Toward Fast Query Serving in Key-Value Store Migration with Approximate Telemetry

Alexander Braverman
Seven Lakes High School
Katy, Texas

Zaoxing Liu
University of Maryland
College Park, Maryland

ABSTRACT

Distributed key-value stores scale data analytical processing by spreading data across nodes. Frequent migration of key-value shards between online nodes is a key technique to react to dynamic workload changes for load balancing and service elasticity. During migration, the data is split between a source and a destination, making it difficult to query the exact location. Existing solutions aiming to provide real-time read and write query capabilities during migration may require querying both source and destination servers, doubling the compute/network resources. In this paper, we explore a simple yet effective measurement approach to track the key-value migration status, in order to improve the query-serving performance under migration. In our preliminary prototype, we use a Bloom filter on the destination server to keep track of individual key-value pairs that have been successfully migrated. For key-value pairs that have yet migrated, the information stored in the Bloom filter enables fast forwarding to the source server without the need to check the database. We prototype this design on a local cluster with Redis deployments. Our preliminary results show that this approximate measurement-based design minimizes query losses during migration.

1 Introduction

Modern cloud services (e.g., e-commerce, mobile gaming, and social networking) depend on large-scale key-value stores as the backend to perform various kinds of jobs (e.g., content caching, real time analytics and machine learning) [16, 3]. These services often require backend databases to process requests over ever-growing data volumes and dynamic workload distributions. However, static sharding limits the ability of such systems to adapt to rapidly changing workloads. This can result in degraded performance and Service Level Agreement (SLA) violations due to load imbalance and insufficient provisioning of cloud resources [8, 9].

To tackle the problem of imbalance of load and resources, a variety of key-value migration techniques are adopted [8, 9, 12] to efficiently migrate data between nodes (i.e., the source and destination servers). However, the migration process itself is often time-consuming, and the actual query serving performance varies depending on the workload distribution and the migration progress. During migration, the client is unknown about which keys have reached their destination (migrated) at any given time without actually accessing them on the relevant databases. Therefore, it is difficult for the client to always query certain keys at the *right* location because the client is unsure of the current location of the queried key. To ensure query serviceability, a straightforward solution is to query both the source and destination servers, incurring doubled overheads for the client. Alternatively, the client needs to access a database that records the migrated keys, but such maintaining such a database is resource-heavy (e.g., network, compute, and storage), since the number of keys can be prohibitively large. Ideally, the client should know where to access the queried key in the right location without expensive bookkeeping and without actually reaching the key-value store hosted on the servers during the migration process.

To this end, we explore and leverage approximate telemetry approaches to track the status (e.g., "not started", "in transmission", or "completed") of the key-value pairs during migration, allowing client requests to be served at the right location as soon as possible. Approximate measurement design brings a major benefit to serving client queries in key-value store migration: the underlying data structures used in the design are often probabilistic and require only sublinear resources, such as sketches [7, 14] and Bloom filters. Such resource savings enables wider adoption of approximate telemetry on resource-constrained devices. Using these devices to track migration status, including SmartNICs [1] and programmable switches [5], can direct client requests to the appropriate location as early as possible. Thus, in this preliminary work, we deploy a Bloom filter on the Redis server to track migration progress and evaluate potential query-serving performance improvements.

We implement a key-value migration protocol with a Bloom filter deployed on the destination server. We deploy a large Redis key-value store [2] (100GB) to migrate from one commodity server to another. We evaluate several experimental scenarios with client workloads following Zipf distributions [13] with varied write ratios in client requests. Our preliminary results demonstrate that using the Bloom filter to track key-value migration status can significantly reduce query mishits (defined as query loss rate).

2 Preliminary Design and Prototype

Our workflow to perform key-value migration consists of four main components: (1) the migration process of Redis key/value pairs from a source server to a destination server, (2) a Bloom filter that identifies whether a key-value pair has migrated successfully and if a query is missed in the local database, the filter tells the client where to forward the missed requests, (3) a client that sends read and write requests to the destination server, and (4) a forwarding mechanism which allows requests to be served before the respective key reaches the destination server. Below, we provide the details of each component using Redis [2] as an example.

- **Migration**: The migration process [9] consists of extracting all the key/value pairs from a Redis instance of the source server and sending them to the destination server and storing them in its Redis store. To allow integration of Bloom filters, we reimplement a UDP-based migration protocol to extract Redis key value pairs from source to destination by extending the implementation of DistCache [13].

- **Bloom filter**: During the migration process, when an individual key-value pair reaches the destination server and is updated in the Redis store, this key will be added to a Bloom filter [4] on the destination side to keep track of which key-value pairs have already successfully migrated. Employing a simple Bloom filter at the destination has two benefits: (1) It does not have false positives and can be adjusted to achieve relatively low false negative rates as shown in Figure 1. No false positives ensure that queries to the migrated keys will never go back to the source server. (2) The memory efficiency of the Bloom filter makes it possible to serve as a "cache". This additional saving of space comes from the fact that we only store keys and not values in the bloom filter.

- **Serving client requests**: When the client queries certain key/value pairs currently in the destination, the destination server will first check whether this key has reached the server by probing the Bloom filter. The advantage of using a probabilistic filter over directly accessing Redis to see if a key-value pair has already migrated is that the Bloom filter is small in space and allows for faster (parallel) memory accesses. If the key has migrated, we know for sure that it is in the filter, and we can access the Redis store locally and directly respond to the client.

- **Request forwarding**: If a key is queried by the client, which is not already in the Bloom filter, we know it still has not reached the destination server. Therefore, the destination server can simply forward [6] the client request to the original sending server, who will be able to serve the request on its own local Redis instance and respond directly to the client.

3 Experiments

In our evaluation, every experiment conducted consisted of using 50 million Redis key-value pairs for migration, which held 100 GB of data. Our experiments run on a testbed consisting of 3 servers, each of which has two Intel Xeon Gold 5317s, 512GB DRAM, and a 10G NIC. Specifically, each key-value pair held 2000 bytes, the keys ranging from $1 - 50,000,000$, and the values being a randomly generated string with 1992 bytes to 1999 bytes of data depending on the length of its respective key. Every experiment we run generates 750,000 queries from the client. Furthermore, we use different Zipfian distributions for our experiments, such

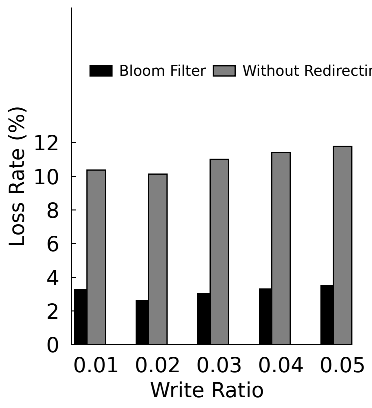

Figure 1: Query loss rates with varying write ratios.

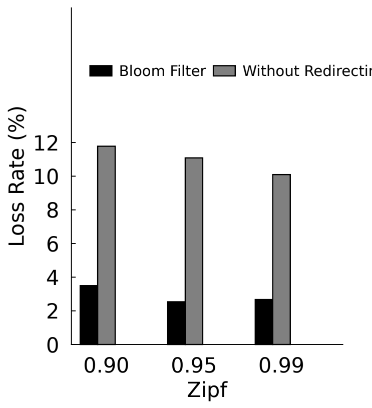

Figure 2: Query loss rates with varying Zipf distributions.

as .9, .95, and .99. Second, we also use different write ratios, such as .01 to .05. We evaluate the loss rates when the queries go to the wrong locations without being forwarded to the right places for Bloom filter-based vs. no filter-based.

To evenly split the migration data and serve client requests in parallel, we used 10 threads for both the source and destination servers. Moreover, the data on the destination server is randomly split amongst 5 Redis servers, to mitigate the loss due to the large volume of migrated data. On the destination server, the Bloom filter implementation [17] we used has 125 MB of data stored overall in the data structure.

3.1 Effect of Query Distribution

Our first experiment revolves around the effect of the write ratio of queries to the loss rate. This experiment on average takes 14 minutes and 39 seconds across all write ratios. The client issues key read/write queries following a Zipfian distribution of 0.9 skewness with write ratios of 0.01 to 0.05 as shown in Figure 1. The inclusion of Bloom filter and request forwarding demonstrates a 3.5 $\times$ improvement in mitigating query loss on average (from 3.162 to 3.870).

3.2 Effect of Write Ratio

Our second experiment measures the effect of how the skewness in Zipf distribution affects the loss rate. This experiment on average takes 14 minutes and 38 seconds across all Zipf distributions tested (0.9, 0.95, 0.99) and we use the write ratio of 0.05. As shown in Figure 2, the inclusion of the forwarding of requests results in a 3.837 × improvement in mitigating query loss on average (from 3.367 to 4.367).

3.3 False Negatives in Bloom Filter

One of the main concerns of using bloom filters is the false negatives. In our key-value migration case, false negatives occur when the key is shown to be in the Bloom filter, but in reality it has yet been migrated to the destination. In each experiment, we calculate the number of false negatives by checking when the Redis server returned *null*. In our experiments, the number of such false negatives is always less than 0.043 percent of all queries. Thus we can conclude that with a relatively large filter (e.g., 150MB), false negatives have negligible impacts on the average query-serving performance.

3.4 Evaluation Summary

Overall, these experimental results demonstrate the need for designing a proper measurement approach to using forwarding, as without the bloom filter, there is a notable drop in performance that could prove costly in a real setting. Furthermore, as the results demonstrated the positive effect of forwarding, using higher power devices such as SmartNICs [1, 15] could prove to be even more beneficial [10] in serving client queries during migration.

4 Discussion

We conclude by highlighting a subset of new opportunities for further research in this space.

Approximate telemetry for key-value migration on near-user programmable devices. Our prototype demonstrates the benefits of tracking the detailed migration status with a Bloom filter-based design. However, for simplicity, the filter for tracking the migration progress is deployed on the server side. Every client request needs to pay the cost of reaching the server first before it can be directed to the right location. We posit that, with emerging flexibility and programmability in the network devices (e.g., SmartNICs and programmable switches), we can find a vantage point in the network to easily measure how key-value pairs are being migrated while not being far from the client.

Adaptive filters for improving the false negative rates. The current prototype is limited to the standard Bloom filter with fixed false negatives. While our experiments show small performance degradation on average, the false positives can incur performance problems at the tail (e.g., tail query latency for some keys). Recent advances in adaptive filters [11] have shown some practical designs to dynamically tune the filters to reduce and control the error rates if we have seen such false negatives so far.

5 References

[1] Bluefield data processing units. `https://www.nvidia.com/en-us/networking/products/data-processing-unit/`.

[2] Redis. `https://redis.io/`.

[3] Redis use cases. `https://redis.com/blog/5-industry-use-cases-for-redis-developers/`.

[4] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, jul 1970.

[5] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. *ACM SIGCOMM Computer Communication Review*, 43(4):99–110, 2013.

[6] J. Chen, P. Druschel, and D. Subramanian. An efficient multipath forwarding method. In *Proceedings. IEEE INFOCOM '98*, pages 1418–1425, 1998.

[7] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[8] J. Kang, L. Cai, F. Li, X. Zhou, W. Cao, S. Cai, and D. Shao. Remus: Efficient live migration for distributed databases with snapshot isolation. In *Proceedings of the 2022 International Conference on Management of Data*, pages 2232–2245, 2022.

[9] C. Kulkarni, A. Kesavan, T. Zhang, R. Ricci, and R. Stutsman. Rocksteady: Fast migration for low-latency in-memory storage. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 390–405, 2017.

[10] Y. Le, H. Chang, S. Mukherjee, L. Wang, A. Akella, M. M. Swift, and T. V. Lakshman. Uno: Uniflying host and smart nic offload for flexible packet processing. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 506–519, New York, NY, USA, 2017. Association for Computing Machinery.

[11] D. J. Lee, S. McCauley, S. Singh, and M. Stein. Telescoping filter: A practical adaptive filter. *arXiv preprint arXiv:2107.02866*, 2021.

[12] Y.-S. Lin, S.-K. Pi, M.-K. Liao, C. Tsai, A. Elmore, and S.-H. Wu. Mgcrab: transaction crabbing for live migration in deterministic database systems. *Proceedings of the VLDB Endowment*, 12(5):597–610, 2019.

[13] Z. Liu, Z. Bai, Z. Liu, X. Li, C. Kim, V. Braverman, X. Jin, and I. Stoica. Distcache: Provable load balancing for large-scale storage systems with distributed caching. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 143–157, 2019.

[14] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 101–114, 2016.

[15] H. Seyedroudbari, S. Vanavasam, and A. Daglis. Turbo: Smartnic-enabled dynamic load balancing of µs-scale rpcs. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1045–1058, 2023.

[16] R. K. Singh and H. K. Verma. Redis-based messaging queue and cache-enabled parallel processing social media analytics framework. *The Computer Journal*, 65(4):843–857, 2022.

[17] T. Wang. Integer hash function. `https://gist.github.com/badboy/6267743`, 2007.