Lightweight Asynchronous Repartitioning for Local State Partitioned Systems

Douglas Pereira Luiz[1], **Odorico Machado Mendizabal**[1]

[1]Departamento de Informática e Estatística
Universidade Federal de Santa Catarina (UFSC) – Florianópolis – SC – Brazil

douglas.pereira@posgrad.ufsc.br, odorico.mendizabal@ufsc.br

Abstract. *Partitioning strategies combined with rebalancing algorithms can be used to balance the load in high-throughput systems. Keeping the load balanced constantly is desirable, but the cost of repartitioning can be high. This work presents a rebalancing strategy based on balanced graph partitioning algorithms with low impact during rebalancing operations. The new technique allows for frequent partitioning updates by decoupling the repartitioning process from the rest of the system, thereby avoiding disruptions within the system due to the calculation of new partitioning schemas. In experimental evaluation, the proposed strategy, implemented in an in-memory key-value store prototype, eliminated scheduler pauses and increased throughput by 19% in workloads predominantly composed of scanning requests.*

1. Introduction

State partitioning is a common approach to increase throughput. However, predicting an efficient partitioning schema is challenging, and strategies that establish the level of parallelism at startup may be inadequate for handling dynamic workloads [Alchieri et al. 2017]. To fully exploit the potential of available parallelism, partitioning can be reconfigured during execution. This dynamic reconfiguration helps balance the workload among threads and reduces the impact of synchronization overhead caused by conflicting operations [Goulart et al. 2023].

Graph partitioning algorithms can be employed to define distributed partitioning schemes based on observed workload patterns [Curino et al. 2010, Quamar et al. 2013, Hoang Le et al. 2019]. This approach enhances the quality of data placement decisions in systems with partitioned state, providing an alternative to workload-unaware placement policies such as hash, range or other static partitioning methods.

Systems with partitioned local state can also benefit from graph-based partitioning strategies. By monitoring recent workload, these systems can reconfigure the partition schema during execution, in a process known as repartitioning [Goulart et al. 2023]. However, pausing the execution or restricting repartitioning to idle periods may be inadequate strategies for high-throughput systems where minimizing downtime is crucial. In light of this, we aim to reduce the cost of repartitioning, enabling applications to benefit from frequent or even continuous repartitioning.

Our new technique avoids system downtime by introducing an additional execution thread dedicated to graph partitioning. Due to the minimal disruption caused by repartitioning with our strategy, it is possible to present the scheduler with new partitioning schemes consecutively.

2. Related work

Graph partitioning algorithms are used in [Quamar et al. 2013] to address the data placement problem in a distributed environment, aiming to achieve load balancing and minimize the number of distributed transactions. The system models the workload as a hypergraph and employs a compression technique to reduce partitioning costs. This approach includes repartitioning the system to adapt to the current load. However, the repartitioning process is conducted incrementally due to the high cost associated with transferring data between nodes.

A dynamic load balancing approach was presented in [Didona and Zwaenepoel 2019], where the authors introduce a partitioned-state key-value store that makes scheduling decisions based on the size of the requests. The proposed strategy divides the execution threads into two disjoint sets, assigning different processor cores to handle requests for small and large items. This prevents fast requests from being queued behind slower ones. The threshold differentiating large and small requests is automatically reconfigured during execution. Experimental evaluations show that this technique promotes faster processing of requests with lower computational weight, resulting in significantly lower latencies in the 99^{th} percentile and a throughput 7.4 times higher than similar key-value store solutions.

A different approach to handling repartitioning overhead is presented in [Goulart et al. 2023]. Recognizing that the costs of repartitioning are inevitable, the authors mitigate the effects of scheduling downtime associated with rebalancing by utilizing system downtime during system recovery checkpoints. During these pre-programmed idle periods, the system's partitioning scheme is redefined, rendering the scheduling downtime imperceptible. Although the cost of repartitioning itself is not eliminated, it is effectively masked by the system's inactivity during snapshots. This solution, however, is limited in its applicability, as it depends on the presence of known or expected idle periods.

3. State partitioning execution model

In this work, we assume a replicated system where each replica implements a scheduler-based multi-threaded execution model. In each replica, the scheduler dispatches incoming requests to worker threads according to a partition map. This scheduling strategy based on partitioning is for scaling up performance and is similar to the approaches presented in [Goulart et al. 2023, Mendizabal et al. 2017, Li et al. 2016, Li et al. 2018]. The application state S is a set partitioned into n disjoint sets, each of which is called a *partition*. The set of partitions is $\{p_1, p_2, ..., p_n\}$ and the union of the elements of each p_i is equal to S. The system executes n worker threads, where thread t_i is responsible for the execution of requests involving the partition p_i. Requests are dispatched to the appropriate queues respecting their delivery order.

Replicas implement a key-value store service that handles read, write, and scan requests, given by commands $read(k)$, $write(k, v)$, and $scan(k_i, k_j)$. The $read$ command returns the value v associated to k, the $write$ command updates the variable given by the key k with a value v, and the $scan$ command returns a list of values $[v_i, v_j]$ associated to keys in the range interval $[k_i, k_j]$. The choice of the key-value store service stems from its simple and lightweight execution, as well as its wide adoption in today's data-intensive applications. Our motivation is to address a relevant class of application for

state management while keeping application costs low so that our analysis emphasizes the costs inherent to repartitioning.

The replicas include the following components:

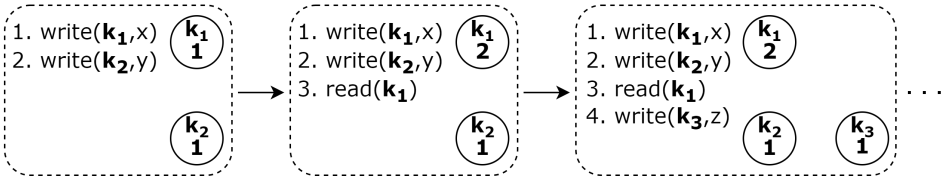
- *Worker threads:* these threads ($t_i, .., t_n$) receive requests from a queue ($q_i, .., q_n$, respectively) and execute them. Requests that depend on keys within a single partition are executed immediately, while requests involving keys in multiple partitions are coordinated with the *worker threads* serving all the partitions involved;
- *Workload tracker*: is a thread responsible for recording the access pattern to keys in the form of a graph. This component has a queue from which it retrieves the scheduled requests to update the workload graph at runtime;
- *Scheduler:* it receives incoming requests and dispatch them to both the *worker threads* and *workload tracker*. The scheduler follows a *partition map* to determine to which thread queues each request will be delivered. Single partition requests are just added to the proper queue while cross-partition multi-variable requests are added to multiple thread queues, as they require synchronization between threads to implement atomicity and, consequently, maintain consistency.

The requests received by the scheduler may be for reading, writing, or scanning, and each request necessarily includes a key. If it is the first time a request with a particular key is submitted to the system, the scheduler assigns the key to a partition in a round-robin manner. The synchronization required for scanning requests is ensured with a special synchronization command *sync*, which holds a barrier for the number of partitions involved in the request. When a worker removes a *sync* command from the queue, it waits for the other involved workers to reach the barrier, thus ensuring atomicity of the scanning execution.

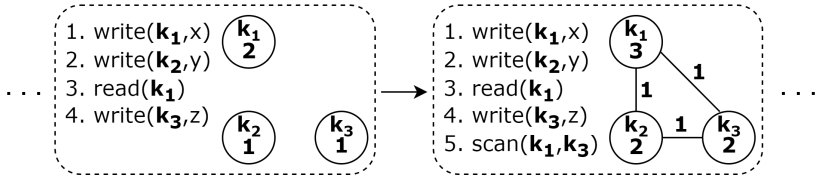
Partition reconfiguration, or repartitioning, modifies the *partition map* based on information from the *workload graph* generated by the *workload tracker*. Figures 1(a) and (b) illustrate how different types of requests affect the weights in the graph, showing the received requests' history alongside the corresponding graph. For each key read or written in a request, the weight of the corresponding vertex is incremented by one. For each pair of keys within the read range of a scan request, the weight of the edge between the corresponding vertex pairs is incremented by one. If a specific vertex or edge does not exist, it is created with an initial weight of one.

Repartitioning is performed by the scheduler every Δp scheduled operations, where Δp is defined at system initialization. At every repartitioning interval, the scheduler synchronizes with the workload tracker, performs the repartitioning, and updates the *partition map*. Before resuming scheduling, the scheduler ensures the safe execution of requests by adding a *sync* command to all *worker threads* queues, allowing them to complete all the operations scheduled according to the previous partition scheme before starting executing the operations dispatched according to the new *partition map*.

There is a tradeoff in choosing the repartition periodicity. While frequent repartitions may balance the load distribution among worker threads and reduce the expensive multi-partition operations, the repartition itself forces scheduling interruptions. Therefore, Δp must be chosen carefully, as small values lead to frequent stops, while large values might provide insufficient rebalancing.



(a) Read or write operation.



(b) Scan operation.

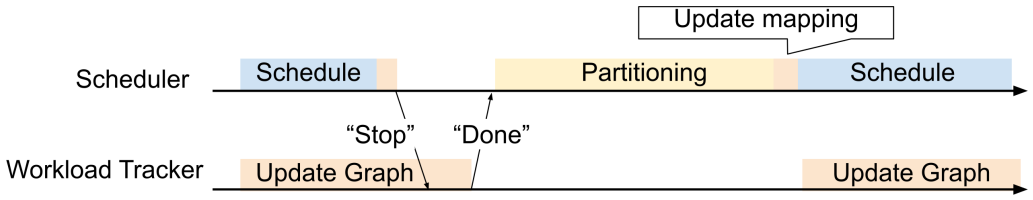
Figure 1. Example of *workload graph* update.

4. Asynchronous repartitioning scheduler

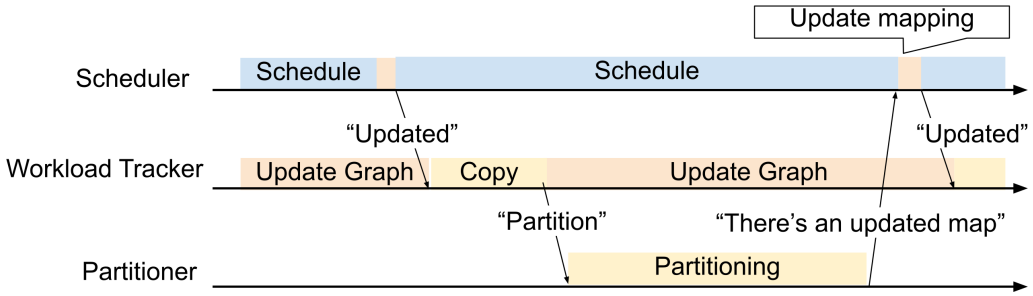
In this section, we present a new repartitioning approach designed to minimize scheduling interruptions caused by repartitioning. In the asynchronous repartitioning approach, referred to as *Async*, a dedicated partitioning thread obtains a copy of the workload graph generated by the workload tracker and creates a new partition map using a partitioning algorithm. During this process, the scheduler continues to dispatch requests, and the worker threads continue to process them without interruption. Repartitioning occurs asynchronously with respect to the scheduler.

At the end of the partitioning process, the scheduler is notified of the new partition map and simply performs the mapping switch, which is a quick operation. After the switch, the process repeats in a new iteration. This way, the *Async* strategy does not require setting a fixed Δp, as it can run continuously with minimal overhead. Figures 2(a) and (b) illustrate the service execution and partitioning processes as presented in Section 3 and in this work, respectively. In the traditional repartitioning (see Figure 2(a)) the costs associated with updating the workload graph and executing the partitioning algorithm directly impact scheduling interruptions. In contrast, with the asynchronous repartitioning approach (see Figure 2(b)), scheduling interruption is practically unnoticed. As observed, when the workload tracker detects that no partitioning process is in progress, it creates a copy of the graph, notifies a dedicated thread, called the *partitioner*, and resumes updating the workload graph based on new requests. Upon notification, the partitioner creates a new partition map using the graph copy and notifies the scheduler. When the scheduler detects the existence of the new map, it replaces the current map with the updated one and signals that the rebalancing has been completed. At this point, the workload tracker initiates a new rebalancing process.

The behavior of the components shown in Figure 2 can be described in Algorithms 1, 2 and 3. For simplicity, synchronizations in the $INSERT$ and $WAITANDREMOVE$ calls on the $trackerQ$ queue, as well as in the reads and writes of the $isAvailable$ and $isPartitioning$ variables, are omitted. The procedure in Algorithm 1 is executed by the scheduler thread for each incoming request. The $DISPATCH$ method forwards the request to a worker thread based on the mapping defined in



(a) Common repartitioning approach.



(b) Asyncronous continuous repartitioning approach.

Figure 2. Comparison between repartitioning approaches.

$PartitionMap$, managing synchronizations for scan requests. The $SYNC(Workers)$ call inserts a synchronization command for all threads into each worker queue.

Algorithm 1 Scheduler

```
1:  procedure SCHEDULE(request)
2:      DISPATCH(request)              ▷ Forwards request to the respective workers
3:      trackerQ.add(request)          ▷ Forwards request to workload tracker
4:      if isAvailable = true then     ▷ There's a new map available
5:          PartitionMap ← UpdatedPartitionMap      ▷ Updates map
6:          SYNC(Workers)              ▷ Synchronizes workers
7:          isAvailable ← false
8:          isPartitioning ← false     ▷ Signals end o repartitioning iteration
9:      end if
10: end procedure
```

In Algorithms 2 and 3, the semaphore $partitionSem$ is used to keep the partitioner in a wait state until it is signaled by the pattern tracker. The $PARTITION(GraphCopy)$ call generates a partition map based on the graph copy.

4.1. Optimizations

The *Async* repartitioning uses a dedicated thread for partitioning, allowing the scheduler to continue dispatching requests while the new partition map is created. Although this strategy might seem only advantageous at first glance, it can introduce an unintended consequence. Since scheduling is a rapid process, by the end of a partitioning iteration, many requests using the outdated partition scheme might still be enqueued in the worker threads. This phenomenon can delay the benefits of the new map, as they will only become apparent after the last request forwarded before the map switch is processed by a worker.

Algorithm 2 WorkloadTracker

on thread run:
1: **while** $true$ **do**
2: $request \leftarrow$ WAITANDREMOVE$(trackerQ)$ ▷ Waits and removes request
3: UPDATEGRAPH$(Graph, request)$ ▷ Updates graph
4: **if** $\neg isPartitioning$ **then** ▷ When there's not ongoing partitioning
5: $isPartitioning \leftarrow true$
6: $GraphCopy \leftarrow Graph$ ▷ Copies the workload graph
7: SIGNAL$(partitionSem)$ ▷ Starts a repartitioning iteration
8: **end if**
9: **end while**

Algorithm 3 Partitioner

on thread run:
1: **while** $true$ **do**
2: WAIT$(partitionSem)$ ▷ Wait for signal
3: $UpdatedPartitionMap \leftarrow$ PARTITION$(GraphCopy)$ ▷ Creates new mapping
4: $isAvailable \leftarrow true$ ▷ Signals availability of the new map
5: **end while**

To address this problem, we modified our approach to include bounding the worker threads' queues to a predefined size. With this modification, the scheduler can delay dispatching a request if the queue it is trying to insert into already holds Q requests. The *bounded queue* enhancement ensures that the maximum number of requests processed before the new partition scheme takes effect is known, potentially leading to faster performance improvements.

In the workload tracking method described in Section 3, the graph's vertices and edges are created or their weights incremented for each request received throughout the system's entire lifetime. As a result of this continuous accumulation, the overall size or total weight of the graph increases with each received request, leading to high partitioning times as more requests are processed.

To improve partitioning time, we propose constructing the workload graph using only the most recent W requests. This approach, based on *sliding window*, limits the maximum number of vertices and edges in the workload graph by disregarding older requests. The *sliding window* method may be particularly effective for workloads where the most recent W requests accurately represent the current workload.

5. Experimental evaluation

To experimentally evaluate the *Async* technique, we considered the key-value store prototype presented in [Goulart et al. 2023], which we refer to as *Base*, and it represents the repartitioning strategy described in Section 3. We extended this version by implementing the techniques described in Section 4, resulting in the *Async* version.[1] The *Base* version requires the configuration of the interval Δp between rebalancings, in number of opera-

[1] `https://github.com/douglaspereira04/kvpaxos`

tions. The *Async* version can be configured with window size W and queue size Q. When omitted, these sizes are considered unlimited.

The prototype allows configuration of the number of partitions/worker threads and can operate without performing partitioning, using a round-robin scheduling policy that assigns new keys to worker threads in a circular manner. To solve the minimum cut problem in graphs, METIS [Karypis and Kumar 1998] was used for partition reconfiguration. The experiments were configured with: 10^6 key-value pairs initially in the system; 8 worker threads; keys of 4 bytes; values of 4 kbytes.

The set of requests executed in the experiment is based on workloads A, D, and E from the Yahoo! Cloud Serving Benchmark (YCSB) [Cooper et al. 2010]. Workload A was configured with 5×10^7 requests, consisting of 50% updates and 50% reads. Workload D was configured with 5×10^7 requests, comprising 5% writes of new key-value pairs and 95% reads. Workload E was configured with 5×10^6 requests, with 95% scans, having a uniform variation in the number of keys accessed, ranging from 2 to 8, and 5% updates to existing keys in the system. The tests were conducted on a computer with two Intel Xeon E5-2630 processors, each running at 2.4 GHz, with 8 cores and 20 MB cache, and 64 GB of DDR4 RAM. The operating system used was Ubuntu v22.04 64-bit. The test programs were developed in C++17 and compiled with gcc v9.4.0.

Experiments were conducted with workloads A, D, and E, and graphs were produced showing execution time in seconds on the x-axis and throughput in thousand operations per second on the y-axis. Experiments were performed with the *Base*, *Async*, *round-robin* (RR), and single worker thread (SW) versions. When configured, the values for Δp, W, and Q are defined in labels.

5.1. Sliding window results

In the experiments with workload A, Δp was set to 100×10^3, 1×10^6, and 10×10^6 for the *Base* version. For this workload, the lowest makespan observed with *Base* was achieved using $\Delta p = 10 \times 10^6$, which resulted in an 11% decrease compared to the round-robin version (RR). The SW version, with a single worker thread, completed the execution in 817 seconds, which is 5.7 times longer than the RR version. These results highlight the advantages of employing repartitioning and multiple workers under this workload.

Figure 3 presents the results for the *Async* versions with different *sliding window* sizes, *Base*, and RR The RR version, unlike the other versions, experienced a stair-stepped decline in throughput after 100 seconds due to the load imbalance among worker threads. This imbalance can be quantified with the help of the coefficient of variation (CV) of the sizes of the worker queues measured every second throughout the experiment. While the median CV of RR was 23.44%, *Base* displayed a median CV of 1.41%, and *Async* measurements were lower than 0.7%. The *Base* version, which achieved the lowest makespan, experienced throughput drops of up to 75% during partitioning due to the scheduler halting during reconfigurations.

The default *Async* achieved a makespan that was 9.9% lower than the RR version. The configuration of sliding windows led to an increase in execution time of up to 9.2%. Compared to the *Base* version, which completed in 125s, the *Async* version had a higher makespan of 127s. However, the *Async* version did not experience significant throughput drops, as the strategy kept the scheduler free to forward requests.

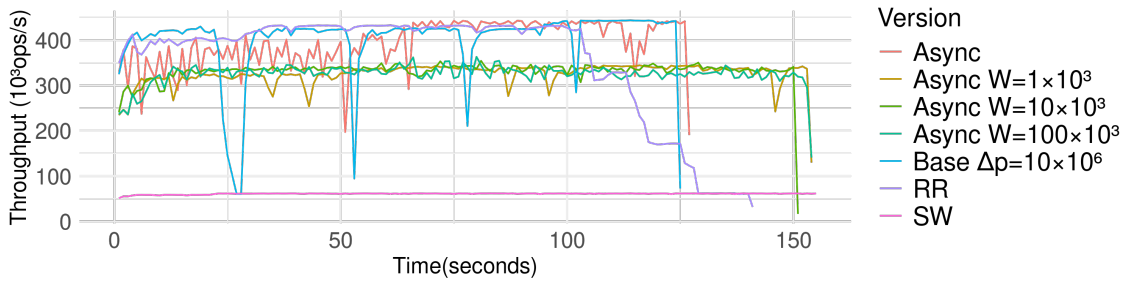


Figure 3. Execution under Workload A with sliding windows. Higher throughput and lower execution time indicate better performance.

By recording the number of repartitions performed during the experiments, we observed that *Async* repartitioned 60 times without using sliding windows. In contrast, the highest-performing *Base* version performed 4 repartitions, a number dictated by the workload's request count. For the versions with $W = 1 \times 10^3$, $W = 10 \times 10^3$, and $W = 100 \times 10^3$, the numbers of partitioning iterations were 825, $6,836$, and $60,065$, respectively. Since partitioning time is proportional to the size of the partitioned graph, the smaller the window, the more repartitions can be performed within the same time interval. As each repartitioning causes the synchronization of all workers, the high frequency of repartitions was responsible for the decrease in the maximum throughput observed.

Figure 4 shows the results of the sliding window approach under Workload D. This workload continuously inserts new values and is dominated by read requests for recently inserted keys, simulating scenarios like trending topics in social networks. In such cases, versions with repartitioning offer little or no advantage over the RR version, as the rebalancing relies on keys from previous requests that quickly become outdated and rarely accessed.

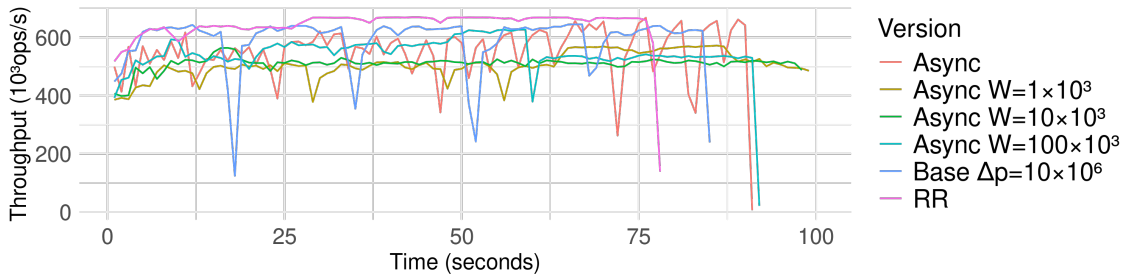


Figure 4. Execution under Workload D with sliding windows. Higher throughput and lower execution time indicate better performance.

Figure 5 presents the results under Workload E, a synchronization-intensive workload. In this workload, the RR approach with 8 threads performed worse than SW with single worker. The SW, which does not require synchronization commands, completed in 270s, whereas RR, which must process synchronization commands for every scan across different partitions, experienced decreased throughput, leading to a completion time of 969s. To improve visibility, we limited the x-axis (time value) to 300 seconds.

The *Async* version without optimizations takes as much time as RR to process the workload because the scheduling of the whole workload is completed even before the

partitioner finishes the graph cut of the first partitioning iteration. Thus, no repartitioning takes effect. Although the throughput of *Async* $W = 1 \times 10^3$ drops at the 27^{th} second of execution, this version achieved the shortest makespan, completing the execution in 117s. This represents a 22% reduction in execution time compared to the *Base* version and a 56.6% reduction compared to the SW version.

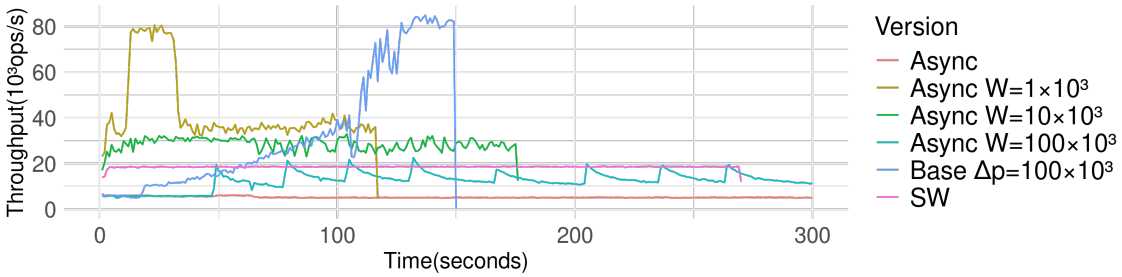


Figure 5. Execution under Workload E with sliding windows. Higher throughput and lower execution time indicate better performance.

The behavior displayed by Async $W = 1 \times 10^3$ under Workload E is caused by a combination of factors. The first repartitioning, for which the map switch occurs 0.0017s after the experiment begins, causes the sudden increase in throughput seen about the 13^{th} second. The map switch of the second repartitioning iteration occurs 0.1082s after the first, allowing the processing of all requests forwarded in this interval to be free from the interference of forced synchronizations of all threads due to repartitioning. The subsequent intervals between map switches range from 0.0052s to 0.0315s, with a median of 0.0068s and a mean of 0.0071s. These short intervals provide little time for the worker threads to process requests without the interference of map switches which are accompanied by synchronization commands in every worker thread queue, thereby limiting parallelism. As a result, throughput decreases and remains limited due to the excessive number of forced synchronizations of all workers. This suggests that, even if there is no cost to produce a new map, very frequent repartitioning may be detrimental to the system's throughput.

5.2. Bounded queues results

The bounded queues optimization was tested with values of $1,000$, $10,000$ and $100,000$ for Q. As seen in Figure 6, we could not improve throughput or makespan with this modification under Workload A. Since this workload consists entirely of update and read requests, the constructed graph has no edges, and the number of vertices remains equal to the number of keys in the initial population, which is $1,000,000$ throughout the execution. This makes the time taken to cut the graph very short.

This way, the effects of the repartitioning benefit postponement, as described in Section 4.1, are not as intense as those under Workload E. Thus, under Workload A, the most significant effect of this technique is the negative impact caused by introducing additional synchronizations: while the default *Async* version completes 60 partitioning iterations, 141 to 146 complete repartitionings were registered with bounded queues.

As shown in Figure 7, using bounded queues provided no benefits under Workload D. Versions with repartitioning, as discussed earlier, struggle under this workload.

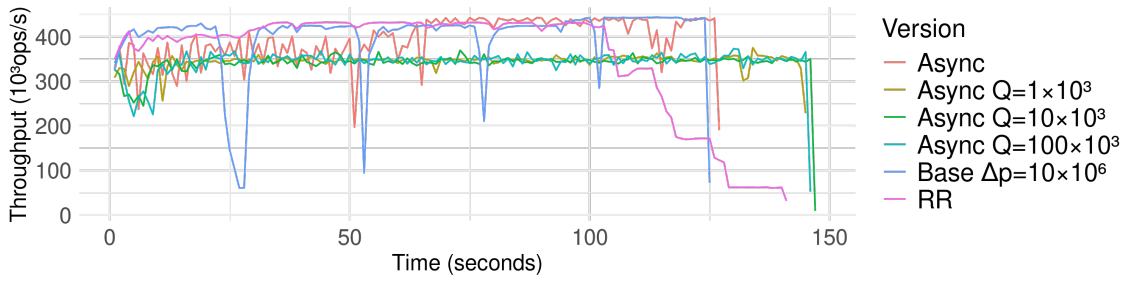


Figure 6. Execution under Workload A with bounded queues. Higher throughput and lower execution time indicate better performance.

Additionally, the increased number of repartitionings compared to the default *Async* version causes the bounded queues version to underperform relative to the version with unbounded queues. The time taken to process the workload was 3.29% to 14.28% longer than the *Async* version with no queues, and 20.51% to 33.33% longer than that of RR.

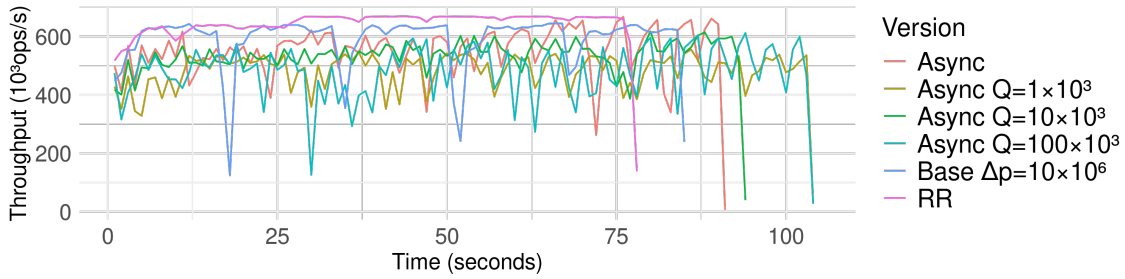


Figure 7. Execution under Workload D with bounded queues. Higher throughput and lower execution time indicate better performance.

The experiments with Workload E showed that the bounded queues modification allows to overcome the problem faced with the default *Async* version. In Figure 8, *Async* versions with bounded queues can be seen outperforming default *Async* and SW versions, processing up to $85,000 ops/s$. Although *Base* achieved the highest throughput and lowest makespan, processing up to $81,000 ops/s$ and completing the entire workload in 150s with $\Delta p = 100 \times 10^3$, the Δp value must be chosen carefully, given the other tested values for *Base's* Δp resulted in poor performance, with makespans of 328s and 1141s.

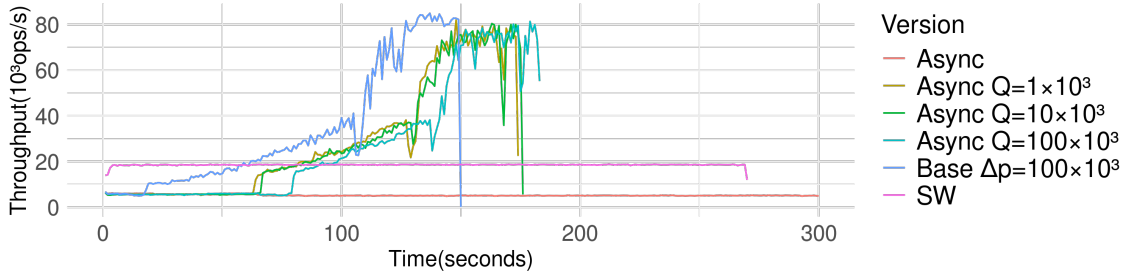


Figure 8. Execution under Workload E with bounded queues. Higher throughput and lower execution time indicate better performance.

The experimental evaluation highlighted the strengths and weaknesses of the *Async* repartitioning strategy. The proposed strategy, designed to prevent scheduler in-

terruptions during repartitioning, has successfully resolved this issue, as seen in Figure 3. Given the possibility of repartitioning without interrupting execution, the asynchronous strategy was employed to assess the benefits of repartitioning as frequently as possible. However, as already discussed, the introduction of an excessive number of synchronizing commands can decrease performance.

Table 1 presents the makespan in seconds and the median throughput measures in thousands of operations per second for each version under each workload. The highest throughput results for each workload are highlighte in green. Although *Async* could not match Base and RR in throughput under Workloads A and D, it showed significantly higher throughput medians under Workload E, particularly for *Async* with $W = 1 \times 10^3$ and $W = 10 \times 10^3$. While SW and Base displayed medians of $18,606 ops/s$ and $24,194 ops/s$, respectively, the medians for *Async* $W = 1 \times 10^3$ and $W = 10 \times 10^3$ were $36,115 ops/s$ and $28,863.5 ops/s$. This represents increases of 49.27% and 19.3% compared to the *Base* version.

Table 1. Summary of experimental results

	Workload A		Workload D		Workload E	
	Makespan	**Throughput (10^3ops/s)**	**Makespan**	**Throughput (10^3ops/s)**	**Makespan**	**Throughput (10^3ops/s)**
Async	127.02	406	91.01	568	976.12	5
Async Q=100×10^3	146.02	349	104.02	497	183.02	19
Async Q=10×10^3	147.02	344	94.01	544	176.02	20
Async Q=1×10^3	145.02	348	104.02	506	174.02	20
Async W=100×10^3	154.02	328	92.01	536	385.05	13
Async W=10×10^3	151.02	336	98.01	513	176.02	29
Async W=1×10^3	154.02	335	99.01	503	117.02	36
Base Δp=100×10^3	461.56	100	1,079.30	31	1,141.30	3
Base Δp=1×10^6	137.02	376	202.74	232	150.02	24
Base Δp=10×10^6	125.02	423	86.02	624	328.04	5
SW	817.11	61	545.07	92	270.04	19
RR	141.02	418	78.01	664	969.12	5

Although the *Async* strategy shows good performance, the results highlight the limitations of repartitioning too frequently. Intervals between repartitionings are beneficial for execution, allowing worker threads to operate freely and take advantage of the parallelism gained from reconfiguring the partitioning scheme.

6. Discussion

We implemented a rebalancing technique based on graph partitioning that takes the workload into account and does not interrupt scheduling. We introduced two modifications to the initially proposed strategy: the use of bounded queues to avoid delaying the adoption of the new partitioning scheme, and sliding windows to optimize partitioning time by limiting the graph to represent only a recent portion of the workload.

The experiments suggest an additional cost associated with the use of the new strategies when services are subjected to workloads primarily consisting of commands on a single variable and frequent changes in access patterns for variables. However, unlike the compared repartitioning strategy, the new approach does not cause undesirable periods of service interruption, allowing the system to remain operational during repartitioning.

When the workload consisted primarily of scan requests, the repartitioning techniques yielded better results. The highest median of throughput measures was achieved with the new strategy, significantly surpassing the medians of the other tested approaches.

Acknowledgements

We wish to acknowledge the contribution of João Trombeta, Henrique Goulart, and Álvaro Junio Pereira Franco, who were involved in earlier versions of the prototype and offered valuable insights into the graph-based repartitioning approach.

References

Alchieri, E., Dotti, F., Mendizabal, O. M., and Pedone, F. (2017). Reconfiguring parallel state machine replication. In *Proceedings of SRDS '17*, pages 104–113.

Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. (2010). Benchmarking cloud serving systems with ycsb. In *Proceedings of SoCC '10*, page 143–154.

Curino, C., Jones, E., Zhang, Y., and Madden, S. (2010). Schism: a workload-driven approach to database replication and partitioning. *Proc. VLDB Endow.*, 3(1–2):48–57.

Didona, D. and Zwaenepoel, W. (2019). Size-aware sharding for improving tail latencies in in-memory key-value stores. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, NSDI'19, page 79–93, USA. USENIX Association.

Goulart, H., Trombeta, J., Franco, A., and Mendizabal, O. (2023). Achieving enhanced performance combining checkpointing and dynamic state partitioning. In *Proceedings of SBAC-PAD '2023*, pages 149–159.

Hoang Le, L., Fynn, E., Eslahi-Kelorazi, M., Soulé, R., and Pedone, F. (2019). Dynastar: Optimized dynamic partitioning for scalable state machine replication. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 1453–1465.

Karypis, G. and Kumar, V. (1998). A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, pages 359–392.

Li, B., Xu, W., Abid, M. Z., Distler, T., and Kapitza, R. (2016). Sarek: Optimistic parallel ordering in byzantine fault tolerance. In *2016 12th European Dependable Computing Conference (EDCC)*, pages 77–88. IEEE.

Li, B., Xu, W., and Kapitza, R. (2018). Dynamic state partitioning in parallelized byzantine fault tolerance. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 158–163. IEEE.

Mendizabal, O. M., De Moura, R. S., Dotti, F. L., and Pedone, F. (2017). Efficient and deterministic scheduling for parallel state machine replication. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 748–757. IEEE.

Quamar, A., Kumar, K. A., and Deshpande, A. (2013). Sword: Scalable workload-aware data placement for transactional workloads. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, page 430–441, New York, NY, USA. Association for Computing Machinery.