparison and summary of these are missing and hence this paper is devoted to that.

3 Research Questions

Our research looks into several variants of consistent hashing algorithms, which differ greatly regarding their computing complexity, elasticity, and balancing degrees. We also discuss possible limitations of each solution and select some of them for our tests. To find the impact of skewed workload on the performance of consistent hashing algorithms, we engineered a testbed where synthetic data are used to measure throughput and latency. Our quantitative analysis provides a way to compare each algorithm and serve as a future reference.

4 Research Methodology

Our paper is based on a project which includes implementations and experiments. For traditional STEM (Science Technology Engineering Mathematics) research, experiments are a classic research methodology. Our analysis is based on data from experiments.

4.1 Preparations

Our project starts as a repository on GitHub[20]. After some tries, we decided to implement these algorithms in Rust as it provides fast ways to do prototypes and great runtime efficiency.

4.1.1 Algorithm & Implementation

We designed an in-memory key-value store with several different balancing strategies. The key and value are both unsigned integers (8 bytes on most operating systems). In addition to the basic consistent hashing algorithm, we implemented Jump Hash[15], Multi-Probe Consistent Hashing[16], and Maglev Hashing[21]. Each algorithm represents a different implementation but they all provide a uniform set of interfaces: *insert* and *get*. For simplicity, we do not consider the fault tolerance of any of these schemes. Moreover, we do not consider the read/write ratios (i.e., the ratio of insert to get).

4.1.2 Workload Generation

We use synthetic data to run our experiments. To observe the impact of workloads, we generated three different kinds of key distributions: uniform, normal, and lognormal. Each key is paired with a random value and the length of operation streams vary from 1000 to 50,000, step being 1000.

4.1.3 Experiment Environment

We ran all experiments on a machine with an Intel Core™ i5-8265 at 1.6 GHz running Ubuntu Linux 18.04 with a 5.4.0 kernel and 8 GB RAM. R language is our visualization tool.

4.2 Procedure

At the beginning of a test, a workload generator is instantiated. It outputs a number of requests whose keys obey a certain distribution. Our key-value database is initialized with a fixed number of nodes.

After initialization, each request invokes a *get* operation and it returns the index of the node to which this request should be routed. With this index, a request is assigned to a node. A node needs some time to handle a request and therefore some requests have to wait in a queue before being processed.

This is repeated until all requests are processed, then we output the throughput of our load balancer as well as the actual load on each node.

4.3 Measurement & Data Analysis

In our tests, we record the overall time to distribute all requests and the number of requests each node handled. To calculate the throughput, we divide the number of requests by the overall time.

The latency for each request is also kept as a measurement of the response time. It consists of a queueing time and a processing time. In our model, each node is running in a single process so the queueing time depends on how many requests are there in the queue, whereas the processing time obeys normal distribution. This is a simplified way to simulate network communication time as if our KVS were distributed instead of in-memory. We use the 99th percentile to represent the upper limit of response time.

The balancing degree between nodes is another important factor for our experiment. Since our model does not consider heterogeneous nodes, the expected load for each node is the same and therefore equals the reciprocal of the node number, while the actual load for each node is the number of requests handled divided by the total number of requests. We use the ratio between actual and expected load to measure the balancing degree for a node. If this ratio is close to 1, it means that the actual load is close to the expected load. We calculate the standard error of this ratio and use it to indicate the system's balancing degree.

If the standard error is close to zero, it means that requests are almost evenly distributed to nodes.

5 Results and Analysis

5.1 Results

5.1.1 Throughput

Throughput (i.e., million operations per second) represents the processing efficiency of the load balancer. It measures the number of requests an algorithm can distribute within a unit of time. As Figure5.1 shows, different implementations show different levels of throughput. When the number of server is fixed and the number of requests is increasing, the throughput remains stable.

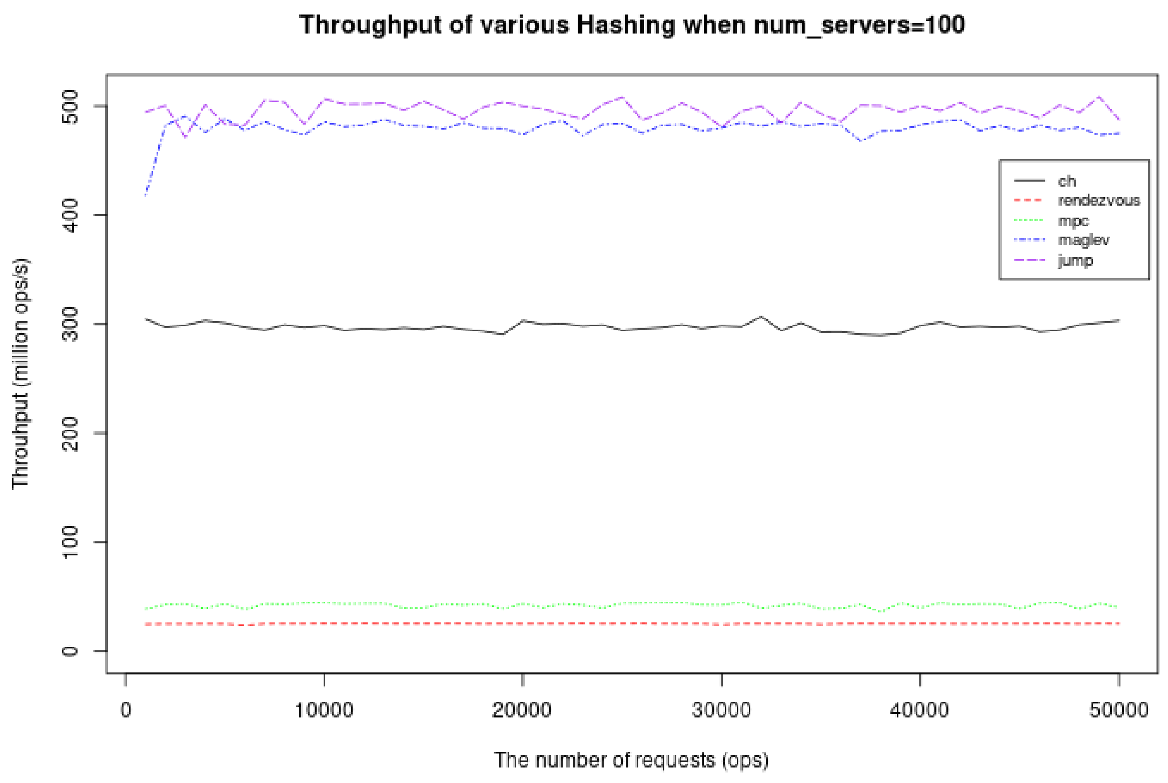

Figure 5.1 The load balancer’s throughput of various hashing algorithms with 100 servers

Figure 5.2 shows the throughput’s change with a fixed number of requests and an increasing number of servers. While throughput remains stable for most algorithms, the throughput of rendezvous hashing decreases significantly along with the increase of servers.

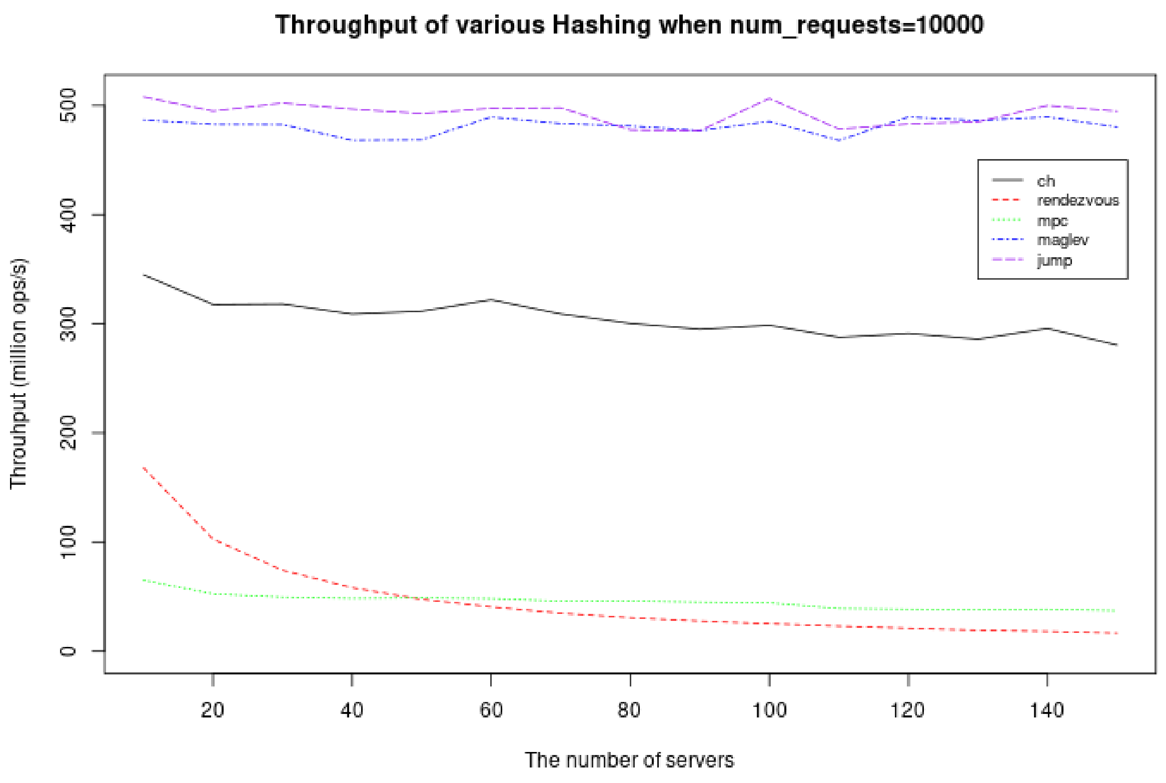

Figure 5.2 The load balancer's throughput of various hashing algorithms with 10,000 requests

5.1.2 Latency

Latency(in milliseconds) describes the time required for each request and it sums up the queueing time and processing time. It generally conforms to a long-tailed distribution. For such distribution, the longer the tail, the longer the maximum time a response takes. And the higher the peak, the more even the response time.

Figure 5.3 shows the density plot for Consistent Hashing and Rendezvous Hashing under different key distributions. The x-axis is the latency in milliseconds, and the y-axis is the proportion of value(x). For the independent variables, we assume 100 servers and 10,000 requests.

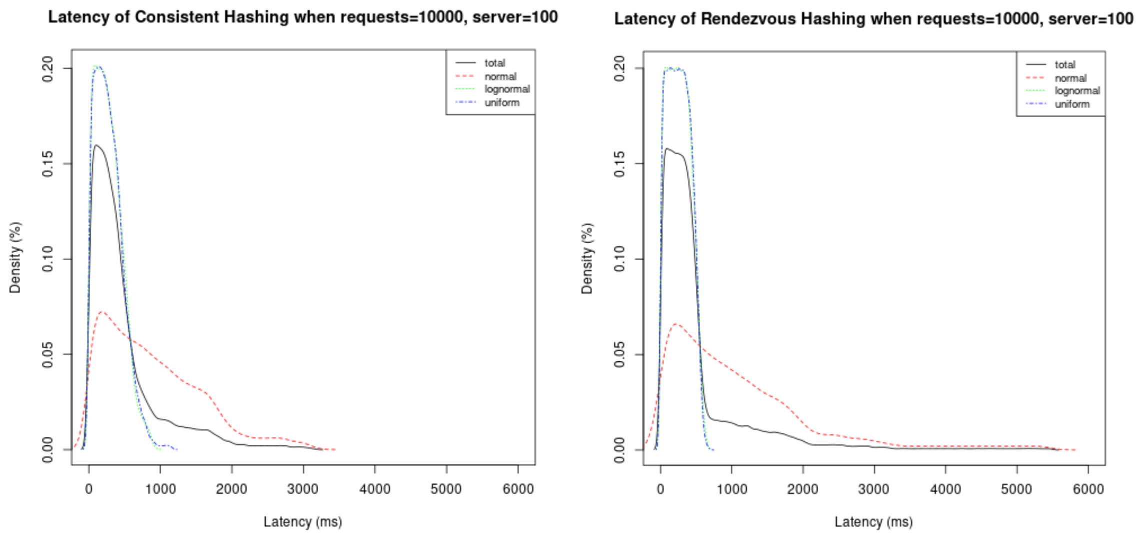

Figure 5.3 The latency density plot of Consistent Hashing and Rendezvous Hashing

These two plots are examples of latency distribution. For normally distributed keys, the requests' latency shows longer tails than the other two cases. On the other hand, the tail latency of Rendezvous Hashing is obviously higher than Consistent Hashing.

Based on these results, we integrated all the hashing algorithms together. In the following Figure 5.4, requests are uniformly and normally distributed while the number of servers and requests is 100 and 10,000 respectively. In the first case, request latency gathers around 500 ms for all algorithms while the latency diverges greatly in the second case.

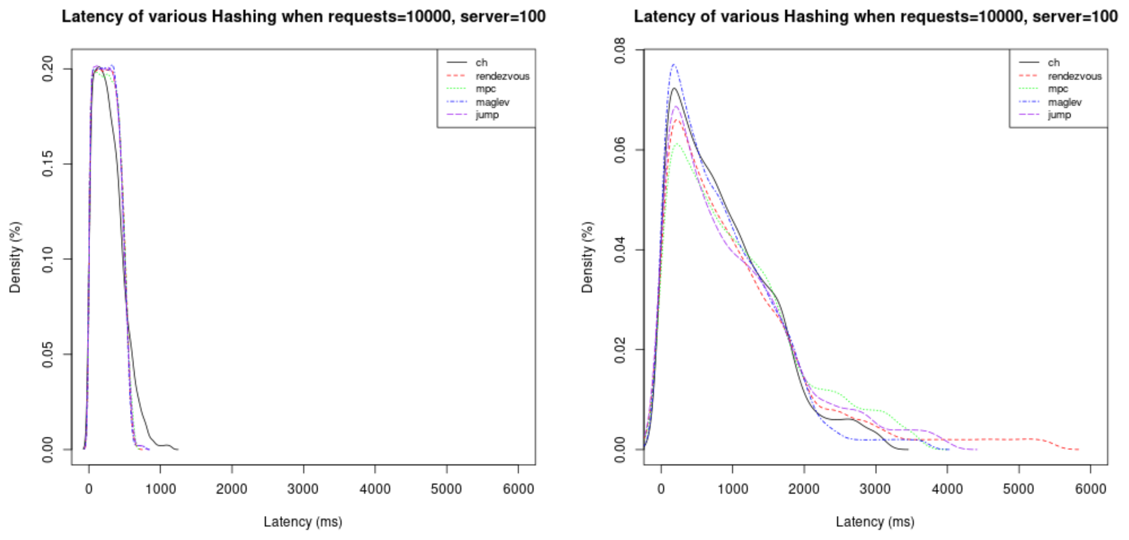

Figure 5.4 Latency Density Plot for uniform(left) and normal(right) distribution

5.1.3 Balancing Degree

To evaluate the load pressure on all nodes, we introduce a way to measure the balancing degree. Assume we have N nodes in our system, for each node, its expected load is 1/N and its actual load is recorded in the experiment. We use a variable x_i = actual_load / expected_load to measure the variance of load. Its standard error σ is defined as sqrt(Σ(x_i - μ)^2 / N), where μ stands for the expected load (1/N). From its definition, we can say that the higher the standard deviation, the more uneven the distribution of requests is. To show the trend more clearly, we assume 100 servers and increase the number of requests.

In Figure 5.5, the standard error of normally distributed requests is overall higher than the other two cases, just as expected. However, for lognormally distributed (highly skewed) requests, their standard error is close to uniformly distributed (random) requests.

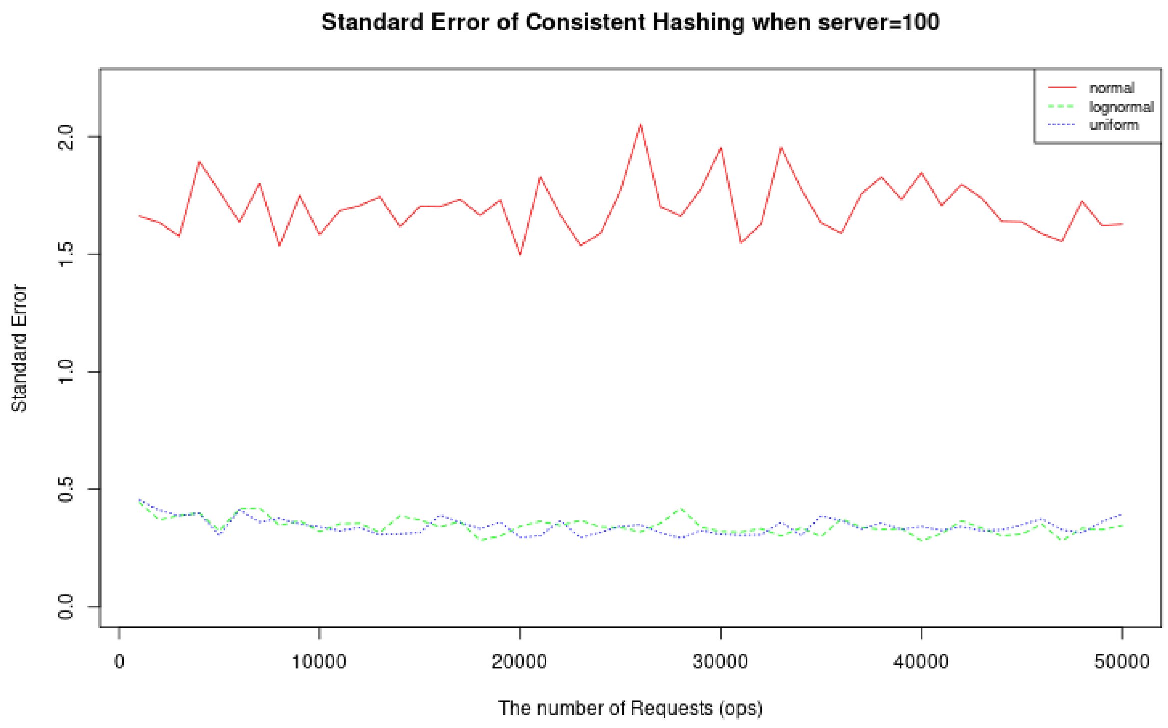

Figure 5.5 The Standard Error of Consistent Hashing with 100 servers

5.2 Analysis

5.2.1 Throughput

From Figure 5.1 we can see that the overall throughput for each algorithm remains stable regardless of the number of requests. In other words, the time needed to route a request is constant if the number of servers is fixed. This verifies the fact that the computing is done through a set of given hash functions. However, for Rendezvous Hashing, having more nodes reduces the throughput since it has to calculate a hash function value *h* for each candidate node and returns the index of the node with the highest value.

Also, we notice that with the same amount of servers, Jump Hashing and Maglev Hashing have greater throughput than Rendezvous Hashing and MPC Hashing. This difference is mainly due to their algorithmic complexity. Both Rendezvous Hashing and MPC Hashing need to compute a value *h* for each candidate node, hence its computing complexity is proportional to the number of servers. In contrast, the Jump Hashing algorithms do not need to store the information about servers and its complexity is logarithmic. Hence the throughput of Jump Hashing is higher than Rendezvous Hashing.

5.2.2 Latency

We displayed the latency distribution of two different algorithms in Figure 5.3. For both algorithms, we all observe a long-tailed distribution. In the case of the normally distributed keys, there is a lower crest and a longer tail. It means higher overall latency and long-tail latency, which shows the impact of workload imbalance caused by the skewed keys. For lognormally distributed requests, hashing function can map their keys evenly into the keyspace just like random keys. This indicates the importance of choosing an appropriate hash function with great balance ability.

Figure 5.4 shows the impact of key distribution over latency. In comparison, latency for uniformly distributed keys remains low for all algorithms. This indicates that when the keys are totally random, they are routed evenly to all nodes and every node takes nearly the same time to finish processing all requests. However, for skewed workload, the density lines of different algorithms vary greatly. For example, Rendezvous Hashing shows the longest tail latency. In contrast, Jump Hashing and Maglev Hashing have a higher peak and a shorter tail; hence, they are more resilient to skewed workloads.

5.2.3 Balancing Degree

From Figure 5.5, we can see the balancing degree of different workloads on all nodes. When the keys of requests are normally distributed, some nodes need to handle more requests than others. For totally random keys, although the distribution of requests is uniform, the load is still slightly unbalanced.

Meanwhile, the result also shows that the traditional Consistent Hashing algorithm has a good performance for lognormally and uniformly distributed keys (the blue and green lines are low and very close). We give the credit to a robust hashing function that may map unevenly distributed keys evenly into the keyspace, although this is not true for the case of a normal distribution of keys.

6 Conclusion and Future Work

Consistent hashing algorithms help to create a uniform distribution of the requests among a dynamic set of servers. This guarantee fails in the presence of skewed workloads. This paper demonstrates the impacts through three aspects: throughput, latency, and balancing degree. This comparison of available algorithm variants helps us to identify possible flaws in them. Our testbed can serve as a framework to compare algorithms in the future.

We engineered a testing key-value store with different partitioning strategies. However, in the real world, applications such as web services may adopt other feasible solutions, such as round-robin or least-connected balancing methods. And key-value databases may partition their keyspace by key range, which enables efficient range queries. A more holistic survey needs a framework where different partitioning strategies are integrated and compared. In addition, data transfer due to server crashes also needs to be considered. Moreover, experiments based on real-world data can tell us more about the feasibility of each algorithm and therefore this needs further investigation and analysis.

References

[1] C. Labovitz, S. Iekel-Johnson, D. McPherson, J. Oberheide, and F. Jahanian, ‘Internet inter-domain traffic’, in Proceedings of the ACM SIGCOMM 2010 conference, New York, NY, USA, Aug. 2010, pp. 75–86. doi: 10.1145/1851182.1851194.

[2] G. DeCandia et al., ‘Dynamo: amazon’s highly available key-value store: ACM SIGOPS Operating Systems Review: Vol 41, No 6’. https://dl.acm.org/doi/10.1145/1323293.1294281 (accessed Sep. 14, 2021).

[3] F. Chang *et al.*, ‘Bigtable: A Distributed Storage System for Structured Data’, *ACM Trans. Comput. Syst.*, vol. 26, no. 2, p. 4:1-4:26, Jun. 2008, doi: 10.1145/1365815.1365816.

[4] 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*, New York, NY, USA, May 1997, pp. 654–663. doi: 10.1145/258533.258660.

[5] L. Monnerat and C. L. Amorim, ‘An effective single-hop distributed hash table with high lookup performance and low traffic overhead’, *ArXiv14087070 Cs*, Aug. 2014, Accessed: Oct. 11, 2021. [Online]. Available: http://arxiv.org/abs/1408.7070

[6] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, ‘Workload analysis of a large-scale key-value store’, SIGMETRICS Perform. Eval. Rev., vol. 40, no. 1, pp. 53–64, Jun. 2012, doi: 10.1145/2318857.2254766.

[7] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, ‘Benchmarking cloud serving systems with YCSB’, in Proceedings of the 1st ACM symposium on Cloud computing, New York, NY, USA, Jun. 2010, pp. 143–154. doi: 10.1145/1807128.1807152.

[8] L. Suresh, M. Canini, S. Schmid, and A. Feldmann, ‘C3: cutting tail latency in cloud data stores via adaptive replica selection’, in Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation, USA, May 2015, pp. 513–527.

[9] J. Dean and L. A. Barroso, ‘The tail at scale’, Commun. ACM, vol. 56, no. 2, pp. 74–80, Feb. 2013, doi: 10.1145/2408776.2408794.

[10] V. Jalaparti, P. Bodik, S. Kandula, I. Menache, M. Rybalkin, and C. Yan, ‘Speeding up distributed request-response workflows’, SIGCOMM Comput. Commun. Rev., vol. 43, no. 4, pp. 219–230, Aug. 2013, doi: 10.1145/2534169.2486028.

[11] R. Taft et al., ‘E-store: fine-grained elastic partitioning for distributed transaction processing systems’, Proc. VLDB Endow., vol. 8, no. 3, pp. 245–256, Nov. 2014, doi: 10.14778/2735508.2735514.

[12] C. Metz, ‘How Instagram Solved Its Justin Bieber Problem | WIRED’, Nov. 11, 2015. https://www.wired.com/2015/11/how-instagram-solved-its-justin-bieber-problem/ (accessed Sep. 22, 2021).

[13] N. Venkateswaran and S. Changder, ‘Handling workload skew in a consistent hashing based partitioning implementation’, in *2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, Sep. 2017, pp. 1163–1169. doi: 10.1109/ICACCI.2017.8125999.

[14] D. G. Thaler and C. V. Ravishankar, ‘Using name-based mappings to increase hit rates’, *IEEEACM Trans. Netw.*, vol. 6, no. 1, pp. 1–14, Feb. 1998, doi: 10.1109/90.663936.

[15] J. Lamping and E. Veach, ‘A Fast, Minimal Memory, Consistent Hash Algorithm’, *ArXiv14062294 Cs*, Jun. 2014, Accessed: Sep. 14, 2021. [Online]. Available: http://arxiv.org/abs/1406.2294

[16] B. Appleton and M. O’Reilly, ‘Multi-probe consistent hashing’, *ArXiv150500062 Cs*, Apr. 2015, Accessed: Sep. 14, 2021. [Online]. Available: http://arxiv.org/abs/1505.00062

[17] V. Mirrokni, M. Thorup, and M. Zadimoghaddam, ‘Consistent Hashing with Bounded

Loads', *ArXiv160801350 Cs*, Jul. 2017, Accessed: Sep. 14, 2021. [Online]. Available: http://arxiv.org/abs/1608.01350

[18] J. Chen, B. Coleman, and A. Shrivastava, 'Revisiting Consistent Hashing with Bounded Loads', *ArXiv190808762 Cs*, Jun. 2020, Accessed: Sep. 22, 2021. [Online]. Available: http://arxiv.org/abs/1908.08762

[19] J. Chen *et al.*, 'HotRing: A Hotspot-Aware In-Memory Key-Value Store', presented at the 18th USENIX Conference on File and Storage Technologies (FAST 20), Santa Clara, CA, 2020. [Online]. Available: https://www.usenix.org/conference/fast20/presentation/chen-jiqiang

[20] K. Wu, 'Hash Rings', 2021. https://github.com/Wkkkkk/hash_rings

[21] D. E. Eisenbud *et al.*, 'Maglev: A Fast and Reliable Software Network Load Balancer', in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, Santa Clara, CA, 2016, pp. 523–535. Accessed: Sep. 14, 2021. [Online]. Available: https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/eisenbud