Shrinking Logs by Safely Discarding Commands

Luiz Gustavo C. Xavier[1], **Fernando Luís Dotti**[2],
Cristina Meinhardt[1], **Odorico M. Mendizabal**[1]

[1]Departamento de Informática e Estatística
Universidade Federal de Santa Catarina (UFSC)

[2]Escola Politécnica
Pontíficia Universidade Católica do Rio Grande do Sul (PUCRS)

l.gustavo.x@posgrad.ufsc.br, fernando.dotti@pucrs.br,

{cristina.meinhardt, odorico.mendizabal}@ufsc.br

Abstract. *Logs are crucial to the development of dependable distributed applications. By logging entries on a sequential global log, systems can synchronize updates over distributed replicas and provide a consistent state recovery in the presence of faults. However, logs account for a significant overhead on fault-tolerant applications' performance, and many studies present alternatives to alleviate servers from such costs. This paper proposes an approach to reduce log footprint by safely and efficiently discarding entries from logs. The expected benefits are twofold: minimize durability costs and speed up recovery. Besides shrinking logging information, the proposed technique splits the log into several files and incorporates strategies to reduce logging overhead, such as batching and parallel I/O. The proposed approach was compared to a standard logging scheme using realistic workloads. Results demonstrate that our logging approach is capable to generate compressed logs and reduce recovery time, imposing half the throughput overhead of a standard logging scheme.*

1. Introduction

Logs are nested in the hearth of many distributed applications. By providing a single sequence of records ordered by time without the need of a physical clock, and ensuring durability, logging mechanisms play a central role in the developing of database management systems and key-value stores [Mohan et al. 1992], replication and coordination protocols [Lamport 1978, Junqueira et al. 2011, Ongaro and Ousterhout 2014], and data integration [Kreps et al. 2011, Liu et al. 2014]. Database management systems entrust logs the role to synchronize updates over various data structures and indexes, allowing a safe state recovery to replicas [Kreps 2014]. Logs can also be used as a consistency mechanism to order input commands to replicated services, as is the case of replication protocols, such as *State Machine Replication* (SMR) [Lamport 1978]. and primary backup [Budhiraja et al. 1993].

Logs are still the *de facto* technique to provide fault tolerance on a distributed system [Zhang et al. 2015]. On a standard logging approach, every command is persisted to stable storage. *Pessimistic logging* schemes ensure a greater consistency level by logging commands before executing them. Especially when considering applications with strict consistency requirements, this scheme must be implemented to guarantee safety in the

presence of catastrophic failures, such as power outages or simultaneous failures on replicas sites. Although synchronous writing achieves a recovery point objective of zero lost data, I/O costs may represent a major overhead on command execution [Yao et al. 2016].

Besides adding extra costs during the normal operation, logging directly affects recovery time. Since traditional logging mechanisms typically do not benefit from operations semantic, it is unknown to the recovery protocol whether a command result is later overwritten or its execution does not modify the application's state. Thus, the entire sequence of logged commands must be replayed during recovery to reach a consistent state. Principally for high throughput systems, the processing of large logs during recovery incurs significant downtime periods. A fast recovery procedure is always wanted to keep up with availability levels, a growing concern in today's online systems [Chaczko et al. 2011, Mendizabal et al. 2017], where any downtime can result in substantial losses [Clay 2013]. Even considering general practices to provide durability while leveraging log costs during recovery, such as *checkpointing*, logs are still needed and account for non-negligible performance overhead [Bessani et al. 2013, Çelikel and Ovatman 2021].

In this paper, we present an approach to minimize log management costs on log-based protocols. By safely discarding unnecessary log entries, the proposed logging approach reduces the amount of recorded information necessary to restore a consistent state. It means that recovering from the reduced log leads to the same result as replaying an entire log sequence, but at a lower cost. A smaller sequence affects both transferring and installation of logs, and can directly reduce the application's downtime period, thus increasing availability levels. Although shrinking the log at runtime might seem inefficient at first glance, the proposed logging enables concurrency between commands execution and persistence, demonstrating a comparable performance to the traditional approach in systems equipped with a single storage device. As the proposed approach favors parallel I/O, it outperforms traditional logging when configured with multiple storage devices. We implemented a key-value store prototype and compared our approach with a standard logging approach. Test scenarios reproduce realistic workload patterns from the consolidated Yahoo! Cloud Serving Benchmark (YCSB) [Cooper et al. 2010].

The remainder of this paper is organized as follows. Section 2 presents the system model and assumptions. Section 3 briefly introduces the standard log-based protocols. Section 4 presents our approach, implementation aspects, and recovery implications. Section 5 describes our experimentation environment, the methodology, and the evaluation results. Section 6 presents some related work, and Section 7 concludes this paper.

2. System model and assumptions

We consider a distributed system composed of interconnected processes, with a limited number n of replicas defined by the set $R = \{r_1, r_2, ..., r_n\}$ and an unbounded set $C = \{c_1, c_2, ...\}$ of client processes. The system is asynchronous, *i.e.*, there is no upper bound to the processes speed and message delays. To ensure liveness, we assume the existence of synchronous intervals during which messages sent between processes are received and processed with a bounded delay.

We assume the crash-recovery failure model and exclude malicious or arbitrary behavior (*e.g.*, no Byzantine failures). A process may fail by crash and subsequently recover, although they are not obligated to recover once they failed. Failures may be

correlated, leading to catastrophic failures, where up to n simultaneous failures may happen. Power outages on replicas site or deterministic bugs in the service exemplify this behavior. Replicas are equipped with volatile memory and stable storage. Upon a crash, a process loses the content of its volatile memory, but the content of its stable storage is not affected. Stable storage data cannot be corrupted or lost. Therefore, state information saved on this device during failure-free execution can be used for recovery.

3. Log-based protocols

Logs can be seen as an append-only sequence of records ordered by time. Log records may have different meanings depending on the application. For the sake of simplicity, we assume each record stores an application command. Updating commands are represented by the command $w(k, v)$ and reading commands by $r(k)$, where w writes the variable given by the key k with a value v, and r reads the value associated with the key k. A logger process stores records in the log following the order they arrive. Each entry appended to the log is assigned to a unique and sequential entry number.

Figure 1 illustrates a standard log-based protocol, where each command is logged to stable storage before the command reply is sent to clients. The upper line E represents the execution of application's commands by an *executor task*, whereas S illustrates the *store task*. The received commands ($w(x, 14), r(x)$, *etc.*) are forwarded to S, where they are written to the log in the order they arrive, given by the indexes $0, 1, ..$, and so forth. In practical terms, the store task can be exemplified by a synchronous writing system call. Thus, records $log\ 0$, $log\ 1$; .., are persisted by successive writings to the log file.

In the literature, most log-based protocol implementations follow a similar recovery approach. The first step is to retrieve the latest application's checkpoint from local storage or a replica further ahead in the processing of commands. By installing the snapshot, the replica is informed of the latest index (*i.e.*, the log entry) λ reflected on the installed state. Then, it determines the lower-bound index $i = \lambda + 1$ for the remaining interval of commands to be recovered. If no checkpoint is received, i is set to 0. Finally, the replica retrieves a log suffix starting in i from its local storage or another replica.

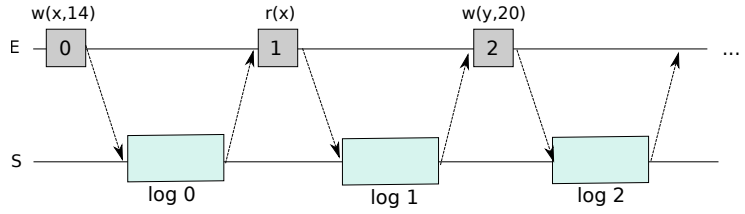


Figure 1. Standard logging approach (SL). Commands are logged to persistent storage before any reply is sent to clients.

4. Reducing log state

The key idea for shrinking the log size is to consciously avoid the logging of unnecessary commands to reach a consistent state. The omission of some commands transferred to the recovering replicas must not modify the final state after recovery. For logging purposes, incoming commands are seen as batches of commands, and the *logging reduction* is performed per batch. The reduction procedure discards all unnecessary commands in the batch, and only the necessary ones are recorded to the log.

To demonstrate the potential of safely discarding unnecessary log entries, we adopted a key-value store data model advocating its representativeness for various applications. Our approach relies on the application's command semantics to reduce log footprint. Applications execute single-variable read, $r(k)$, and write, $w(k, v)$, commands over an address space. Read operations and writes to values that are subsequently overwritten are examples of unnecessary log entries. Only the last write command per batch needs to be logged to reach a consistent state.

Figure 2 illustrates the reduction and persistence of commands into the log. The sequence of incoming commands is evaluated in the form of successive virtual batches, given by $batch\ 0$, $batch\ 1$, and $batch\ n$. Each batch is reduced so that only the necessary commands are kept. Solid lines represent the projection of these commands to the persistent log. For instance, only commands with indexes 1, 4, and 5 are stored in the log. Differently from the traditional logging, in this approach, the log is split into several files, one per batch. These files may contain 0 to m commands, where m is the batch size. As shown in the figure, one command is stored in the first log file, two commands in the second file, and the last file is empty because only unnecessary commands are present in the n^{th} batch. Notice that the elimination of write operations to outdated values is possible only for successive writes in the same batch. To illustrate this, let us assume the commands $c_1 : w(x, 10)$ and $c_2 : w(x, 20)$ are associated with indexes 0 and 1 in the figure. In this case, only c_2 is recorded in the log. A different case will occur if c_1 and c_2 are associated with indexes 1 and 4. Although c_1 and c_2 update the same variable, both must be appended to the log.

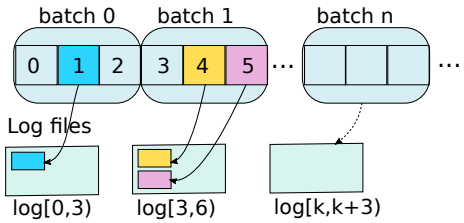


Figure 2. Reduction of batch of commands and split log.

4.1. Design aspects

The reduction procedure delimits the commands in a batch by establishing a batch size of m commands. That means the system does not need to pack commands in a batch explicitly. After m commands have been reduced, the logging procedure stores the resulting commands in the log and starts reducing the next m commands.

During the reduction of a batch, a hash table is used to keep information of which keys are updated by the commands. The table indexes serialized commands into their corresponding keys. Thus, a write operation issued to an already populated key overwrites the current command associated with that key. The proposed strategy separates the *reduction procedure* from the logging in stable storage. Thus, while a set of commands resulting from a reduction is being persisted, other batches can be reduced in parallel. When multiple storage devices are available, writings to devices can also occur in parallel, boosting I/O throughput.

The concurrent execution model between log reduction and *log persistence* adopts multiple reduction tables and log writers. When the reduction procedure is filling one

table, other tables' content can be flushed to the persistent storage by log writers. Tables are accessed in mutual exclusion by the reduction procedure and log writers. A mutex enforces the synchronization over a cursor variable. The cursor indicates which table is being used by the reduction procedure. Besides, each table has a lock to ensure that only one log writer is persisting the table data to the log. If the reduction procedure tries to acquire the lock to perform a batch reduction, it will wait until the table's contents are fully flushed into stable storage.

Table swapping and log persistence are ruled by command intervals according to the batch size. Right after reducing the last batch command over a certain table t_n, the reduction procedure acquires the mutex, signals a log writer to record t_n data in the log, updates the cursor to the next table following a circular order, and releases the mutex. The rapid cursor update allows new batch operations to be immediately applied over the next table. The log writer writes all serialized values from t_n to the stable storage with its corresponding $[first, last]$ interval as metadata. Finally, it resets t_n and unlocks it.

Figure 3 depicts an execution trace with two hash tables, 2 log writers, and a batch size of 3 commands. As shown, the reduction procedure only modifies the current table if the incoming command is an update. If a command updates a key for the first time, a new entry in the table is created; otherwise, the previous command is overwritten. After finishing the batch processing (*i.e.*, after executing commands 0 to 2), the table contains only the most recent write commands associated with x and y. The logging procedure then signals the log writer associated with $table\ 0$ and updates the cursor for the next table (*i.e.*, $table\ 1$). Log writer 0 starts the persistence of $table\ 0$. Although only commands $w(x, 21)$ and $w(y, 20)$ are present in the log, they represent the reachable state by executing commands 0 to 3. The whole command interval information is kept on the log file header, and it is crucial to ensure safety during recovery. After the log is fully persisted to the stable storage, $table\ 0$ is reset, and it is now free to receive commands from another batch. In parallel to the $table\ 0$ logging, the reduction procedure executes commands from the next batch (i.e., $batch\ 1$) by using $table\ 1$. When the whole batch is processed, a signal is sent to log writer 1, so it locks $table\ 1$ and saves its content to the log. After finishing the persistence of the table, the log writer releases the lock.

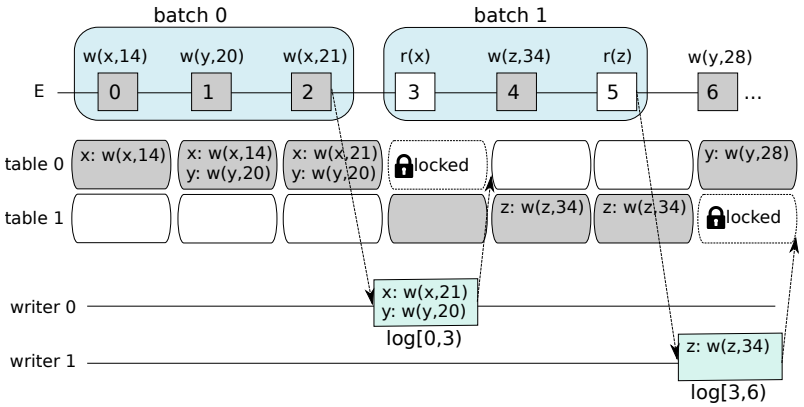


Figure 3. Concurrent execution between log reduction and log persistence.

The I/O operations are substantially more expensive than updates in the hash table. So, it is expected that log writers take longer to flush the content of a table to stable storage

than the reduction procedure to fill the table. It might lead to a situation where a batch is going to be reduced with the support of a table being accessed by a log writer. Since log writers lock the table before starting their job, the reduction procedure would wait for that table to be released. This situation does not hurt safety, and multiple storage devices can alleviate the contention to the tables access. In this case, every writer would store log files in different devices, improving throughput by minimizing the I/O bottleneck. This approach assures a sequential ordering of logs being written on the same storage device.

4.2. Recovery protocol

The recovery protocol allows a replica to restore a consistent state and catch up with other system replicas. Recovering a failed replica requires retrieving the commands the replica missed while it was down. These commands can be obtained from other replicas' logs. At initialization, a recovering replica checks the last instance i stored in its local log and sends a request to other replicas asking for more up-to-date logs. Since log files store in their metadata the batch they correspond to and the lower and higher indexes of commands in that batch, retrieving the highest index value is straightforward. A request informing index i is sent to other replicas, and those ahead in the execution reply with their highest index number h. The recovering replica requests the log represented by all files within the $[i, h]$ interval to the correspondent replica.

The retrieved log is composed of a set of files, each one representing a reduced set of commands in a batch. These files are processed in ascending order so that the commands' execution follows the same delivered order. If, after the recovery, a replica receives any command with an index j, where $j \leq h$, it only has to ignore and not execute it, since its effects would have already been applied to its state. If $j > h + 1$, there are missing commands that were not recoverable by the log. In this case, the replica keeps all the incoming requests in a temporary queue and restarts the recovery procedure. Eventually, some replica will have a h index higher than j and the gap of commands would be fulfilled.

5. Experimental Evaluation

The experimental evaluation of our approach seeks to answer two main questions:

1. **Does our protocol allow a faster recovery procedure?** We evaluate the generated log files on different interval configurations against the traditional approach, comparing the number of commands and total file sizes. We appraise a recovery time reduction due to the minimal number of commands on our persistent log state and lower storage usage.
2. **Does our approach represent a significant overhead on the application's performance?** We evaluate this by comparing the standard logging scheme's execution against our approach with different batch sizes and number of storage devices. Our analysis compares the throughput and latency impact of these configurations on different workloads.

5.1. Key-value store prototype

We implemented a key-value store prototype in Go.[1] The prototype is configurable to execute the standard or the proposed logging approach. Read and write operations are illus-

[1]Prototype implementation is public available at: `https://github.com/Lz-Gustavo/beelog/tree/master`

trated by the commands $r(k)$ and $w(k, v)$, where keys are represented as integer numbers and values as fixed-size strings of 100 bytes. Commands are logged in batches, and the replies to the batch commands are sent to clients only after logging and execution. All persisted commands are serialized with Protocol Buffers.

The load generation produces read and write commands according to an input file following a specific workload pattern. However, we abstract the client-side and network layer, focusing only on the execution of the requests. Thus, the only considered costs are those caused by command execution and logging. This decision aims to avoid the interference of performance bottlenecks unrelated to the logging itself.

5.2. Workloads and configuration

For the experiments, we utilized a subset of the standard Yahoo! Cloud Serving Benchmark (YCSB) [Cooper et al. 2010] workloads with two variations of YCSB-A, named YCSB-AW and YCSB-AWL. We opted for YCSB due to its large utilization as a benchmark for storage engines and cloud services [Yu et al. 2016, Chandramouli et al. 2018]. To cope with our proposed model, both inserts and updates are mapped into write operations. The corresponding read (r) and write (w) percentages and request distributions of each workload are shown below:

- **YCSB-A:** 50% of reads and 50% of writes, following a uniform distribution;
- **YCSB-B:** 95% of reads and 5% of writes, following a uniform distribution;
- **YCSB-C:** 100% of reads, following a uniform distribution;
- **YCSB-D:** 95% of reads and 5% of writes, following a latest distribution (*i.e.*, recent accessed variables are more likely to be accessed next);
- **YCSB-AW:** 100% of writes, following a uniform distribution;
- **YCSB-AWL:** 100% of writes, following a latest distribution.

The standard workloads YCSB-A, YCSB-B, and YCSB-C, vary in the percentage of read and write commands, but all of them follow a uniform distribution to choose which key is going to be accessed. YCSB-D uses a different distribution that increases the chances of subsequent access to the most recent updated keys. This behavior is very common in social networks, as it simulates access to trending topics. Our custom-defined workload YCSB-AW constitutes a write-only workload with records being uniformly accessed. It represents the worst-case scenario for the proposed technique where no command discard is stimulated. The YCSB-AWL is also write-only, but most recent records are more likely accessed with the latest distribution.

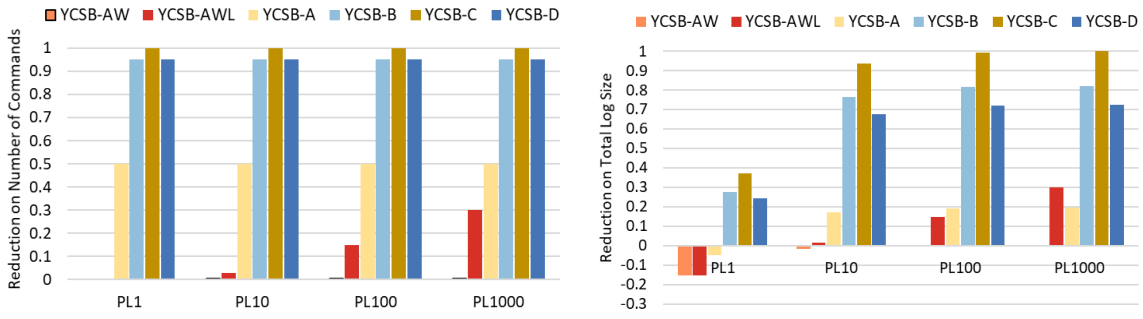
All workloads consider 10^6 distinct keys during load distribution. Registered logs captured during the experiments account for the execution of 10^6 requests. Experiments were executed on the *Emulab Utah* [White et al. 2002] cluster, utilizing a Dell Poweredge R430 node equipped with two 2.4 GHz 8-Core Xeon E5-2630v3 processors; 64 GB 2133 MT/s DDR4 random access memory; and two 1 TB HDD with 7200RPM. The node operates under a Ubuntu 18.04LTS image. Go binaries were compiled on go-1.15.

5.3. Recovery impact

We first analyze the generated logs to conjecture about the recovery impacts caused by our proposed logging approach (PL) by comparing it against the standard logging (SL). Figure 4(a) shows the reduction achieved on the number of commands for each combination of

workload and batch size, with data normalized to SL values. The x-axis shows the results when batch size is set to 1, 10, 100, and 1000. For all workloads, the number of commands written on SL logs was exactly 10^6. The same can be said for YCSB-AW since it was not observed discarding of commands in this case. So, it presents a 0% reduction compared to the SL regardless of the batch size. It is explained by the write-only workload with records evenly distributed. For YCSB-AWL, the latest distribution shows an interesting scenario by stimulation the removal of write operations over recent outdated variables on our technique. As can be seen, the greater the batch size, the greater the odds of such operations being identified and, thus, safely discarded during log procedures. On this same workload, it is observed gains of 32% fewer commands on PL-1000. Read-intensive loads such as YCSB-B, YCSB-C, and YCSB-D depict our best scenarios with all commands consciously eliminated for YCSB-C, a 100% reads workload, and more than 90% of reduction seen for YCSB-B and YCSB-D.

Figure 4(b) shows the total log size reductions compared to SL. On the standard approach, the log size observed were: 130.64MB for YCSB-AW and AWL; 80.99MB for YCSB-A; 36.42MB for YCSB-B; 31.46MB for YCSB-C; and 41.26MB for YCSB-D. On the PL-1 scenarios, where a logging procedure is triggered after each command, our approach presents a penalty of $\approx$17% on total log sizes for workloads YCSB-AW and YCSB-AWL and a 5% increase for YCSB-A. This is explained by the fact that PL generates a new log file with its proper metadata on every batch reduction, and the sum of all these generated files yields a slight size increase when compared to an individual log file generated on SL. This effect is not noticed for other workloads, where considerable size reductions are shown for read-intensive workloads. Considering larger batches, we observe significant improvements for various workloads. For batch sizes equal or superior to 10 commands, YCSB-A maintains a $\approx 20\%$ total size reduction. YCSB-AWL shows incremental benefits on larger batches, with a 30% decrease in log sizes for PL1000. Substantial differences can be seen for YCSB-C, B, and D, where a nearly 100% decrease is shown for the former and $\approx 80\%$ for the other two for batch sizes of 1000 commands.



(a) Number of commands (b) Log size reductions

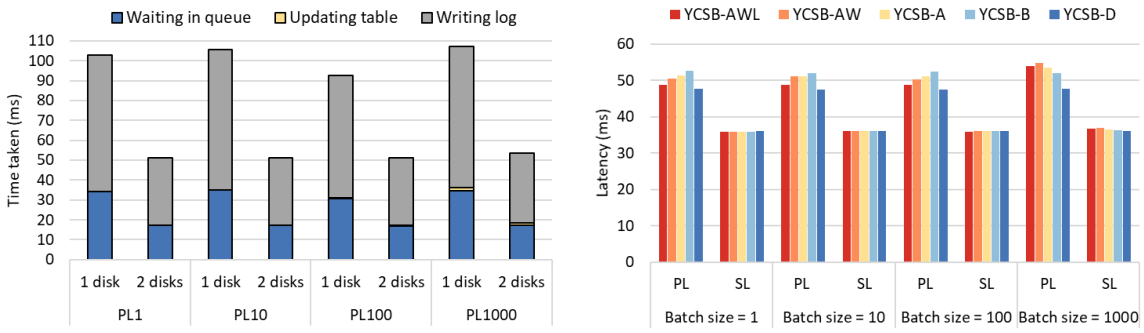
Figure 4. Reduction of recovery logs for 1M execution commands: (a) number of commands; and (b) total size normalized to SL approach values.

5.4. Latency and throughput assessment

Figure 5(a) depicts a fine-grained analysis for PL's latency values, considering only the YCSB-A workload and varying one and two storage devices. We break down latency measurement to capture each time taken to (i) write the first command on a log batch,

reporting the costs related to table swapping and synchronization; (ii) update an entire table, representing the time taken to reduce a batch; and (iii) write log to stable storage, measuring the time taken on the synchronous log flush to stable storage. As shown, flushing to stable storage accounts for $\approx 66\%$ of the time on all studied batch sizes, with a low variation on each of them. By doubling the number of disks, we approximately halve the flush measurement and waiting time for the first command to be recorded on a table. This latter effect happens because log contention is decreased when exploiting parallel I/O to multiple devices, thus accelerating table swapping. In this sense, we estimate that more disks, together with more tables, could represent significant improvements by reducing overall latency on log persistence for our approach.

In Figure 5(b), we analyze the average latency measured for log persistence, comparing PL with its two disks configuration against SL on different workloads and batch sizes. At first glance, both strategies manifest low variations for latency values as batch size increases. This result is explained by the fact that synchronous log flush to stable storage is orders of magnitude higher than the time needed for batching commands, even considering processing in between. Also, SL values do not vary with the simulated workload since this strategy logs every command independently of its operation or accessed key. That is different for PL because of commands discard, where latency values differentiate upon workload and present lower values on read-intensive scenarios. Considering all studied settings, PL displays a $\approx 30\%$ average increase in log persistence latency when compared to SL. Although increasing latency, our approach favors throughput, as discussed next.

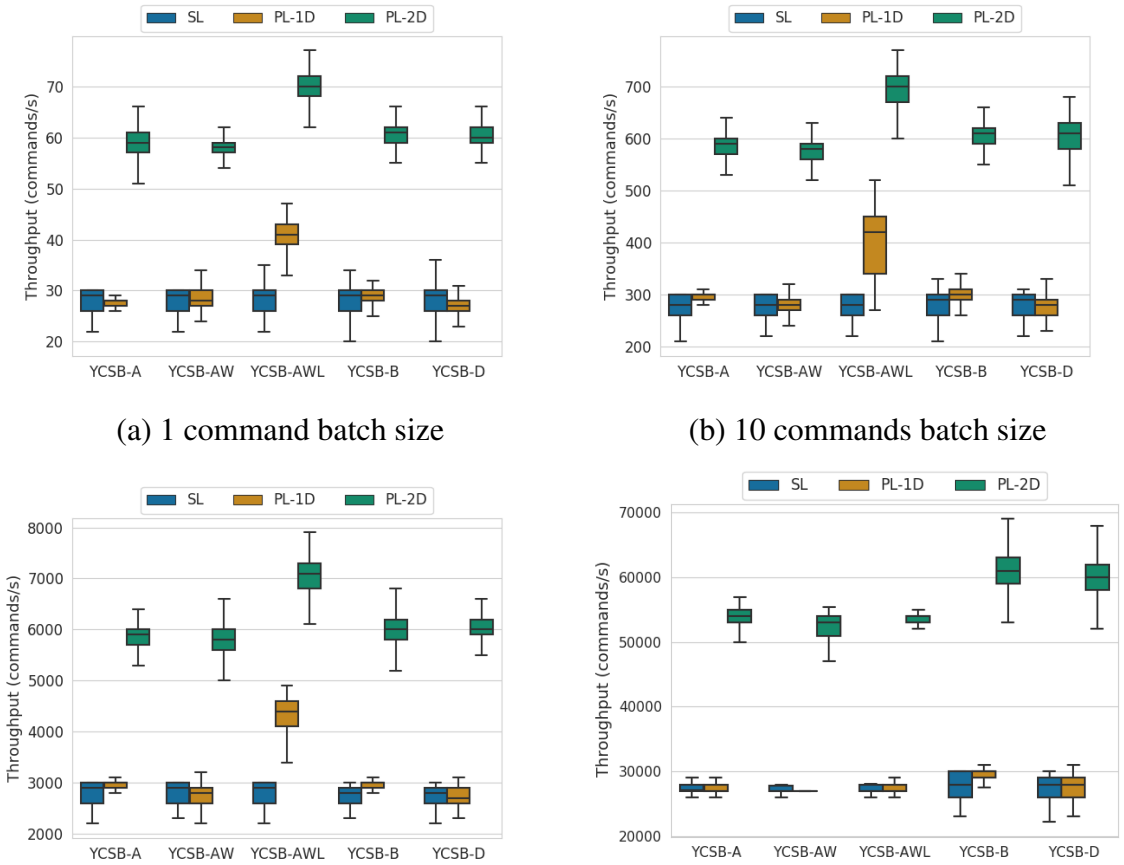


(a) Latency breakdown

(b) Average latency

Figure 5. Evaluation of Latency breakdown for PL YCSB-A (a) and the average latency of SL and PL strategies for each workload (b), considering different batch sizes.

Figure 6 depicts throughput box plots for five workloads, considering values of SL, and PL with 1 (PL-1D) and 2 (PL-2D) storage devices being used. Each graph shows the throughput of these three scenarios, considering the same batch size configuration. As shown for all workloads and configured batch sizes, our approach stands with similar throughput values for PL-1D compared to SL. Especially on YCSB-AWL, a workload with only write operations and latest distribution (*i.e.*, that mimics a scenario where most recent records are constantly updated), PL-1D presents significant improvements, being $\approx 50\%$ on median throughput for 1000 command batch size in Figure 6(c). This effect is a consequence of eliminating subsequent write operations, where only the latest update of a particular key is kept for each batch of commands. The removal of writes incurs a



(a) 1 command batch size

(b) 10 commands batch size

(c) 100 commands batch size

(d) 1000 commands batch size

Figure 6. Throughput analysis for different workloads and batch sizes with SL, PL-1D and PL-2D configurations.

reduction in data usage compared to SL, being more prominent than eliminating reads due to the value size associated with each operation. Each read takes only an integer argument indicating the corresponding key, while a write also records a 100B byte sequence value associated with its key.

Regarding PL-2D scenarios, consistent gains can be seen for all workloads. This effect emerges because of the I/O bottleneck's mitigation perceived during log flush to stable storage. PL-2D median values show a $\approx 100\%$ increase in throughput for most workloads on 1, 10, and 100 batch size configurations.

6. Related work

Despite being a performance limiter, logging strategies are crucial for providing durability and state recovery. This section highlights related approaches that rethink logging mechanisms to reduce overhead and recovery time.

ARIES [Mohan et al. 1992] implements a data-level logging approach to keep track of tuple values instead of logging operations or transactions. It aims to accelerate state recovery by overwriting old values from their data-log structure. This strategy allows a parallel recovery since only a single value is recorded for each key. A drawback

of this approach is the overhead caused during normal execution. Being a verbose logging method, it excels in persistent storage databases where I/O costs are orders of magnitude higher than transactions' processing time. However, it yields significant overheads for in-memory datastores. In [Yao et al. 2016], the authors propose an adaptive approach that alternates between ARIES *data logging* and *command logging* methods during the execution. They adjust the percentage of data logging versus command logging on-the-fly based on a dynamically parsed cost model and online heuristics. Their motivation is to reduce the costs originated by the heavy-weight ARIES logging method, while still taking advantage of data logging recovery. Even though ARIES allows a parallel recovery procedure, its extensive use represents an I/O bottleneck on various workloads, whereas command logging significantly reduces transactions' processing costs. Our proposed approach bears some similarities with ARIES, such as the overwrite of the outdated entries in the log, but it implements a command logging strategy. While ARIES overwrites old *values*, our approach discards unnecessary *commands* for recovery. Our logging scheme demonstrates low overhead by separating the log reduction procedure from its persistence into stable storage. In particular, when multiple storage devices are available, it can even increase the system's overall throughput. With regard to recovery, our approach hastens state recovery by reducing the number of commands to be transferred and executed.

Apache Kafka [Kreps et al. 2011, Narkhede et al. 2017] is a popular event streaming platform that serves as a publish/subscriber middleware for real-time data processing. To viably maintain long-term data, Kafka implements a log compaction approach that aims to accelerate state recovery and reduce memory usage by eliminating intermediate state changes from a topic log. This compaction procedure is asynchronously executed by a set of *cleaner threads*. Cleaners treat the log as a combination of two segments: the clean portion that corresponds to already-compacted messages, storing only one value for each key; and the dirty portion, that contains messages written after the last compaction. The compaction procedure first executes over the dirty segment, arranging an in-memory hash map that stores the offset of the latest message for each key. After, the cleaner traverses all clean records, starting from the oldest one and checks if keys are on the map. If so, it decides whether the record must be kept (*i.e.*, if it is not on the offset map) or later deleted. By default, Kafka executes compaction whenever half of the topic contains dirty entries, since the compaction procedure compromises message throughput on the topic. Different from Kafka, we tackle compaction during the log procedure itself, where we immediately discard unnecessary operations and overwrite updates while tracking its corresponding segments. Our execution model aims to concurrently flush logs to stable storage while logging new operations instead of asynchronously compacting the log.

In Taurus [Yu et al. 2016], the authors present an approach that relaxes the sequential logging by tracking fine-grained dependencies among transactions. On uniform distributed workloads Taurus can perform both logging and recovery in parallel. Taurus classifies dependencies into three categories to arrange a global dependency graph. *Read-after-write* transactions are compressed during recovery by exploiting the fact that the log is flushed to persistent storage sequentially. *Write-after-write* transactions are never relaxed and enforce a *must happen before* relation between two transactions during recovery. *Write-after-read* dependencies are always discarded; they do not constraint commit order because reads do not leave side effects in the system. This procedure allows logging on different devices in arbitrary orders, in addition to enabling faster recovery. We

implement a similar approach by allowing logs to be concurrently flushed to stable storage, exploiting different storage devices if available, but we restrict this granularity to the configured batch size and assure a sequential ordering to logs being written on the same device. On our approach, we tackle dependencies among operation during the logging procedure, where reads are idempotent and unnecessary for state recovery, and writes are only dependent if they operate over the same key. Once only one write operation is retrieved per key on each generated log file, parallel processing of commands on each file is allowed as long as subsequent files are processed in the ascending order of log indexes.

RocksDB [Facebook 2013] uses a *Log-Structure Merge-tree* (LSM) to represent the application's state. During execution, a LSM-based storage first buffers all writes in memory using an append-only operation over a mutable structure, and subsequently flushes them to stable storage in a form of a new node of the tree. To manage the database size, a compaction procedure runs periodically. Compaction works by merging different tree nodes, checking for overlapping key ranges and removing intermediate writes and deletes. Although executing a similar approach to discard operations as our technique, both compaction strategies differ in terms of their goal. The LSM architecture is utilized to represent an application's state. The compaction algorithm allows an efficient state representation and maintains a manageable size to preserve the database's performance by running asynchronously at sparse periods. Differently, our approach shrinks logged information on-the-fly in order to allow a faster recovery procedure.

Corfu [Balakrishnan et al. 2013] is a distributed and shared log that allows client operations to run in parallel. During operation, Corfu's clients maintain a local projection map that stores record references to physical logs divided into *pages* distributed across a cluster of nodes. Concurrent operations indexed to pages located in distinct nodes run in parallel, improving throughput and scalability. Similar to the Corfu, in [Xavier et al. 2020] the authors present a complementary study regarding decoupled logging for SMR applications. The results indicate that shared logs can easily attend several clients, not compromise application's performance, and incur in large monetary savings for cloud infrastructures. Other hacks and tricks allow the system to use storage devices efficiently. For instance, in [Bessani et al. 2013], authors propose parallel logging, which attempts to postpone and batch synchronous writes in order to reduce their number and alleviate their latency. Whereas in [Chandramouli et al. 2018], the authors present an alternative log structure called *HybridLog*, which allows in-place updates for a mutable portion of the log. These strategies are orthogonal and could be combined to our approach once they aim to optimize performance of I/O operations and logging management.

In [Mendizabal et al. 2017], the authors present alternatives for state recovery in Parallel State Machine Replication (P-SMR – *Parallel State Machine Replication*). This replication model is an extension of the traditional SMR, which allows parallel execution of independent commands to achieve higher throughput. *Speedy Recovery* is a recovery protocol that explores the semantics of application's commands to map possible dependencies between commands, considerably reducing the recovery time when exploring the parallel execution of independent commands. This dependency identification is performed by combining three strategies: evaluation of commands in batch, in which dependencies are considered when analyzing the execution of a whole group of commands; fast conflict detection, where each set of grouped commands has its own signature that

represents all variables impacted by the execution of the batch; and an efficient dependency handling, since the detection of dependency between batches of commands is done through bitmap comparisons. Their protocol speeds up recovery by anticipating the execution of incoming commands that do not depend on the log. Therefore, new commands can be processed while the log is retrieved and processed by the recovering replica. Our method follows a different approach, where we focus on log reduction. However, the strategies are complementary. It would be beneficial to reduce the log size and still anticipate the execution of incoming commands while the logging is being processed.

7. Conclusion

This paper presented a logging approach that exploits application semantics to safely discard entries from command logs, delivering reduced log files that permit a faster state recovery by benefiting both transferring and log installation. Although shrinking the log, the state achieved by processing the log of commands is identical to the state produced by the execution of a standard unmodified log. In order to alleviate I/O bottlenecks and reduce logging overhead, our approach implements optimizations regarding log management, such as batching and parallel I/O.

Results show that our approach can produce reduced logs with minimal impact on the application's performance, exhibiting less overhead than a standard logging scheme on most analyzed workloads due to command discard and concurrently execution. For instance, in balanced workloads composed of 50% of reads and 50% of writes, our approach delivers a recovery log with 50% fewer commands and 20% smaller file size. When equipped with a single storage device, our technique shows similar throughput against the standard log, and double throughput on median values when exploiting parallel I/O with two disks. By separating log reduction from persistence, the approach has demonstrated to scale-up with the addition of more persistence devices.

References

Balakrishnan, M., Malkhi, D., Davis, J. D., Prabhakaran, V., Wei, M., and Wobber, T. (2013). Corfu: A distributed shared log. *ACM Transactions on Computer Systems (TOCS)*, 31(4):1–24.

Bessani, A., Santos, M., Felix, J., Neves, N., and Correia, M. (2013). On the efficiency of durable state machine replication. In *USENIX Annual Technical Conference*, pages 169–180.

Budhiraja, N., Marzullo, K., Schneider, F. B., and Toueg, S. (1993). The primary-backup approach. *Distributed systems*, 2:199–216.

Çelikel, Ö. and Ovatman, T. (2021). Distributed application checkpointing for replicated state machines. *Scalable Computing: Practice and Experience*, 22(1):67–79.

Chaczko, Z., Mahadevan, V., Aslanzadeh, S., and Mcdermid, C. (2011). Availability and load balancing in cloud computing. In *ICCSM, Singapore*.

Chandramouli, B., Prasaad, G., Kossmann, D., Levandoski, J., Hunter, J., and Barnett, M. (2018). Faster: A concurrent key-value store with in-place updates. In *Proceedings of the 2018 International Conference on Management of Data*, pages 275–290.

Clay, K. (2013). Amazon.com goes down, loses $66,240 per minute. https://www.forbes.com/sites/kellyclay/2013/08/19/amazon-com-goes-down-loses-66240-per-minute/.

Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. (2010). Benchmarking cloud serving systems with YCSB. In *1st ACM SoCC*.

Facebook (2013). RocksDB. https://rocksdb.org/.

Junqueira, F. P., Reed, B. C., and Serafini, M. (2011). Zab: High-performance broadcast for primary-backup systems. In *IEEE DSN*, pages 245–256. IEEE.

Kreps, J. (2014). *I Heart Logs: Event Data, Stream Processing, and Data Integration*. O'Reilly Media, Inc.

Kreps, J., Narkhede, N., Rao, J., et al. (2011). Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, volume 11, pages 1–7.

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565.

Liu, X., Iftikhar, N., and Xie, X. (2014). Survey of real-time processing systems for big data. In *Proceedings of the 18th International Database Engineering & Applications Symposium*, pages 356–361.

Mendizabal, O. M., Dotti, F. L., and Pedone, F. (2017). High performance recovery for parallel state machine replication. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 34–44. IEEE.

Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., and Schwarz, P. (1992). Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS)*, 17(1):94–162.

Narkhede, N., Shapira, G., and Palino, T. (2017). *Kafka: the definitive guide: real-time data and stream processing at scale*. O'Reilly Media, Inc.

Ongaro, D. and Ousterhout, J. (2014). In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference*, pages 305–319.

White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., and Joglekar, A. (2002). An integrated experimental environment for distributed systems and networks. *ACM SIGOPS*.

Xavier, L. G. C., Dotti, F. L., Meinhardt, C., and Mendizabal, O. M. (2020). Scalable and decoupled logging for state machine replication. In *38th Brazilian Symposium on Computer Networks and Distributed Systems (SBRC)*, pages 267–280. SBC.

Yao, C., Agrawal, D., Chen, G., Ooi, B. C., and Wu, S. (2016). Adaptive logging: Optimizing logging and recovery costs in distributed in-memory databases. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1119–1134.

Yu, X., Zhu, S., Kaashoek, J., and Pavlo, A. (2016). Taurus: A Parallel Transaction Recovery Method Based on Fine-Granularity Dependency Tracking. *CoRR*.

Zhang, H., Chen, G., Ooi, B. C., Tan, K.-L., and Zhang, M. (2015). In-memory big data management and processing: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 27(7):1920–1948.