Benchmarking Hashing Algorithms for Load Balancing in a Distributed Database Environment

Alexander Slesarev[1,2], Mikhail Mikhailov[1], and George Chernishev[1,2(✉)]

[1] Unidata, Saint-Petersburg, Russia
alexandr.slesarev@unidata-platform.org,
{mikhail.mikhailov,georgii.chernyshev}@unidata-platform.ru
[2] Saint-Petersburg State University, Saint-Petersburg, Russia

Abstract. Modern high load applications store data using multiple database instances. Such an architecture requires data consistency, and it is important to ensure even distribution of data among nodes. Load balancing is used to achieve these goals.

Hashing is the backbone of virtually all load balancing systems. Since the introduction of classic Consistent Hashing, many algorithms have been devised for this purpose.

One of the purposes of the load balancer is to ensure storage cluster scalability. It is crucial for the performance of the whole system to transfer as few data records as possible during node addition or removal. The load balancer hashing algorithm has the greatest impact on this process.

In this paper we experimentally evaluate several hashing algorithms used for load balancing, conducting both simulated and real system experiments. To evaluate algorithm performance, we have developed a benchmark suite based on Unidata MDM—a scalable toolkit for various Master Data Management (MDM) applications. For assessment, we have employed three criteria—uniformity of the produced distribution, the number of moved records, and computation speed. Following the results of our experiments, we have created a table, in which each algorithm is given an assessment according to the abovementioned criteria.

Keywords: Consistent hashing · Databases · Benchmarking

1 Introduction

As any organization grows, the volume of its corporate data assets rises as well. There are two general approaches to solving this architectural problem [22]: vertical and horizontal scaling. Vertical scaling focuses on increasing the capabilities of a single server, whereas horizontal scaling involves adding machines to the cluster. To implement horizontal scaling, a database table has to be horizontally split into parts (shards), which are stored on different server nodes.

Horizontal scaling has several significant advantages, such as the possibility to flexibly adjust storage volume by altering cluster size. Another one is the

P. Fournier-Viger et al. (Eds.): MEDI 2022, CCIS 1751, pp. 105–118, 2022.
https://doi.org/10.1007/978-3-031-23119-3_8

ability to deal with data loss by replicating data among servers. Thus, the need for distributed data storage appears.

An important component of distributed storage is the load balancer—a mechanism which determines which particular server will store a data entity (e.g. record, table part, etc.). There are several assessment criteria for load balancers. First of all, data distribution over servers should be as close to uniform as possible. Next, if cluster size changes, the number of moved data entities must be close to optimal. And finally, load balancer computing costs should not be high.

In order to calculate the shard assigned to a given data entity, the load balancer utilizes a hashing algorithm. Since the 90's, many hashing algorithms have been designed specifically for balancing different types of loads such as network connection management, distributed computing optimization, and data storage balancing.

One of the research disciplines focusing on storage and processing of large data volumes is Master Data Management [1,19] (MDM). It is based around the concept of Master Data—a concept that combines objects important for business operations within an organization, such as inventory, customers, and employees. The main goals of MDM are unification, reconciliation, and ensuring completeness of corporate Master Data.

In this paper, we compare several hashing algorithms and assess their applicability to the load balancing problem. We experimentally evaluate them using both simulated and real tests. For the latter, we employ the Unidata platform [10]—an open-source MDM toolkit that has distributed storage capabilities. Following the results of our experiments, we have created a table, in which each algorithm is given an assessment according to the abovementioned criteria.

This paper is organized as follows. In Sect. 2 we describe a number of existing load balancing algorithms, define several terms from the MDM area, and review Unidata storage architecture. Then in Sect. 3 we describe the conducted experiments, and discuss the achieved results in Sect. 4. We conclude this paper with Sect. 5.

2 Background and Related Work

In this section we describe those existing hashing algorithms that we are going to benchmark, and since our last series of experiments is run on a real system, we also provide a general description of the system itself, its purpose, and the used data schema.

This study concerns two research fields that have a rich body of work—load balancing and hashing algorithms. The former has a large number of surveys describing dozens of works. For example, consider study [8], which references many more similar surveys. Unfortunately, such surveys only classify the covered methods using some high-level criteria (e.g. adaptivity, static or dynamic, preemptiveness, etc.). They do not experimentally evaluate surveyed algorithms.

The reason for this is the fact that such surveys are too broad, their reviewed studies belong to many different fields and it will be extremely difficult to run such an evaluation. At the same time, industry is interested in the best method for a particular domain, and the answer can be found only empirically.

Turning to hashing, we must mention a very comprehensive survey [3], which describes many hashing approaches, and proposes an algorithm taxonomy. However, the section that concerns data-oriented hashing is aimed towards data structures and machine learning, but not load balancing. The set of hashing algorithms that we consider in our study is absent in this survey.

Therefore, our work fills the gap in existing studies.

2.1 Considered Methods

Let us start with the hashing algorithms which are used for data balancing. Each of the considered load balancers applies its hash function to some incoming data entity, so we call the result of the application a *key*. The purpose of a load balancer is to match each key to one of the shards, which is represented by an integer number (id). There are several hashing methods that we consider in this paper:

- **Linear Hashing** is one of the overall oldest algorithms. Besides the classic version [2] there is a number of modifications such as LH* [18], LH*M [14], LH*G [16], LH*S [15], LH*SA [13] and LH*RS [17]. The core idea of algorithms of this family is to calculate the remainder of dividing the key by a number of shards in the system. Therefore, they are suitable for solving the problem featuring a fixed number of shards, which in our case is a disadvantage. In our work we have adopted a version used for partitioning in PostgreSQL[1].
- **Consistent Hashing.** Originally, this algorithm [9] was designed for balancing loads of computer networks. Nowadays, it seems to be the most popular method for balancing many various types of loads. For example, distributed systems like AWS DynamoDB [4] and Cassandra [11] use Consistent Hashing for partitioning and replication. This method is based on picking random points on a ring, which is a looped segment of real numbers that represents shards and data entities as points. Points denoting keys are assigned to the clockwise nearest shard. To ensure even data distribution, each shard is represented by several points. Note that the design of this method allows to change the number of shards while moving only the optimal number of records.
- **Rendezvous.** Similarly to Consistent Hashing, this method [21] was also developed to optimize network load. For a given key, the algorithm calculates the value of the cost function for each of the shards and assigns the key to the shard with the highest value. When adding or removing shards, Rendezvous also does not move extra records.

[1] https://github.com/postgres/postgres/blob/master/src/backend/partitioning/partbounds.c.

- **RUSH** [6] was developed to store data in a disk cluster. It has two modifications: RUSH_R and RUSH_T [7]. Authors of RUSH focused on improving the uniformity of data distribution when changing the size of the cluster, therefore, the algorithm is based on the following principle: every time cluster size is changed, a special function is used to decide which objects should be moved to balance the system.
- **Maglev** is an algorithm [5] from Google's load balancer for web services. The goal of Maglev is to improve data uniformity (compared to Consistent Hashing) and cause "minimal disruption", e.g. if the set of shards changes, data records will likely be sent to the same shard where they were in before. It was proposed as a new type of Consistent Hashing, in which the ring is replaced with a lookup table by which a key can be assigned to a shard. The size of the lookup table should be greater than the possible number of shards to decrease collision rate. Average proposed lookup time is $O(Mlog(M))$, where M is size of lookup table.
- **Jump** is another load balancer from Google [12]. Its authors presented it as a superior version of Consistent Hashing which "requires no storage, is faster, and does a better job of evenly dividing the key space among the buckets". Jump generates values for shard numbers only in the $[0; \#shards]$ range, so the addition of a new shard is fast. However, a deletion of an intermediate shard will cause rehashing of many records. It takes $O(log(N))$ time to run Jump, where N is the number of shards.
- **AnchorHash.** According to the authors, AnchorHash [20] is a "hashing technique that guarantees minimal disruption, balance, high lookup rate, low memory footprint, and fast update time after resource additions and removals". A notable difference of AnchorHash from other discussed algorithms is that it stores some information about previous states of the system.

Each paper that proposed a novel hashing algorithm compared it only to a small number of other such algorithms. To the best of our knowledge, there were no dedicated comparisons of such algorithms applied to the horizontal scaling problem. At the same time, ensuring high performance of horizontal scaling is a pressing problem that is in demand by the industry. Thus, there is a need to evaluate all of these algorithms and study their applicability to this problem.

2.2 Basic Definitions

To understand the specifics of the data storage in which the load balancers will be evaluated, it is necessary to introduce some MDM terms:

- **Golden Record.** One of the main problems in MDM is the compilation and maintenance of a "single version of the truth" [1] for a given entity, e.g. person, company, order, etc. To achieve such a goal, an MDM system has to assemble information from many data sources (information systems of a particular organization) into one clean and consistent entity called the golden record.

– **Validity Period** is a time interval in which the information about an entity is valid. For each golden record, several validity periods may exist. This fact should be taken into account while querying the data. There are two temporal dimensions: time of an event and time of introduction of this new version of information to the system. This leads to a special storage scheme for managing this information.

2.3 System Architecture

MDM systems are a special class of information management systems [10]. Their specifics impose requirements on the platform storage architecture and data processing.

First of all, versioning of stored objects must be supported. For this reason, data assets describing stored objects have validity periods and they should be taken into account while querying the data.

Second, deletion operations can be performed only by an administrator, while a user can only mark data entity as removed. This is necessary to avoid information loss and ensure correct versioning support. Sometimes there are legal requirements for this data handling semantics. Such an architectural pattern is often called tombstone delete.

Third, provenance should be provided. It means that any system operation should be traceable. For example, there must be a way to roll back all changes in records after each operation.

The proposed approach is based on the following four tables, where three of them represent entities:

– Etalon stores the metadata of the golden record itself.
– Origin stores the metadata related to the source system of the record.
– Vistory (version history) is the validity period of origin, which in turn may have revisions.
– External Key is a table needed for accessing the data from within other parts of the Unidata storage.

The relations between these tables are shown in Fig. 1. The links with empty arrowheads denote “shared” (inherited) attributes and full arrowheads show the PK-FK relationship. The detailed descriptions of table attributes can be found in [10].

3 Evaluation

In order to select the best hashing algorithm for the load balancing problem, we have performed an experimental evaluation.

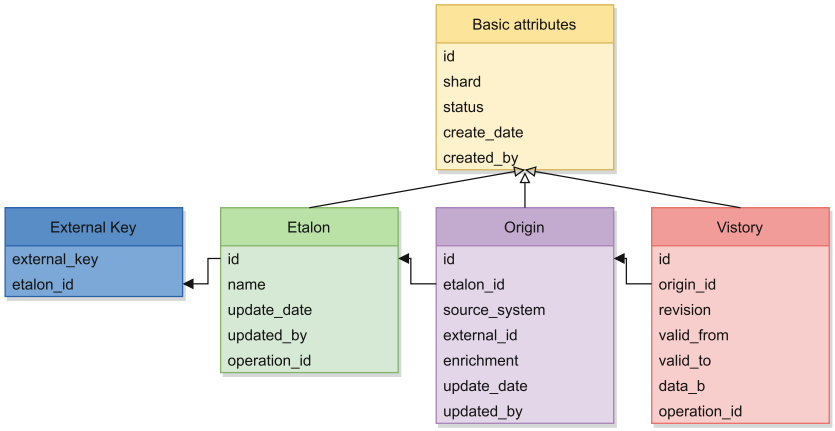


Fig. 1. Tables used for data storage in the Unidata platform

3.1 Experimental Setup

Experiments were performed using the following hardware and software configuration:

- Hardware: LENOVO E15, 16GiB RAM, Intel(R) Core(TM) i7-10510U CPU @ 4.90 GHz, TOSHIBA 238GiB KBG40ZNT.
- Software: Ubuntu 20.04.4 LTS, Postgres 11.x, JDK 11.x, Tomcat 7.x, Elasticsearch 7.6.x.

Some algorithms have parameters that affect their performance:

- For Consistent Hashing, 16 points for each shard were selected. This number was chosen experimentally, as a compromise between the hashing speed and uniformity of the initial distribution.
- For Maglev, lookup table size was set to 103. Similarly to Consistent Hashing, this number was selected experimentally. Note that this value is important in rebalancing process, but not for lookup.
- For AnchorHash, we have set the $|\mathcal{A}|$ (the number of buckets that algorithm works with) to double the maximum number of shards (i.e. 64) as it was recommended in the original paper [20].

3.2 Results

In order to evaluate the load balancing algorithms, we have defined three criteria which we ranked by their importance (in descending order):

1. Uniformity of the produced data distribution.
2. Redundant movement of records during shard addition or removal.
3. Lookup speed.

To select the best algorithm, we have conducted three experiments. First, we have performed a load balancing simulation experiment in Google Colab[2] (in Python). This step was needed to run a shallow, preliminary assessment of algorithm performance. It was conducted as follows: first, 10K records were generated and distributed (via hashing function) into 32 shards. Thus, uniform distribution will result in 312 records per shard. After this, 8 shards were scheduled for removal and system was forced to rebalance the data. Therefore, uniform distribution should result in 416 records per shard. This procedure was run for each of the considered load balancers (hashing functions).

The mean values of ten such experiments are presented in Table 1. The first column of the table contains the average time of shard calculation, the second and the third present variance of records assigned to shards before and after rebalancing, respectively. The last column shows the ratio of the number of actually moved records to the optimal number.

Table 1. First experiment, simulation in Colab

Algorithm	Shard id calculation time (ns)	Variance before drop	Variance after drop	Moved records ratio
Consistent	55049	72	94	1.00
Rendezvous	105331	18	21	1.00
RUSH_R	547044	95	125	1.57
Maglev	1146	16	21	1.39
Jump	18077	21	29	3.63
AnchorHash	3539	17	20	1.00

Based on the experimental results, we have decided to exclude RUSH_R from further consideration due to it failing to conform to all three criteria. We have also excluded Jump due to the poor quality of rebalancing.

Our next experiments involve the Unidata platform, which is implemented in Java. To verify the consistency and transferability of previously obtained results, we have decided to re-evaluate the shard id calculation time inside the platform. Therefore, the second experiment was also to distribute 10K records among shards. The measured averages were as follows:

- Linear Hashing—808 ns
- Consistent Hashing—2419 ns
- Rendezvous—5945 ns
- Maglev—807 ns
- AnchorHash—2015 ns

[2] https://colab.research.google.com/drive/1pbJUFFP9JsSTSn7nrWv0tYdiUg2uRxAv?usp=sharing.

As one can see, the order of algorithm run times has not changed compared to the previous experiment. Therefore, we can conclude that switching the programming language did not affected the results of the previous experiment and we can continue using the Unidata platform.

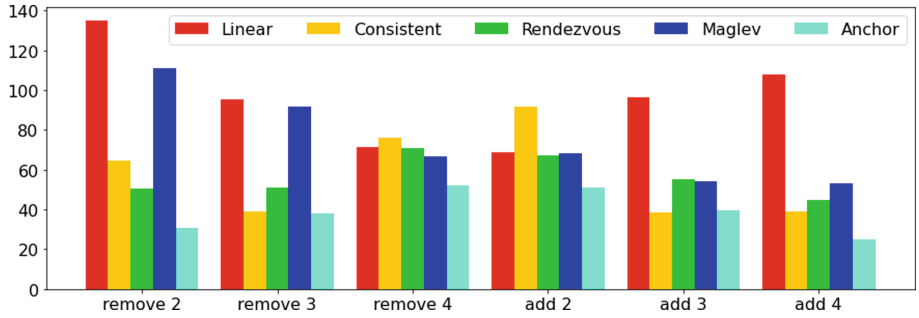


Fig. 2. Rebalance time (seconds).

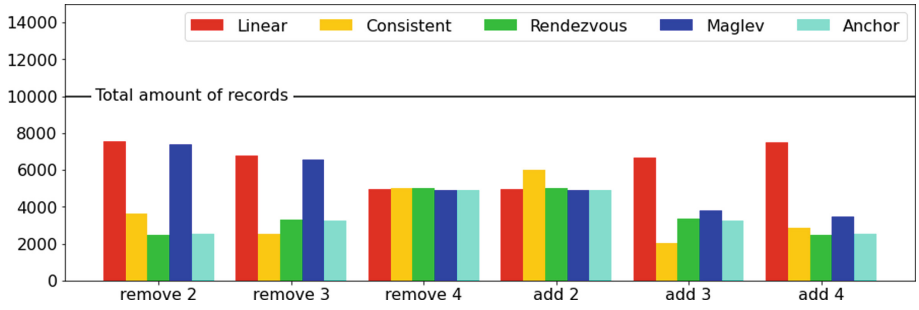
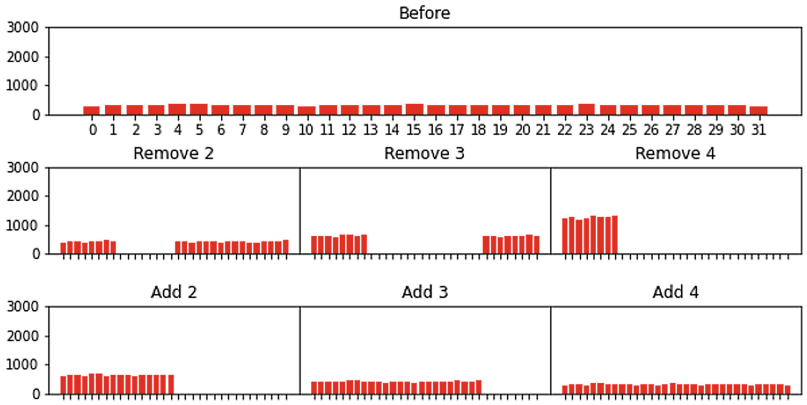


Fig. 3. Number of records from the Etalon table that were moved in the rebalancing process.

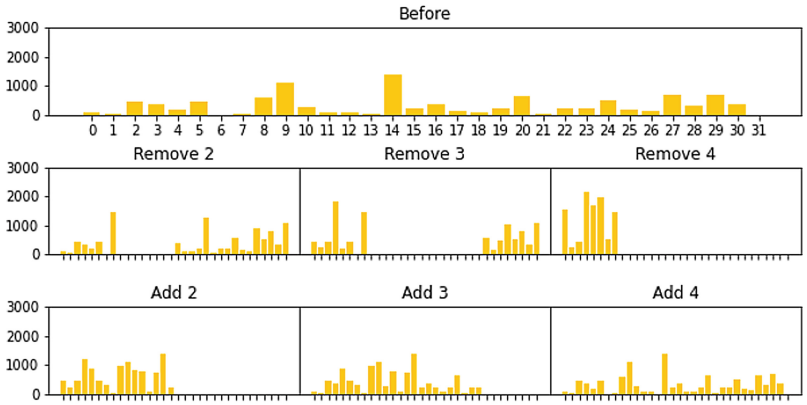
The third experiment was performed using a deployed Unidata platform. Its storage configuration was the following: four Docker nodes with PostgreSQL with eight shards on each node. We have generated 10K external keys and etalons as a workload. The idea of the experiment was as follows: remove three nodes one by one and then add them back in a similar manner.

The results of the evaluation are displayed in the following figures. Total time spent on each rebalance step is shown in Fig. 2, and the number of moved etalons is shown in Fig. 3. We have omitted such figure for external keys since it is largely the same (it is 1:1 mapping). Data distribution among shards on each step is shown in Figs. 4 and 5.

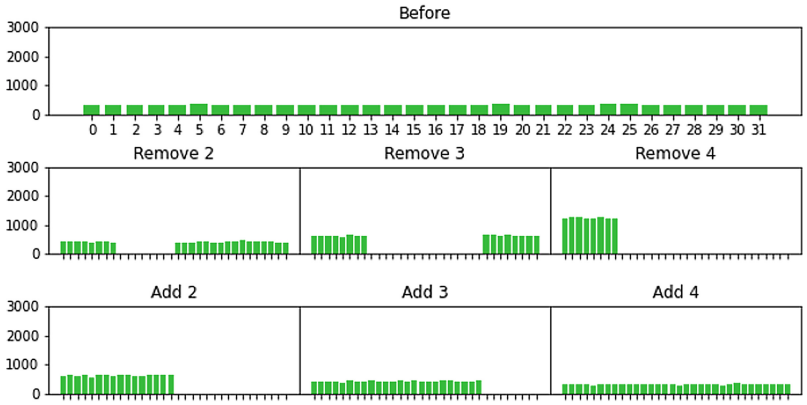
This experiment allows us to draw the following conclusions:



(a) Linear Hashing.

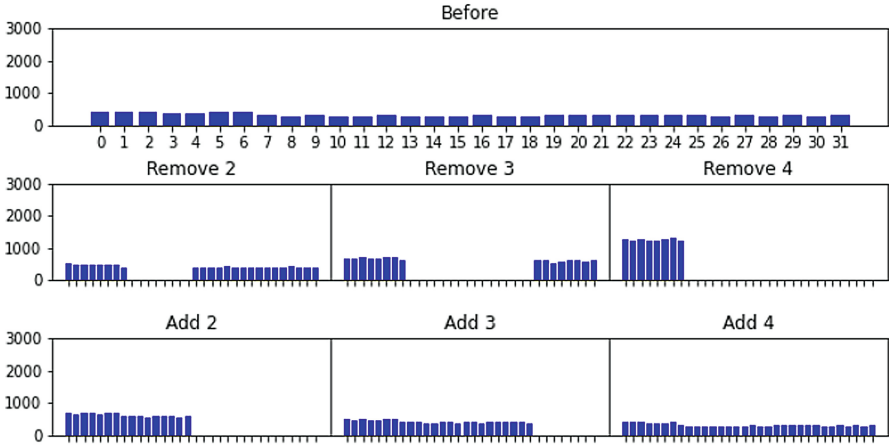


(b) Consistent Hashing.

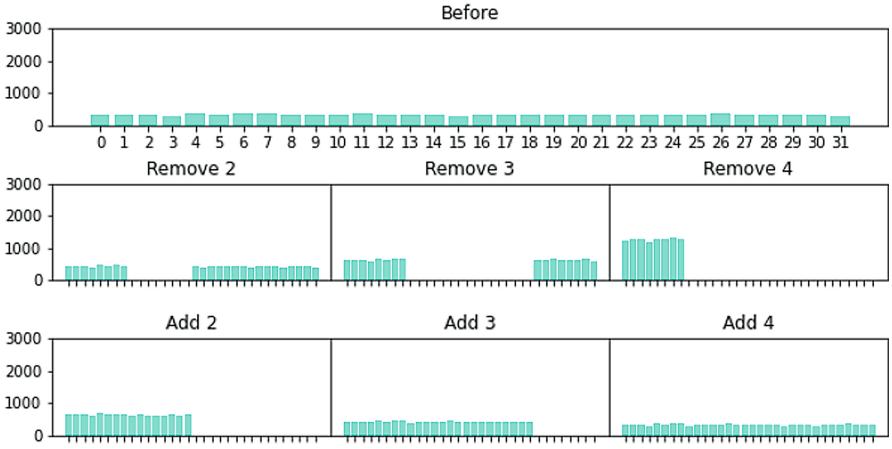


(c) Rendezvous.

Fig. 4. Distribution of Etalon table records among shards.



(a) Maglev.



(b) AnchorHash.

Fig. 5. Distribution of Etalon table records among shards.

– Consistent Hashing, Rendezvous and AnchorHash move more than 50% records less than Linear Hashing.
– During the first two rebalancing steps Maglev moved approximately the same number of records as Linear Hashing, but on the last two steps Maglev moved significantly less records and got close to the other three methods.
– Linear Hashing, Rendezvous, Maglev and AnchorHash distribute data uniformly enough, but Consistent Hashing has significant differences in shard volumes.

4 Discussion

Let us now discuss the compliance of each of the considered algorithms with the criteria defined in Sect. 3.

- **Linear Hashing** has proper record distribution among shards and high lookup speed, but it moves up to 80% of records on each rebalancing step, so this method does not meet our criteria. However, Linear Hashing can be applied in systems with a constant number of shards.
- **Consistent Hashing** has acceptable lookup times while moving an optimal number of records, but distributes data extremely unevenly. Methods with more uniform distribution should be preferred. To improve the distribution quality, one can increase the number of points for each shard on the ring, but this will slow down lookups.
- **Rendezvous** is optimal in case of data rebalancing and distribution, but has the longest lookup time. Since lookup speed is the least prioritized criterion, this method is suitable for us.
- **RUSH$_\mathbf{R}$** does not satisfy all three criteria, so it does not suit our goals.
- **Maglev** provides fast lookup and relatively uniform distribution, but in some cases it can move more than 50% of all records (see the horizontal line on Fig. 3). Therefore, Maglev is appropriate for systems with a fixed number of shards.
- **Jump** moved the largest number of records (Table 1), so it is unsuitable as well.
- **AnchorHash** appears to be the winner so far as it satisfies all the requirements.

Following the results of all three experiments, we have created a table where we listed all evaluated algorithms (Table 2). We have assessed them according to our three criteria and assigned a rating out of three quality grades—low, medium, and high.

It is evident from the table that there are two winning algorithms—Maglev and AnchorHash, which fail to reach either top rebalancing quality (number of moved records) or the top lookup speed.

AnchorHash distributes data uniformly, moves an optimal amount of records and its lookup time is small enough. Rendezvous also fits first and second criteria, but its lookup time is more than two times larger than that of AnchorHash. These two methods are appropriate for systems with frequent shard addition or removal.

On the other hand, Maglev's lookup is more than two times faster, therefore it is suitable for static systems, similarly to Jump and Linear Hashing.

Consistent Hashing seems to be effective for both types of systems, but its main drawback is non-uniform data distribution among shards.

RUSH$_\mathrm{R}$ has been proven to be the worst algorithm out of all.

Table 2. Load balancer criteria satisfaction table

Algorithm	Data distribution uniformity	Rebalancing quality	Lookup speed
Linear	High	Low	High
Consistent	Low	High	Medium
Rendezvous	High	High	Low
RUSH	Low	Low	Low
Maglev	High	Medium	High
Jump	Medium	Low	Medium
AnchorHash	High	High	Medium

5 Conclusion and Future Work

In this paper we have studied several hashing algorithms and assessed their applicability to data balancing in distributed databases. For this, we have performed both simulated and real experiments. The real experiments were run using the Unidata platform, an open-source tookit for building MDM solutions. In these experiments we have employed three criteria of applicability, namely uniformity of produced data distribution, the amount of moved records, and computation costs.

Experiments demonstrated that out of seven considered algorithms there are two clear winners—AnchorHash and Maglev. Another two, Linear Hashing and Jump, may have some applicability too.

There are several possible directions for extending this paper. First of all, we noticed some impact of the random number generator on the behavior of some algorithms. In the current paper, we fixed it for all algorithms, but it may be worthwhile to explore this influence. Secondly, it may be interesting to study the impact of varying the parameters of the algorithms. In this paper we have used either the default or the recommended ones, but it is possible that careful tuning may yield positive results.

Acknowledgments. We would like to thank Anna Smirnova for her help with the preparation of the paper.

References

1. Allen, M., Cervo, D.: Multi-Domain Master Data Management: Advanced MDM and Data Governance in Practice, 1st edn. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2015)
2. Alon, N., Dietzfelbinger, M., Miltersen, P.B., Petrank, E., Tardos, G.: Linear hashing. Technical report (1997)
3. Chi, L., Zhu, X.: Hashing techniques: a survey and taxonomy. ACM Comput. Surv. **50**(1), 1–36 (2017). https://doi.org/10.1145/3047307
4. DeCandia, G., et al.: Dynamo: amazon's highly available key-value store. SIGOPS Oper. Syst. Rev. **41**(6), 205–220 (2007). https://doi.org/10.1145/1323293.1294281

5. Eisenbud, D.E., et al.: Maglev: a fast and reliable software network load balancer. In: Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation, pp. 523–535. NSDI'16, USENIX Association, USA (2016)
6. Honicky, R., Miller, E.: A fast algorithm for online placement and reorganization of replicated data. In: Proceedings International Parallel and Distributed Processing Symposium, p. 10 (2003). https://doi.org/10.1109/IPDPS.2003.1213151
7. Honicky, R., Miller, E.: Replication under scalable hashing: a family of algorithms for scalable decentralized data distribution. In: 18th International Parallel and Distributed Processing Symposium, 2004. Proceedings, pp. 96 (2004). https://doi.org/10.1109/IPDPS.2004.1303042
8. Jafarnejad Ghomi, E., Masoud Rahmani, A., Nasih Qader, N.: Load-balancing algorithms in cloud computing: a survey. J. Netw. Comput. Appl. **88**, 50–71 (2017). https://doi.org/10.1016/j.jnca.2017.04.007
9. Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., Lewin, D.: 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, pp. 654–663. STOC 1997, Association for Computing Machinery, New York, NY, USA (1997). https://doi.org/10.1145/258533.258660
10. Kuznetsov, S., et al.: Unidata – a modern master data management platform. In: Proceedings of the 1st International Workshop on Data Platform Design, Management, and Optimization (DATAPLAT) co-located with the 25th International Conference on Extending Database Technology and the 25th International Conference on Database Theory (EDBT/ICDT 2022), Edinburgh, UK, March 29, 2022. CEUR Workshop Proceedings, CEUR-WS.org (2022)
11. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. SIGOPS Oper. Syst. Rev. **44**(2), 35–40 (2010). https://doi.org/10.1145/1773912.1773922
12. Lamping, J., Veach, E.: A fast, minimal memory, consistent hash algorithm (2014)
13. Litwin, W., Menon, J., Risch, T.: Lh* schemes with scalable availability (2001)
14. Litwin, W., Neimat, M.A.: High-availability LH* schemes with mirroring. In: Proceedings of the First IFCIS International Conference on Cooperative Information Systems, p. 196. COOPIS 1996, IEEE Computer Society, USA (1996)
15. Litwin, W., Neimat, M.A., Lev, G., Ndiaye, S., Seck, T.: LH*s: a high-availability and high-security scalable distributed data structure. In: Proceedings Seventh International Workshop on Research Issues in Data Engineering. High Performance Database Management for Large-Scale Applications, pp. 141–150 (1997). https://doi.org/10.1109/RIDE.1997.583720
16. Litwin, W., Risch, T.: LH*g: a high-availability scalable distributed data structure by record grouping. IEEE Trans. Knowl. Data Eng. **14**(4), 923–927 (2002). https://doi.org/10.1109/TKDE.2002.1019223
17. Litwin, W., Moussa, R., Schwarz, T.: $LH*_{RS}$-a highly-available scalable distributed data structure. ACM Trans. Database Syst. **30**(3), 769–811 (2005). https://doi.org/10.1145/1093382.1093386
18. Litwin, W., Neimat, M.A., Schneider, D.A.: LH: linear hashing for distributed files. In: Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, pp. 327–336. SIGMOD 1993, Association for Computing Machinery, New York, NY, USA (1993). https://doi.org/10.1145/170035.170084
19. Loshin, D.: Master Data Management. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2009)

20. Mendelson, G., Vargaftik, S., Barabash, K., Lorenz, D.H., Keslassy, I., Orda, A.: Anchorhash: a scalable consistent hash. IEEE/ACM Trans. Netw. **29**(2), 517–528 (2021). https://doi.org/10.1109/TNET.2020.3039547
21. Thaler, D., Ravishankar, C.: A Name-Based Mapping Scheme for Rendezvous. Technical report. https://www.eecs.umich.edu/techreports/cse/96/CSE-TR-316-96.pdf
22. Özsu, M.T., Valduriez, P.: Principles of Distributed Database Systems, 3rd edn. Springer, Cham (2011)