Advanced Distributed Load Balancing

Praktikum: Cloud Databases - Project Report

Carl-Leander Henneking
Technical University of Munich (TUM)
Garching bei München, Germany
c.henneking@tum.de

Levin Tschürtz
Technical University of Munich (TUM)
Garching bei München, Germany
ge95wip@mytum.de

Siradanai Treitel
Technical University of Munich (TUM)
Garching bei München, Germany
siradanai.treitel@tum.de

ABSTRACT

In the continuously advancing domain of distributed systems, there is a pressing need to enhance both data management and computational efficiency. This report presents the progress made in the distributed key-value storage systems across Milestones 4 and 5. While the architecture was set up in Milestone 4, Milestone 5 was directed at refining and improving system performance. The focus was on the dynamic allocation and re-allocation of key-value pairs among key-value servers in a ring-like structure, aiming for a more balanced data distribution and equal computational workload among servers. Using benchmarks that aim to mimic real-world scenarios, the system's performance was evaluated. Through various tests and sensitivity analyses, the optimal configurations for diverse data distributions were determined. The results highlight the effectiveness of the newly introduced Advanced Distributed Load Balancing mechanism, showcasing its capability to adapt to changing workloads and resource demands. This report offers a comprehensive understanding of the design choices and strategies in distributed key-value storage systems, setting a foundation for future developments in this area.

1 INTRODUCTION

Distributed systems continue to evolve, addressing the ever-growing needs of data management and computational efficiency. One key component of this evolution is the advancement and refinement of distributed key-value storage systems. During Milestone 4, we set up the fundamental architecture for our system. However, like many complex systems, it had areas that needed improvement. With Milestone 5, our aim was to tackle these challenges head-on and focused on improving our system. We looked at and improved on inefficiencies in the allocation and re-allocation of key-value pairs among key-value-servers (KV-Servers) in a ring-like structure. This meant better data distribution and more balanced computational requirements.

The report is structured as follows. Section 2 details the architectural design choices made throughout our development process. In Section 3, we clarify the strategies and methodologies employed in Milestone 5 to address the earlier challenges. Our findings, drawn from an extensive performance evaluation, are discussed in Section 4. Finally, Section 5 concludes our discussions and offers insights into potential directions for future advancements.

2 ARCHITECTURE

The whole system, starting in Milestone 1, was implemented in Java.

Throughout Milestones 1 to 4, we incrementally developed a persistent, distributed, and replicated Key-Value Storage Service. It includes a KV-Client that provides a command-line-based user interface to put, get, and delete key-value pairs on the KV-Servers. These KV-Servers are organized in a ring-like structure, covering mutually exclusive key-ranges. The organization, orchestration, and assignment of key-ranges of the KV-Servers are handled by an External Configuration Service (ECS). Below, we will highlight critical design choices we made while building the system until the end of Milestone 4 that differentiate us from other implementations.

2.1 Design Choices

We implemented a ring list data structure to simplify operations that involve updating and finding responsible KV-Servers within the required ring-like structure. This ring list is implemented and used throughout the KV-Client, KV-Server, and ECS. It is modified solely by the ECS and passed down to the KV-Servers as so-called metadata, which then triggers an update of the ring list of the KV-Server(s). For operations such as finding the responsible server for a specific key in the KV-Client, or determining successors and predecessors when transferring data, it can be easily queried in linear time complexity $O(n)$.

Additionally, we implemented command-line flag parsing among all three services that parse all passed flags, verify that the user provides the required flags, set default values, and provide adequate error and help messages, given the user input. The values are passed on to the respective constructors during initialization if all input is valid.

As we were required to use direct standard library calls without any additional libraries, we built wrappers around these functionalities that allow us to read byte arrays from and send data strings to input- and output streams.

2.1.1 KV-Client. The relatively simple logic of the KV-Client led us to refrain from adding more features beyond those we explained above and outlined in the specification sheets throughout the milestones.

2.1.2 KV-Server. In the KV-Server, we applied the strategy pattern for different caching strategies (FIFO, LFU, LRU), allowing all strategies to provide the same set of operations to the KV-Server instance, regardless of the strategy used. Our cache sits on top of the persistent storage and can put/get/delete key-value pairs from it.

We based the persistent storage on a one-file per KV-Server system. For the primary storage of a KV-Server, we write all key-value

pairs into a single file in the format of `key,value;`—we also dedicate each of the two possible replicas that a KV-Server stores to an individual file. We initially tried storing the data in JSON but found too much overhead. After concluding Milestone 4, we experimented with a multithreaded approach where we split the keys between multiple files but encountered overhead during creating and deleting these files. We ultimately chose to continue with our single-file approach.

The persistent storage performs these operations if the cache cannot fetch/update/delete a key. During a get operation, we iterate the storage file until we find the corresponding key-value pair, returning null if not found. In a put operation, we first check for the key's presence in the storage file; we update it or append it to the file's end if not found. Delete operations remove the key-value pair in the file if present.

The distributed nature of the service led us to handle offloading of keys or similar tasks through a peer-to-peer TCP socket connection between two neighboring servers, rather than relying on the central ECS. We use the ring list mentioned above to quickly identify the correct KV-Server to contact.

2.1.3 ECS. Because the communication between the KV-Client and KV-Server needed to rely on commands, we tailored this system to meet the communication requirements between the ECS and the KV-Servers. We spawn two new threads for each connection to the ECS Server. One handles a PingConnection, and the other manages the actual transfer of information between the KV-Server and the ECS via an ECSCommunication. We separated these connections to prevent interference, as the transfer of critical data has clear priority over ping requests and responses.

3 APPROACH

The goal of our Milestone 5 project was to improve the system's scalability through usage-based re-allocation of key-ranges between neighboring servers while keeping the whole system as distributed as possible. We added:

- Customizable start & end key-ranges for each KV-Server
- Usage metrics measure the load on an individual KV-Server
- KV-Server key-range partitioning into **N** equal buckets
- Offload of buckets (based on an offload threshold of **T**%) to neighboring nodes, as soon as load-spike is detected

We named this approach Advanced Distributed Load Balancing and aimed to reach an eventual key-range distribution among all KV-Servers that matches the key distribution of the underlying data and divides computational requirements among all available servers more equally. The system works particularly well when the constant total amount of KV-Servers is not optimally distributed between the entire key space.

3.1 Added Functionalities

During startup, the user can define a fixed end-range of a KV-Server. This tweak allows the user to achieve, e.g., an initial equal distribution of the whole key-space between all KV-Server, or, e.g., an allocation of key-range sized in relation to KV-Server hardware. Now, the user is not forced to use an initial distribution of key ranges subject to the result of the IP:Port string hash, as was done previously.

Each KV-Server initializes a frequency table that splits the full key-range assigned to a server in **N** equally sized buckets. When get, put, and delete operations come in, they are also passed to the frequency table instance that keeps the **N** buckets in sync with what is actually stored on the KV-Server. Every KV-Server also counts the number of get, put, and delete operations it has cumulatively received in 30-second cycles. The offloading mechanism is triggered if a pre-set threshold **O** of said operations is reached. The KV-Server experiencing high-load starts up TCP socket connections to its two neighboring KV-Servers and requests their current usage metrics (i.e., operations received in the last 30s). If both neighbors are already experiencing higher loads, no offloading happens. If, however, we can determine a neighboring KV-Server experiencing less load, that server is selected to receive at least threshold **T**% of key-value pairs from the KV-Server under load.

After the recipient of the key-value pairs is determined, the frequency table calculates the key-range to be offloaded. This calculation determines whether the receiving server is the left- or right-hand neighbor and the offload threshold value of **T**%. In the case of an offload to the right-hand neighbor, the frequency table will add up as many buckets from the right side of its key-range required to reach an offload threshold of **T**%. This is possible, as each bucket in the frequency table keeps track of how many keys are present on the KV-Server, and are allocated to itself based on the subset of the KV-Server key-range that a bucket covers. The offload happens by extracting and deleting the string of key-value pairs from the sender's cache and persistent storage, and sending the data as a command with data to the recipient, which takes on the keys and stores them. When receiving keys, any operations incurred through that are not added to the operations count discussed above to avoid constant back-and-forth re-allocation between neighbors. Additionally, both key-ranges of the sender and recipient are re-split into **N** buckets again. This is done to ensure that in a real-world scenario, high-density key-ranges are split up more granularly than low-density key-ranges, to ensure adequate offloading between servers.

To provide a visualization for the user, we fitted the KV-Server and KV-Client with commands that allow the user to see an ASCII symbol representation of the frequency table and information on the current usage metrics. The frequency table command includes the key-ranges that each bucket serves and the percentage of total keys among all buckets that each bucket stores and displays using a vertical histogram-like format, while the usage metrics command provides the current usage statistic of the connected KV-Server.

Each offload of key-value pairs triggers a metadata update sent to all available KV-Servers, which increases overhead compared to the as-is system after Milestone 4. However, we ran benchmarks to quantify this effect. These results are in section 4.

3.2 Design Choices

3.2.1 KV-Client. The KV-Client received the additional commands to fetch the frequency table from the KV-Server that it is connected to, as well as the usage metrics of that server.

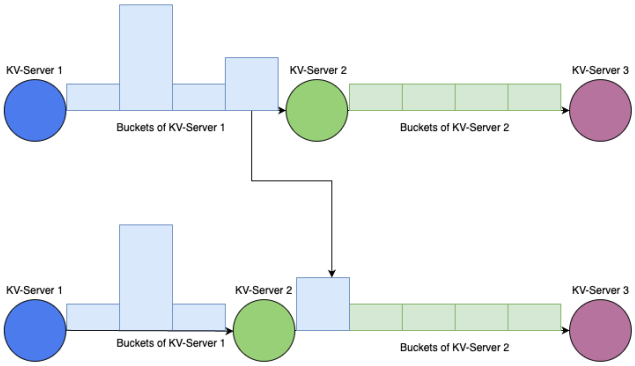

Figure 1: Transfer of bucket between two neighboring KV-Servers.

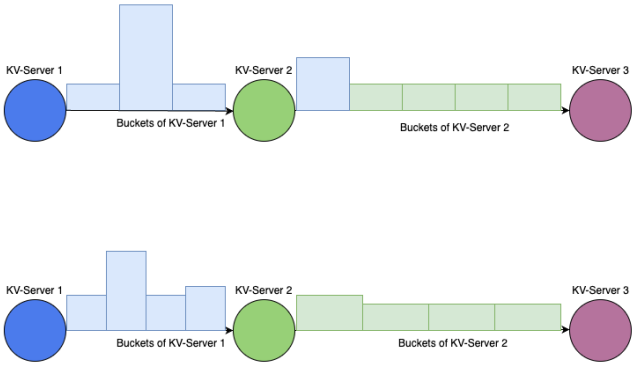

Figure 2: Re-partitioning of key-range into four buckets.

3.2.2 KV-Server. The frequency table stores a list of buckets and the pre-set offload threshold **T**%. Upon initialization, the buckets are generated, and each bucket takes on equal amounts of the KV-Server's full key-range. Each bucket itself stores its keys in a list, which is crucial when re-partitioning the key-range of a KV-Server into the specified **N** buckets again after a transfer of buckets. The re-partitioning is triggered on both KV-Servers as soon as they receive new metadata from the ECS, which has been informed of the update in key-ranges by the sending KV-Server. It works by fetching all the keys from the remaining and added buckets, creating a new set of buckets, and inserting the keys into the suitable buckets. The decision here was clearly made to avoid reading all key-value pairs out of the persistent storage and assigning them to the individual buckets.

The usage metrics class tracks the get, put, and delete operation count within a specified time frame. As part of our system, we set this value to be 30 seconds to ensure we behave adequately when load-spikes occur but are not reallocating key-value pairs when short load bursts of e.g., two seconds occur.

To ensure a high distribution level among the system components, we refrained from orchestrating the transfers via the ECS and instead opted for a peer-to-peer TCP socket connection between the sending and receiving KV-Server for each transfer. After the correct neighboring KV-Server has been determined, a TCP socket connection is initiated for the transfer, which is torn down after the transfer has commenced.

3.2.3 ECS. To allow for the modification of custom start and end key-ranges for each KV-Server, specific calculations inside the ECS were modified to allow for customized value instead of reverting to the hash of the IP:Port string. In line with this, the ECS was fitted with an additional command in which the KV-Server that is offloading keys can announce the new metadata, as the key-ranges of two adjacent KV-Servers have changed. All servers are notified of this change, as they need to adjust their replicas and serve correct metadata to the KV-Clients.

4 PERFORMANCE EVALUATION AND DISCUSSION

The performance of our Milestone 5 database system is best understood when measured and compared directly with its predecessor, Milestone 4. Our Quantitative Evaluation provides concrete metrics highlighting the effectiveness of the modifications and enhancements introduced in the latest milestone. This evaluation is structured around three key benchmarks. First, we analyze the overhead, comparing the new MS5 system with the old MS4 System. The overhead analysis is followed by our sensitivity analysis, where we examine the performance of our system using different configurations and data distributions. This helps us identify the most efficient system parameters for various scenarios. Finally, to get a clear picture of the improvements, we compare the operational performances of Milestone 4 and 5. All tests were conducted in a real-world setting, specifically on a MacBook Air M1 with 8 GB of RAM, ensuring the results reflect practical use cases.

4.1 Overhead Analysis

In refining our database system, understanding the nuances of performance changes between versions is paramount. While new implementations might bring additional functionalities or solve existing issues, they can also introduce overheads affecting the system's responsiveness. This analysis was crucial in determining the trade-off between its enhanced scalability and the decreased processing speed that comes with the new functionalities. With this in mind, we developed a comparative analysis of our recent system versions: We conducted an overhead analysis to evaluate the performance metrics of our prior implementation against our new implementation, ensuring both adhere to the conditions stipulated by Milestone 4. Specifically, these conditions preclude the new implementation that no load-averse offloading of keys takes place. For this evaluation, a single KV-Server was assigned a singular Client. The offload functionality of our system was then turned off to ensure we were only measuring the load of putting the keys into the respective bucket and calculating the correct frequency tables. For this analysis, the client put 10.000 randomly generated keys into the KV-Server. The time taken was measured from start to finish and is illustrated in Figure 3. The new system from MS5, as expected, is around 2 seconds slower than MS4.

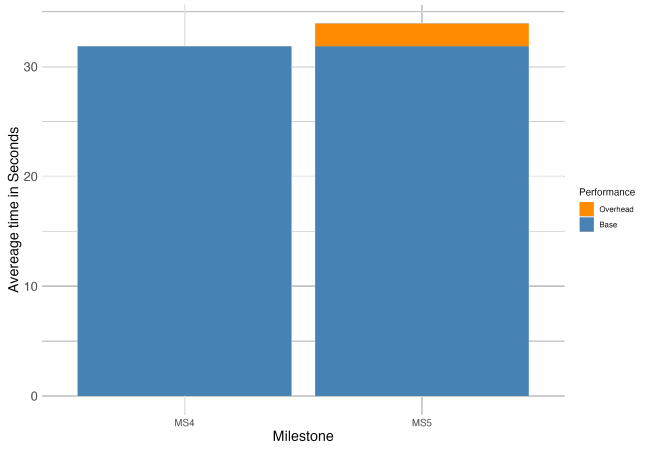

Figure 3: Performance comparison between Milestone 4 and Milestone 5.

4.2 Sensitivity Analysis

One critical aspect is the balance between the number of buckets and the offload threshold, as they dictate how data is managed and accessed. Yet, finding the ideal balance is more complicated, as real-world data can exhibit varying distributions, which can significantly affect performance. To address this, we conducted a comprehensive sensitivity analysis, aiming to determine the configurations that provide the best performance across different data distributions: To achieve results that resemble the real world, we generated key-value pairs that resemble three different distributions. An equal distribution of key-value pairs is shown in Figure 5, a normal distribution as shown in Figure 7 and a spiked distribution as shown in Figure 9. We conducted our tests using different configurations. For the number of buckets N, we tested with three distinct values: 3, 6, and 9. Additionally, we adjusted the offload threshold value **T**, which determines the minimum percentage of keys offloaded. We specifically used the thresholds of 10%, 25%, and 45% since offloading over 50% would obviously not make sense. We used four KV-servers and operated three clients in parallel, putting and getting 3330 keys and measuring the total time it took them to perform set operations. We expected an offloading threshold of 25% in combination with the equal distribution to yield the best results. Furthermore, we also believed that not all distributions would perform equally. Table 1 shows the results of the sensitivity analysis

Table 1: Performance metrics for equal distribution.

Threshold / Bucket Count	10%	25%	45%
9	24.56	24.48	25.97
6	26.67	26.57	27.44
3	28.96	28.27	29.12

for equal distribution. The combination of 9 buckets and 25% offload threshold performs the best under an equally distributed dataset of Key-Value pair with a total time of 24.48 seconds, while 3 buckets with 45% took 29.12 seconds. As seen in Table 2, 9 buckets with a threshold of 25% took the shortest amount of time with 25.62

Table 2: Performance metrics for normal distribution.

Threshold / Bucket Count	10%	25%	45%
9	25.83	25.62	26.76
6	26.96	26.66	28.25
3	29.31	29.58	29.97

seconds. 3 Buckets with **T** = 45%, on the other hand, took 29.97 and therefore performed worst. The spike distribution, as seen in Table 2, took the longest average time out of all the 3 distributions. Here,

Table 3: Performance metrics for spiked distribution.

Threshold / Bucket Count	10%	25%	45%
9	25.96	25.89	27.71
6	28.06	28.05	29.35
3	30.71	30.48	30.62

the trend of the previous 2 distributions continues, and the combination of 9 buckets with **T** = 25% performs best again. However, we could finally see that not all the 3 distributions behave similarly. Here, 3 buckets with **T** = 10% take the longest time at 30.71 seconds.

Overall, our tests suggest that the optimal configuration for most use cases is a total of 9 Buckets with an offload threshold of 25%. Nevertheless, if there is a significant bias in the used data, this configuration might not be optimal, which can be adjusted as needed.

4.3 Performance Comparison

In this final evaluation, we now compare the performance of Milestone 4 vs. the Milestone 5 implementation while now offloading the keys. The goal was to see the performance increase or decrease of Milestone 5. We compared the two implementations regarding get, put, delete, and mixed (a 50-50 blend of get and put) operations. We did this by using the optimal combination of 9 buckets per KV-Server and an offload threshold of 25% determined in our sensitivity analysis. For both Milestones 4 and 5, we used 4 KV-Servers and 3 Clients to concurrently perform the operations. We used the same randomly generated dataset of 10,000 Key-Value pairs for both implementations. Figure 4 shows the results of this comparison.

As seen in Figure 4, there is a performance increase across the board for all operations. The highest increase in performance was measured for PUT and GET operations, with an increase of over 70 operations per second.

4.4 Discussion

As we can see from our overhead analysis, our new system introduces a minor increase in processing time compared to Milestone 4. We attribute this slight delay to the fact that the MS4 system performs no calculations when inserting keys into the respective storage. While in contrast, our new system performs multiple comparisons to insert every key into the right bucket, keeps track of its usage metrics, and needs to recalculate the key range for the individual buckets, which introduces additional computations. Overall, however, the slight performance decrease is acceptable and is outweighed by the system's scalability, load distribution capabilities,

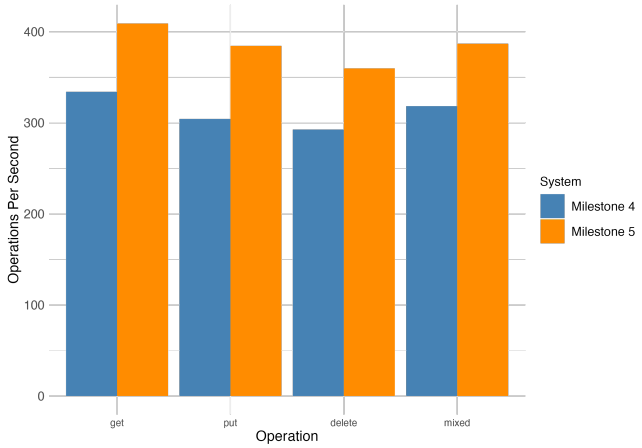

Figure 4: Operational Throughput of Milestones 4 and 5 in OPS/s.

and overall performance, as seen in the performance comparison of MS4 and MS5, and therefore can be seen as a reasonable trade-off.

The optimal configuration of 9 buckets and an offload threshold of 25% we could identify for our system is likely due to a balance between load distribution and communication overhead. As seen in the overhead analysis, a high number of buckets introduces more overhead. In contrast, a lower number of buckets needs to ensure more granularity of data partitioning. 9 buckets were deemed to be an optimal middle way between enough granularity for balanced load distribution while ensuring low computational overhead. An offload threshold ensures optimal key and resource utilization across all KV-Servers. It must ensure we are offloading only a few keys and therefore causing too much overhead and latency when offloading keys, which causes communication delays and system overloading. While also making sure we are not effectively just putting the burden on the other server and overwhelming it with a new workload. We also must guarantee system stability, which also can be affected by sudden and frequent offloading. A threshold of 25% was measured to be optimal. It ensures the correct distribution of keys and system stability; hence it avoids excessive communication and computational overhead.

The increase in performance we determined in our final comparison is assigned to multiple factors. Firstly, the dynamic offloading of keys ensures that heavily loaded servers offload their keys by distributing them to other servers. Therefore, the load is evenly distributed and spread throughout different servers, which leads to an improvement in performance and faster server response times since the old key-range is now distributed among multiple servers. Secondly, resources are more efficiently utilized since our system offloads keys to servers with fewer loads and adaptively scales as needed. Furthermore, our system detects load spikes since it continuously monitors load metrics and can handle them in real-time. All these factors lead to a scalable system that can handle larger workloads more effectively.

5 CONCLUSION

This report detailed the improvements made in Milestone 5 to our distributed key-value storage system. Our primary focus was enhancing data distribution across KV-Servers in a ring-like structure to achieve better efficiency. The introduction of the Advanced Distributed Load Balancing mechanism allowed for a balanced computational load and more dynamic data management.

Through a detailed sensitivity analysis, we demonstrated the system's performance under various data distributions, emphasizing the need for selecting appropriate configurations based on specific use cases to finally determine an optimal value for the threshold parameter **T** and number of buckets **N** being 25% and 9 respectively.

In summary, Milestone 5 marked a notable progression in the development of our distributed system. The findings and results from this report provide a foundation for further research and refinements in distributed key-value storage systems.

A DATA DISTRIBUTIONS AND HEATMAPS FOR SENSITIVITY ANALYSIS

A.1 Equal Distribution

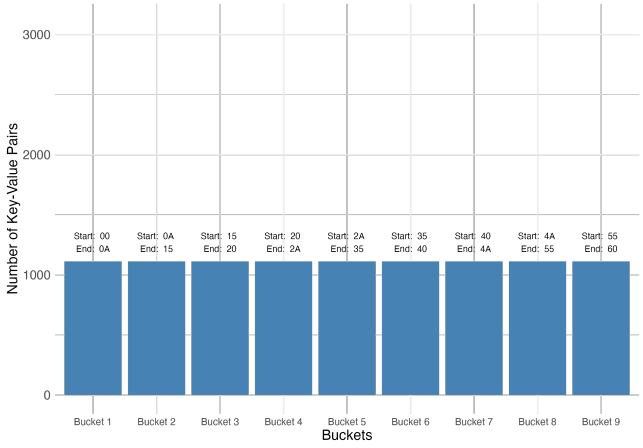

Figure 5: Equal distribution of Key-Value pairs in nine buckets.

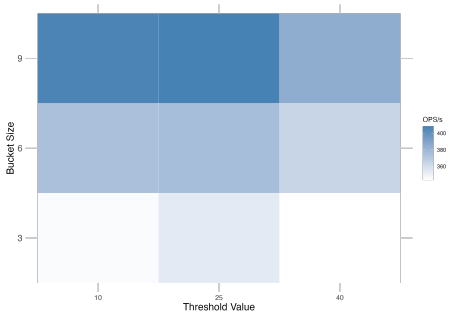

Figure 6: Heatmap for equal distribution of Key-Value pairs based on threshold and bucket size.

A.2 Normal Distribution

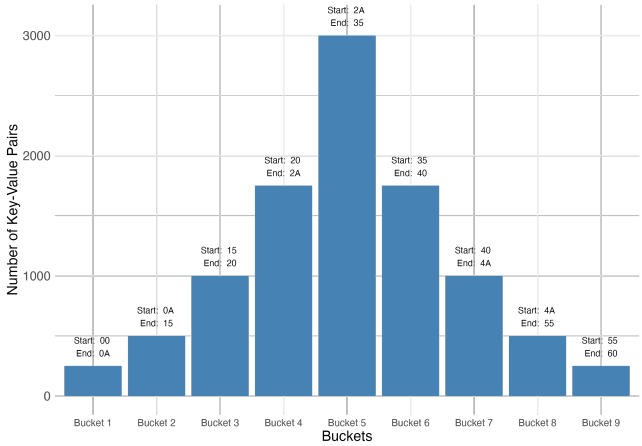

Figure 7: Normal distribution of Key-Value pairs in nine buckets.

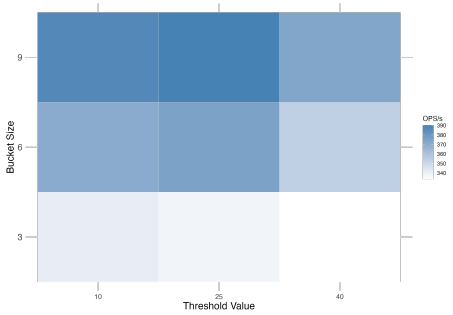

Figure 8: Heatmap for normal distribution of Key-Value pairs based on threshold and bucket size.

A.3 Spiked Distribution

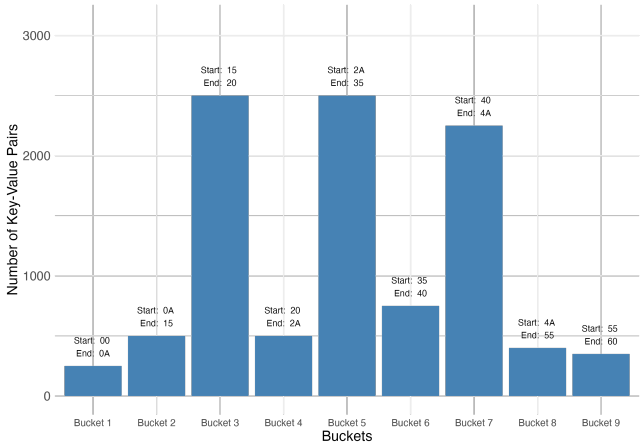

Figure 9: Spiked distribution of Key-Value pairs in nine buckets.

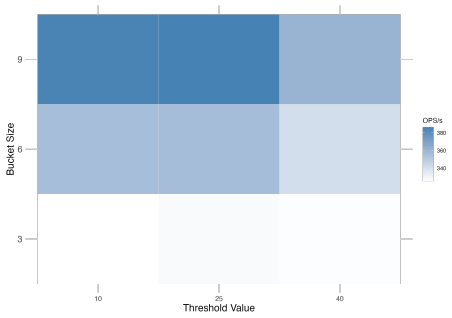

Figure 10: Heatmap for spiked distribution of Key-Value pairs based on threshold and bucket size.