UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Luiz Gustavo Coutinho Xavier

Efficient Log Compaction by Safely Discarding Commands

Florianópolis
2023

Luiz Gustavo Coutinho Xavier

Efficient Log Compaction by Safely Discarding Commands

Dissertação submetida ao Programa de Pós-Graduação em Ciência da Computação para a obtenção do título de mestre em Ciência da Computação.
Orientador: Prof. Odorico Machado Mendizabal, Dr.
Coorientadora: Prof. Cristina Meinhardt, Dr.

Florianópolis
2023

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Xavier, Luiz Gustavo Coutinho
Efficient Log Compaction by Safely Discarding Commands / Luiz Gustavo Coutinho Xavier ; orientador, Odorico Machado Mendizabal, coorientadora, Cristina Meinhardt, 2023.
67 p.

Dissertação (mestrado) - Universidade Federal de Santa Catarina, Centro Tecnológico, Programa de Pós-Graduação em Ciência da Computação, Florianópolis, 2023.

Inclui referências.

1. Ciência da Computação. 2. Sistemas Distribuídos. 3. Tolerância à Falhas. 4. Compactação de Log. 5. Etcd. I. Mendizabal, Odorico Machado. II. Meinhardt, Cristina. III. Universidade Federal de Santa Catarina. Programa de Pós Graduação em Ciência da Computação. IV. Título.

Luiz Gustavo Coutinho Xavier

Efficient Log Compaction by Safely Discarding Commands

O presente trabalho em nível de mestrado foi avaliado e aprovado por banca examinadora composta pelos seguintes membros:

Domingos Soares Neto, Msc.
PicPay

Prof. Márcio Bastos Castro, Dr.
Universidade Federal de Santa Catarina

Prof. Paulo Coelho, Dr.
Universidade Federal de Uberlândia

Certificamos que esta é a **versão original e final** do trabalho de conclusão que foi julgado adequado para obtenção do título de mestre em Ciência da Computação.

Documento assinado digitalmente
PATRICIA DELLA MEA PLENTZ
Data: 26/02/2023 09:09:50-0300
CPF: ***.243.700-**
Verifique as assinaturas em https://v.ufsc.br

Prof. Patricia Della Méa Plentz, Dr.
Coordenadora do Programa

Documento assinado digitalmente
Odorico Machado Mendizabal
Data: 27/02/2023 09:35:57-0300
CPF: ***.941.170-**
Verifique as assinaturas em https://v.ufsc.br

Prof. Odorico Machado Mendizabal, Dr.
Orientador

Florianópolis, 2023.

ACKNOWLEDGEMENTS

Thanks to my parents for all the support. Among many things, they taught me early on the 6^{th} grade that being called a "nerd" was not something to be ashamed of. This certainly motivated me to pursue higher education. Without them I also wouldn't be able to reallocate to Florianópolis and start this scientific endeavor of pursuing a masters degree. Larissa, thanks for all the love, comprehension, and continuous encouragement throughout these last years.

Thanks to the professors who oriented me along the way. Odorico, my academic advisor since undergrad, taught me so much on the last five years that I think I've never learned as much about computer science from any other person. From his teachings I've developed an eager scientific curiosity for Distributed Systems, which turned me into a better student and professional. His keen attention to details set an example of discipline, which I believe not only contributed to the quality of this work, but also made me a more responsible person. Cristina, my co-advisor (which is just a formality, as she advised on the same level of commitment), woke up on me an interest to pursue a masters degree since attending her Computer Organization and Architecture lectures from undergrad. She taught me so much about scientific methodology and writing, specially when to be "straight to the point" and when to further develop an idea. Like Odorico, her demanding, but on the same level, empathetic way of teaching, showed me how impactful an educator can be on somebody's life. From them I perceived how gratifying a teaching career could be.

Thanks to Paulo Coelho and Márcio Castro for the constructive feed-backs provided since the early stages of this work. Fernando Dotti, who also co-authored some papers, made significant contributions on this work. Special thanks to Domingos for being such an amazing leader. He's one the few people in the technology industry that truly values the importance of academia and research. He was also extremely comprehensive on many instances, either by allowing me to dedicated some time studying for exams, or even work on this dissertation.

Huge thanks to Gabriel, Eduardo, Nathan, Lucas, Andrew, Giordano, Giovanni, Rômulo, Yuri, Luke, Murilo and all the other folks from Discord for helping me out to endure the ups and downs of grad school. Thanks for all the gameplay and laugh that allowed me to retain my sanity. You guys were also some of the few people outside academia that seemed truly interested in knowing about my work when asking "How's the masters going?".

This study was partly financed by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) during the first year of 2020.

The first principle is that you must not fool yourself and you are the easiest person to fool.
- Richard P. Feynman

RESUMO

Logs são amplamente utilizados no desenvolvimento de aplicações distribuídas tolerantes à falhas. Ao registrar entradas em um log global sequencial, diferentes sistemas podem sincronizar atualizações em réplicas distribuídas e fornecer uma recuperação de estado consistente mesmo na presença de falhas. No entanto, logs são responsáveis por uma sobrecarga significativa no desempenho de aplicações tolerantes a falhas, e muitos estudos apresentam alternativas para aliviar servidores de tais custos. Nesta dissertação é apresentada uma abordagem para acelerar recuperação de estado em protocolos baseados em log ao seguramente descartar comandos. A técnica envolve a utilização de um procedimento de compactação de log em tempo de execução, executado de forma concorrente com a persistência e execução de comandos. Além de compactar o log, a técnica proposta o divide em diferentes arquivos, e incorpora estratégias para reduzir a sobrecarga gerada pelo uso de logs, como explorar o agrupamento de comandos em lotes e E/S paralela. Avaliamos a abordagem proposta em dois ambientes distintos: (i) em um ambiente controlado, avaliando nossa solução em protótipo de base de dados chave-valor implementando uma estratégia de log padrão; e (ii) em um ambiente mais realista, implementando nossa abordagem como uma nova funcionalidade do etcd, um banco de dados comercial conhecido na indústria, e comparando-a com sua implementação de log padrão. Utilizando cargas de trabalho do YCSB e explorando diferentes configurações para nossa técnica, como variações no tamanho de lote e número de dispositivos de armazenamento, os resultados demonstram que nossa abordagem de compactação é capaz de reduzir significativamente o tempo de recuperação da aplicação. Em cargas de trabalho onde o acesso a chaves recentes é estimulado, é demonstrado uma redução de até 50% no tamanho do log com uma melhora de 65% no tempo de recuperação quando comparado com o protocolo de recuperação padrão do etcd. Em questão de desempenho, com exceção de um aumento de latência devido à abordagem de lotes implementada, nenhuma outra sobrecarga relevante foi observada.

Palavras-chave: Sistemas Distribuídos. Tolerância à Falhas. Logging. Compactação de Log. Sistemas de Armazenamento. Recuperação de Estado. Raft. Etcd.

RESUMO ESTENDIDO

Introdução

Logs são amplamente utilizados no desenvolvimento de aplicações distribuídas tolerantes à falhas. Na forma de uma sequência durável e ordenada, logs podem armazenar comandos de qualquer aplicação, e desempenham um papel central no desenvolvimento de sistemas de gerenciamento de banco de dados e armazenamentos de valores-chave, protocolos de consenso, e *middlewares* de integração de dados. No entanto, embora amplamente utilizados, logs representam uma sobrecarga significativa no desempenho de aplicações tolerantes à falhas, onde além de adicionar custos extras durante a operação normal, afetam diretamente o tempo de recuperação. Abordagens de log tradicionais usualmente não se beneficiam da semântica de comandos da aplicação, exigindo que toda a sequência de comandos registrados seja reproduzida durante a recuperação para permitir uma recuperação de estado completa e consistente. Especialmente para sistemas de alto desempenho, essa condição resulta no processamento de grandes arquivos de log durante a recuperação, o que incorre em períodos de indisponibilidade. Um procedimento de recuperação rápido é necessário para manter altos níveis de disponibilidade de uma aplicação, uma preocupação crescente em sistemas distribuídos atualmente, onde qualquer período de indisponibilidade pode resultar em perdas significativas.

Objetivos

Neste trabalho é apresentada uma abordagem de log com o objetivo de acelerar a recuperação e minimizar os custos evolvidos no gerenciamento de logs. A técnica proposta realiza a compactação do log em tempo de execução, explorando o descarte de comandos considerados desnecessários para uma recuperação de estado consistente. A ideia é que a recuperação a partir um log compactado resulte no mesmo estado que a reprodução de uma sequência de log completa, mas a um custo menor. Uma sequência de comandos mais curta afeta tanto a transferência quanto a instalação de logs, reduzindo diretamente o período de indisponibilidade da aplicação, e aumentando assim seus níveis de disponibilidade. Além disso, são objetivo específico deste trabalho avaliar a solução proposta em um banco de dados comercial e divulgar toda implementação em regime de *software* aberto.

Metodologia

A técnica proposta envolve a utilização de um procedimento de compactação de log em tempo de execução, executado de forma concorrente com a persistência e execução de comandos. Além de compactar o log, a técnica apresentada o divide em diferentes arquivos, e incorpora estratégias para reduzir a sobrecarga gerada pelo uso do log, como explorar o agrupamento de comandos em lotes e E/S paralela. A avaliação da técnica foi efetuada utilizando cargas de trabalho do *benchmark* YCSB, explorando diferentes configurações da nossa técnica, como variações no tamanho de lote e número de dispositivos de armazenamento. Estes cenários de experimentação foram avaliados sob dois ambientes distintos: (i) em um ambiente controlado, avaliando nossa solução em protótipo de base de dados chave-valor implementando uma estratégia de log padrão; e (ii) em um ambiente mais realista, implementando nossa abordagem como uma nova funcionalidade do etcd, um banco de dados comercial conhecido na indústria, e comparando-a com sua implementação de log padrão. Para ambos os ambientes procura-se entender o impacto gerado pelo uso da nossa abordagem na execução normal da aplicação, onde para isso são analisados os valores de vazão e latência, e no tempo de recuperação, ao

analisar a quantidade de comandos eliminados durante a compactação, o tamanho total dos logs reduzidos, e o tempo tomado durante toda recuperação de estado.

Resultados e Discussão

Em todos os experimentos são realizadas comparações entre a abordagem de *logging* proposta (*Proposed Logging Approach* - PL) com a abordagem de log padrão (*Standard Logging Approach* - SL), onde SL varia dependendo do ambiente utilizado. Ao avaliar a abordagem proposta em um protótipo de banco de dados chave-valor, demonstramos que PL é capaz de produzir logs reduzidos com impacto mínimo no desempenho da aplicação, resultado em uma sobrecarga menor que a imposta por SL na maioria das cargas de trabalho analisadas devido a compactação e execução concorrente entre persistência e execução de comandos. Em uma carga de trabalho balanceada composta por 50% de operações de leitura e 50% de escritas, evidenciamos que nossa abordagem é capaz de entregar um log com 50% menos comandos a um tamanho de arquivo 20% menor. Quando equipada com um único dispositivo de armazenamento, nossa técnica demonstrou valores de vazão semelhantes à SL. Porém, ao explorar E/S paralela com dois dispositivos de armazenamento, foi observado o dobro da vazão nos mesmos cenários. Além disso, ao avaliar a nossa técnica no etcd, um banco de dados de código aberto conhecido no setor de computação em nuvem, mostramos que nossa abordagem pode reduzir significativamente o tempo necessário para se recuperar de um estado de 10^5 comandos, atingindo uma redução de até 65% no tempo de recuperação. Esses benefícios surgem da abordagem de compactação realizada em tempo de execução e pela estratégia de processamento de log *Descending*, que explora o descarte de comandos já executados durante a recuperação. Em questão de desempenho, com exceção de um aumento de latência devido à abordagem de lotes implementada, nenhuma outra sobrecarga relevante foi observada.

Considerações Finais

No trabalho é apresentada uma abordagem de log que explora a semântica de comandos da aplicação para realizar a compactação de estado, entregando arquivos de log compactados que possibilitam um procedimento de recuperação rápido, o que beneficia tanto a transferência quanto o re-processamento de logs. A fim de reduzir a sobrecarga envolvida com o uso de logs e aliviar gargalos de E/S, a técnica proposta explora a execução concorrente entre a compactação e persistência de comandos, e implementa outras otimizações, como explorar o agrupamento de comandos em lotes e E/S paralela. Duas diferentes abordagens de recuperação, denominadas *Naive* e *Descending* também são apresentadas e avaliadas durante a experimentação. Com os experimentos conduzidos foi possível demonstrar benefícios claros no procedimento de recuperação com a utilização da abordagem proposta, às custa de um aumento na latência devido à abordagem de lotes implementada. Trabalhos futuros podem se benefeciar desta dissertação caso desejem: (i) estender o modelo de dados assumido para além de uma aplicação chave-valor; (ii) implementar novas fases de compactação realizadas de forma assíncrona com a execução de comandos para reduzir ainda mais o estado armazenado; (iii) explorar novas configurações ou modificar dinamicamente parâmetros de configuração com base na carga de trabalho submetida; ou (iv) explorar a execução paralela de comandos não conflitantes durante a recuperação do estado compactado.

Palavras-chave: Sistemas Distribuídos. Tolerância à Falhas. Logging. Compactação de Log. Sistemas de Armazenamento. Recuperação de Estado. Raft. Etcd.

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. In this dissertation we propose an approach to accelerate recovery on log-based protocols by safely discarding entries from logs. The technique involves the execution of a compaction procedure during run-time, concurrently with the persistence and execution of commands. Besides compacting logging information, the proposed technique also splits the log into several files and incorporates strategies to reduce logging overhead, such as batching and parallel I/O. We evaluate the proposed approach under two distinct setups: (i) on a controlled environment, by comparing against a key-value store prototype implementing a standard logging scheme; and (ii) on a more realistic scenario, by implementing our approach as a new feature of etcd, a known commercial database in the industry, and comparing it against the database's standard logging implementation. Utilizing workloads from YCSB and exploring different configurations for our technique, such as batch size and number of storage devices, results demonstrate that our compaction approach is capable to significantly reduce the application's recovery time. On workloads where the access to the most recent updated keys is stimulated, we reach up to a 50% compaction on the log file size with a 65% improvement in recovery time when compared to etcd's standard recovery protocol. In terms of performance, with the exception of a latency increase due to the implemented batching approach, no other relevant overhead was perceived.

Keywords: Distributed Systems. Fault Tolerance. Logging. Log Compaction. Storage Engines. State Recovery. Raft. Etcd.

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

CONTENTS

1 INTRODUCTION

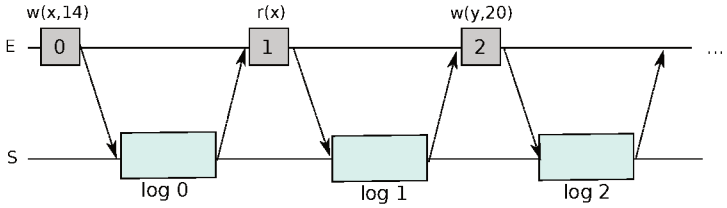
Logs are nested in the heart of many distributed applications. As an append-only durable sequence of records ordered by time, logs can store records that may have different meanings depending on the application. For instance, logging mechanisms play a central role in the development of database management systems and key-value stores (MOHAN et al., 1992; ZHANG et al., 2015), replication and coordination protocols (LAMPORT, 1978; JUNQUEIRA; REED; SERAFINI, 2011; ONGARO; OUSTERHOUT, 2014), and data integration middlewares (KREPS et al., 2011; LIU; IFTIKHAR; XIE, 2014; ASSUNÇÃO et al., 2015). Database management systems entrust logs the role to synchronize updates over various data structures and indexes, allowing a safe state recovery to replicas in case of failures or process migration (KREPS, 2014).

Logs can also be used as a consistency mechanism to order state updates to replicated services, as is the case of traditional replication protocols, such as *State Machine Replication* (SMR) (LAMPORT, 1978). This abstraction affirms that a set of identical and deterministic processes, beginning on the same initial state, will produce the same sequence of outputs if being fed with the same inputs in a total order. By evolving through a same sequence of commands, entry indexes in the log constitute a reliable timestamp for each replica, where the last executed entry attests its currently state compared to other replicas. Since this unique sequential log must be eventually perceived by all replicas, it is safe to announce that entry indexes can be safely utilized to identify missing commands during recovery. Many dependable systems with strict consistency requirements implement SMR (MARANDI et al., 2016), and utilize many variations of log-based approaches for state recovery (BESSANI et al., 2013; MENDIZABAL; DOTTI; PEDONE, 2017).

On a standard logging approach, every command is persisted to stable storage before any reply is sent to clients. These so called *pessimistic logging* schemes (ELNOZAHY et al., 2002) ensure a greater consistency level by logging commands before executing them. Especially when considering applications with strict consistency and durability requirements, this scheme must be implemented to guarantee safety in the presence of catastrophic failures, such as power outages or simultaneous failures. Although synchronous writing achieves a recovery objective of zero lost data, I/O costs may represent a major overhead on command execution (YAO et al., 2016).

Figure 1 illustrates a standard log-based protocol. For the sake of simplicity, we assume that each record stores a command from a simple key-value store data model. In this sense, update commands are represented by $w(k,v)$, and read commands by $r(k)$, where command w writes the value v over a variable given by the key k, and r reads the latest value associated with the key k. A logger process stores records in the log following the order they arrive, where each entry appended to the log is assigned to an unique and sequential number. The upper line E represents the execution of application's commands by an *executor task*, whereas S illustrates the *store task*. The received commands $w(x,14), r(x)$, and $w(y,20)$ are forwarded to the store

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



Source: The Author.

task, where they are written to the log in the order they arrive, given by the indexes $0, 1, ..$, and so forth. In this illustration, the store task performs synchronous writing system call. Thus, records *Log* 0, *Log* 1, *Log* 2, are persisted by successive writings to a log file.

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 recovering application's state. Thus, the entire sequence of logged commands must be replayed during recovery to allow a full and consistent state recovery. Specially for high throughput systems, this condition results in the processing of large log files during recovery, which 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; ZHANG et al., 2015) where any downtime period can result in significant 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 significant performance overhead (BESSANI et al., 2013; MENDIZABAL et al., 2014; MENDIZABAL; DOTTI; PEDONE, 2016; JUNIOR; AVILA, 2020).

1.1 OBJECTIVES

This dissertation presents an approach to speed up recovery and minimize log management costs by safely discarding commands considered unnecessary to a consistent state recovery. Workloads with a high read/write ratio and with intense command overwrite over a same space identifiers are common in today's large scale systems (ATIKOGLU et al., 2012), and could greatly benefit from such contribution. By discarding log entries, the proposed logging approach reduces the amount of recorded information necessary to recover a consistent state. The idea is that recovering from a compacted log leads to the same result as replaying an entire log sequence, but at a lower cost. A shorter command sequence affects both transferring and installation of logs, and can directly reduce the application's downtime period, thus increasing availability levels. In this sense, the following specific goals are also mapped:

- Develop a log compaction library, resulting in an experimental prototype coupled to a state-of-the-art open-source database;

- Evaluate the proposed logging approach costs and gains, understanding its impacts on the application's performance (*i.e.*, latency and throughput) and recovery time.

1.2 ORGANIZATION

The remainder of this manuscript is organized as follows. Chapter 2 presents some related contributions. Chapter 3 presents the methodology and implementation aspects of our approach. Chapter 4 presents an evaluation of our compaction algorithm and its application overhead against a key-value store prototype. Chapter 5 details the integration and implementation of our technique on etcd,[1] a popular commercial key-value store database, and presents results originated from its evaluation. Chapter 6 concludes this manuscript, and briefly describes possible future contributions from this work.

[1] https://etcd.io/

2 RELATED WORK

Log is a general concept in computer science, encapsulating different meanings under the same name. When referring to logs, some people would think of a sequence of errors or debugging messages on a text file following a certain string format and a timestamp (KREPS, 2014). This kind of log is intended for monitoring purposes, structured in a way humans are able to read, and is not the concept of a log approached on this manuscript. When mentioning logs, we are referring to a sequence of records ordered by time, intended to be read by computer systems. Different usages of logs fall on this category, such as: (i) logs used on database systems to provide durability, informing state during recovery to different replicas as is the case of a *Write-ahead log* (WAL); (ii) logs utilized on distributed consensus algorithms as a consistency mechanism to order decided values, and on related recovery protocols; and (iii) data integration middlewares. In this chapter, we describe some related contributions regarding log management optimizations under all three use cases. We divided these studies into two sections, grouping contributions by their goal rather than its application's use case.

2.1 LOG COMPACTION MECHANISMS

Log compaction strategies are utilized to manage logs sizes, allowing handling logs on limited storage environments. When referring to *compaction*, we allude to strategies that aim to reduce the number of records in a log file without compromising its state consistency. This technique is quite different from *compression*, a known technique in computer science to reduce storage footprint, utilizing a series of algorithms and exploring concepts of information theory such as entropy. One could combine both approaches: utilize compaction strategies to eliminate unnecessary log records and utilize compression algorithms to reduce storage size of logged information, but this runs out of scope for this study. Next, we demonstrate different compaction approaches and their corresponding trade-offs.

2.1.1 Kafka: a Distributed Messaging System for Log Processing

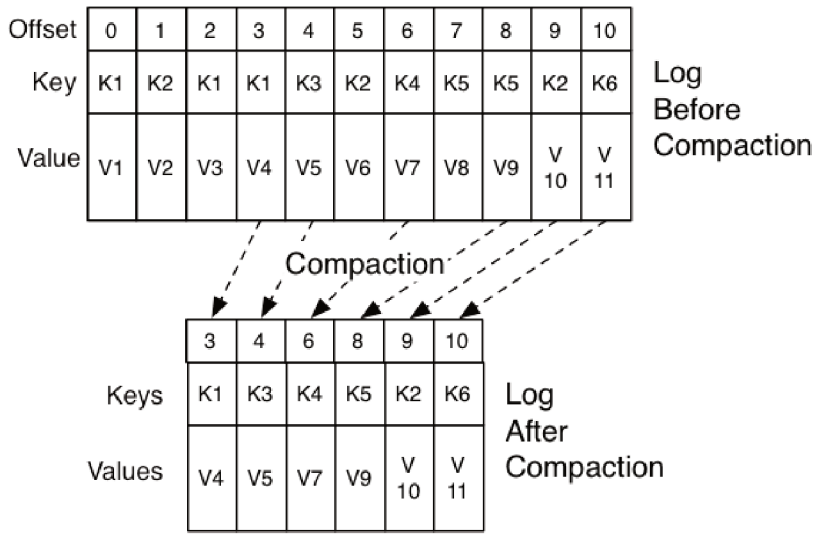
Apache Kafka[1] (KREPS et al., 2011; NARKHEDE; SHAPIRA; PALINO, 2017) is a popular event streaming platform that serves as a publish/subscriber middleware for real-time data processing and system integration. Kafka abstracts a so called *topic* structure, that acts like a persistent log storing a sequence of general events. The key idea behind Kafka's usage is to provide a scalable and reliable way to store information in the form of events. Kafka can be tuned for two main storage purposes: *short-term* and *long-term*. As a short-term storage middleware, a Kafka event topic acts like a conventional message queue, meant to provide an asynchronous integration between different systems. As a long-term storage engine, a topic

[1] Official technical documentation is also available at https://kafka.apache.org/documentation.html#compaction

acts like a database for a certain data scope, keeping events indefinitely and providing additional fault tolerance guarantees.

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 or "tail", that corresponds to already-compacted messages, storing only one value for each key, and the "dirty" portion or "head", containing 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 building the map, the cleaner starts from the oldest clean record checking the existence of each key against the map, deciding whether a record must be kept, in case it is not contained on the offset map, or later deleted. Figure 2 illustrates a Kafka topic log and its resulted state after a log compaction. As seen, intermediate values v1, v2, v3, v6 and v8 were removed in the compaction process, remaining only the latest values for each key in the resulted state.

Figure 2 – Apache Kafka's log compaction algorithm.



Source: Apache Kafka official documentation.[1]

By default, Kafka only starts compaction when half the topic contains dirty entries, since the compaction procedure compromises message throughput. Similarly, our approach discards intermediate state updates observed for each key. This enables a faster recovery procedure obtained by the compaction of retrieved state, a shorter topic in Kafka's case, and compacted state logs in ours. 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 recovery log.

2.1.2 Log-structured Merge-tree and RocksDB

Log-Structure Merge-tree (LSM) (O'NEIL et al., 1996) is a known technique widely used as storage layer of NoSQL databases (LUO; CAREY, 2020). The central idea is to execute out-of-place updates, where each new batch of updates is represented as a different node of the LSM tree. During execution, a LSM-based storage first buffers all writes in memory using an append-only operation and subsequently flushes them to disk in a single write. Once persistent, the contents of different nodes are merged by sequential I/O operations, following different compaction algorithms. The superior write performance against in-place storage engines is the main benefit of this approach, since all updates rely on cheap sequential I/O operations (DONG et al., 2017). Different approaches present optimizations regarding the LSM log compaction algorithm (DONG et al., 2017; PAN; YUE; XIONG, 2017; DAI et al., 2020).

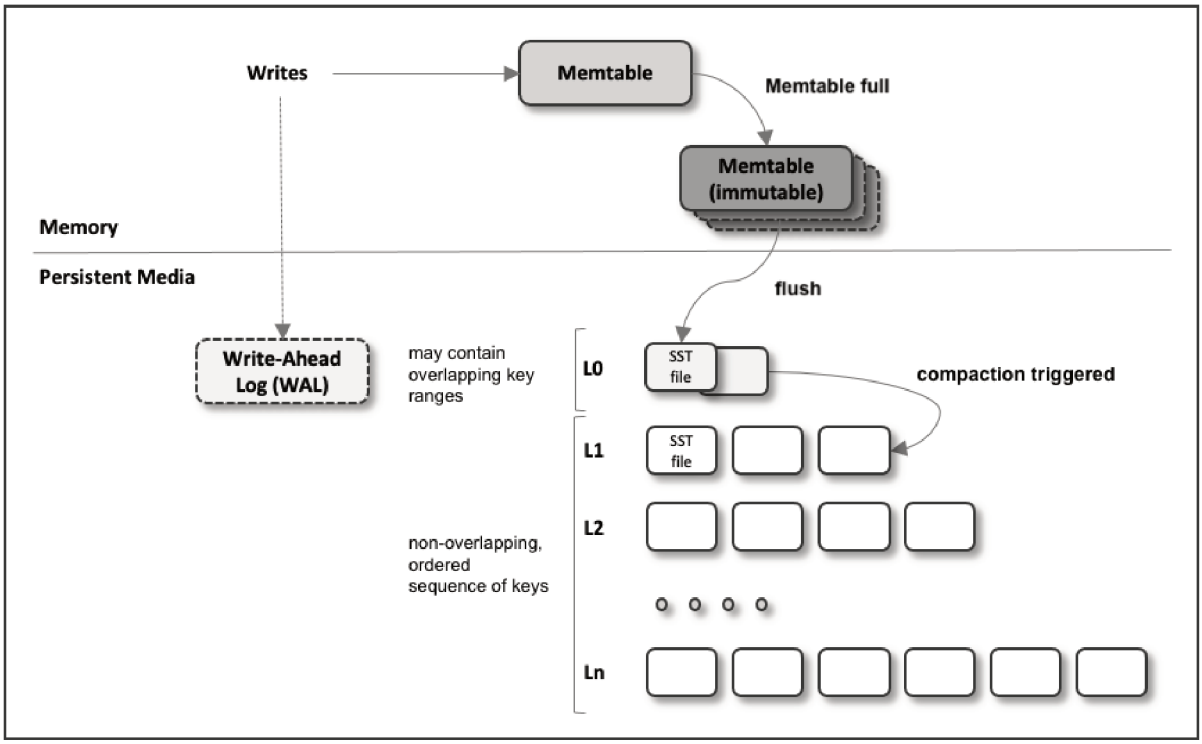
RocksDB[2] (ROCKSDB, 2013) is a persistent key-value store originally built at Facebook that implements LSM storage. It provides a highly adaptable, write-optimized, state store that can be used as a general storage back-end for any application. For instance, Apache Kafka utilizes RocksDB on stateful stream operations (CADONNA, 2021). One important characteristic of RocksDB is that it is not a distributed system. It does not provide high availability and does not have a distributed recovery scheme. Because of that, RocksDB is commonly used only as a storage engine for other databases, relying on additional layers to provide other common guarantees, such as fault tolerance and concurrency control.

RocksDB storage architecture is illustrated in Figure 3. The write operations against the current state are first applied over an in-memory structure called *memtable*, and generate a new record on a persistent WAL. Memtables only store state updates for given keys (*i.e.*, write and delete operations), and are considered *mutable* until a predefined number of operations are executed over it. Once a memtable is filled, it is considered as an *immutable*, read-only structure, and new operations are forwarded into a new memtable. As a new memtable is filled, the immutables structures are asynchronously persisted to secondary storage as so-called *Sorted Sequence Table* (SST) files. SST files are arranged in levels from the first, upper level, l_0 up to l_n. Each level can store up to a configured data size, and usually its capacity exponentially increases following its level order (*e.g.* l_0 storing 300MB, l_1 3GB, l_2 30GB and so on). SST files store the application's state and are only looked on read operations if the value for the requested key couldn't be find on an active memtable or in a superior cache layer. SST files store a bloom filter to allow a faster identification of a given key.

On this architecture, a crucial component to manage the database size, maintaining RocksDB performance, is the *compaction* algorithm. Compaction works by merging different SST files, checking for overlapping key ranges and removing intermediate writes and deletes. While compacting different SST files, represented by the sets F_1 and F_2, the result is the creation of a new file F_3 containing at most one value for each key and the remaining values of $F_1 \cup F_2$

[2] Official documentation is kept on the referenced blog and at a Github repository, accessible at https://github.com/facebook/rocksdb/wiki

Figure 3 – LSM storage architecture on RocksDB.



Source: Adapted from the RocksDB official documentation.[2]

in a sorted sequence, allowing fast read operations and merge procedures. The periodicity in which compaction is triggered may vary depending on the configured algorithm. On level compaction for instance, compaction is triggered once an upper level capacity is reached during the persistence of a new SST file. RocksDB allows different parameters to be configured in order to manipulate compaction and tune it depending on the application workload.

Although executing a similar approach to discard operations as our technique, both compaction strategies differ from one another in terms of their goal. The LSM architecture on RocksDB is utilized to represent an application's state. The compaction algorithm simply allows an efficient state representation, maintaining a manageable size to preserve the database's performance by running asynchronously at sparse periods. Differently, our approach shrinks logged information on-the-fly to reduce storage costs and allow a faster recovery procedure.

2.1.3 Comparison of Related Log Compaction Approaches

Table 1 depicts a comparison of different compaction strategies against our technique. In order to properly format and fit information in a limited space, some words were abbreviated. To differentiate strategies in terms of their compaction activation, we utilized the categories of *online* and *offline*, following a similar meaning of online/offline execution used in (KREPS et al., 2011). We consider that an *online* compaction is that immediately performed during the application's command execution, whereas an *offline* compaction is that eventually executed

during a compaction phase or during the application's execution in an asynchronous manner. Also, for each related work, we indicate its respective target application, the actual compaction technique, the recovery strategy, and its main positive and negative aspects.

2.2 EFFICIENT LOGGING AND RECOVERY MECHANISMS

Besides utilizing compaction strategies, state recovery can be hastened by different techniques exploiting engineering aspects on system design. This section presents related strategies that fall into this category, with the similar objective to provide a faster state recovery with minimal application's overhead.

2.2.1 Distributed Shared Logs

Different studies present alternatives to reduce logging overhead by the usage of specialized logger process. Corfu (BALAKRISHNAN et al., 2013) is a distributed and shared log, that among different features, allows client operations to run in parallel. During operation, Corfu's clients maintain a local projection map that stores record references to physical log positions divided into *pages*, and distributed across a cluster of logging nodes. Concurrent operations indexed to pages located in distinct nodes run in parallel, improving throughput and scalability.

In (XAVIER et al., 2020), the authors present an approach of decoupled logging for SMR applications. This technique involves the utilization of specialized logger process, participating as passive members[3] of the configured consensus protocol. These processes are capable to handle logging requests for multiple applications, and provide logging isolation for these different client states during recovery. The results indicate that shared logs can easily attend several clients, not compromise application's performance, and incur large monetary savings for cloud infrastructures. Both contributions are orthogonal and could be coupled to our approach, once they aim to optimize performance of I/O operations and logging management.

2.2.2 ARIES and *Adaptative Logging*

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 simply overwriting old values from their data-log structure. This strategy allows a parallel recovery procedure since only a single value is recorded for each key. A considerable drawback of this approach is the non-negligible overhead caused during normal execution. Being a verbose logging method, it excels in conventional persistent storage databases, where I/O

[3] A *passive participant* on a consensus protocol is one that is not allowed to propose values, only receiving decided ones. As for its implementation, these processes could act like *Learners* on a Paxos-based protocols, or as *non-Voters* on a Raft implementation.

costs are orders of magnitude higher than transactions' processing time. However, it yields significant overheads for in-memory datastores, where logging costs dominate the overall performance. Leveraging recovery benefits and low overhead on normal execution, in (YAO et al., 2016), the authors propose an adaptive approach that alternates between ARIES *data logging* and *command logging* (MALVIYA et al., 2014) methods during the execution. They adjust the percentage of data logging versus command logging 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 a variety of workloads, whereas command logging significantly reduces transactions' processing costs at the expense of a much slower recovery process. Our proposed approach bears some similarities with ARIES, such as the overwrite of outdated entries in the log, but implements a command logging strategy. While ARIES overwrites old *values*, our approach discards unnecessary *commands* for recovery. However, 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.

2.2.3 Taurus: Parallel Transaction Recovery

In Taurus (YU et al., 2016), the authors present an approach that relaxes the sequential logging and recovery restriction by tracking fine-grained dependencies among transactions. On uniform distributed workloads, where records are evenly accessed, Taurus can perform both logging and recovery in parallel. Taurus classifies operation dependencies into three categories to arrange a global dependency graph. RAW (*read-after-write*) transactions are compressed during recovery by exploiting the fact that the log is flushed to persistent storage sequentially. Therefore, if a record becomes persistent, all records before it in the same log file must be persistent as well. WAW (*write-after-write*) transactions are never relaxed and enforce a *must happen before* relation between two transactions during recovery. WAR (*write-after-read*) dependencies are always discarded; they do not constraint commit order because reads do not leave side effects in the system, not compromising state consistency when omitted. 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.

2.2.4 *Speedy Recovery* on P-SMR

In (MENDIZABAL; DOTTI; PEDONE, 2017), the authors present alternatives for state recovery in Parallel State Machine Replication (P-SMR). 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 application's semantics to map possible dependencies between commands, 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 in the state machine; 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 commands in 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.

2.2.5 Comparison of Related Logging and Recovery Approaches

In Table 2, we present a comparison of efficient log strategies against our proposed technique, which all intend to provide small application overhead or allow a faster recovery procedure. In order to properly format and fit information in a limited space, some words were abbreviated. To better differentiate these related contributions, for each of them we mention its target application, the implemented log optimizations, the recovery strategy, and its main positive and negative aspects.

2.3 CHECKPOINT-RESTORE PROTOCOLS

Checkpointing protocols are a complementary approach to represent state in durable applications. This technique is present on the design of different replication protocols (ONGARO; OUSTERHOUT, 2014), and, it is used by a variety of durable systems as a mechanism to manage log size (BESSANI et al., 2013). The checkpoint-restore recovery technique consists of asynchronously capturing the application's state during normal execution, generating a state representation for a given execution interval (ELNOZAHY et al., 2002). The state, named *checkpoint* or *snapshot*, is then persisted on stable storage to be later utilized on the recovery protocol. Once a checkpoint for a given interval $[i,n]$ is persisted across a majority of replicas,

the application log up to the n^{th} index can be safely truncated, since its command outputs are now satisfied on the saved state.

As for its recovery protocol, most 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.

Regarding performance, checkpoint protocols incur performance overheads observed during normal execution (BESSANI et al., 2013; MENDIZABAL; DOTTI; PEDONE, 2016). Since the checkpoint process executes concurrently against the application during normal execution, it requires synchronization primitives to ensure the persistence of a consistent snapshot. The synchronization costs directly impact the checkpointing overhead on the application's performance. For example, if implemented as a simple mutual exclusion for the application state, implementing a *Stop-and-Copy* approach, the application's throughput is degraded to zero during the checkpoint's execution.

Different contributions aim to reduce the checkpoint related costs. UpRight (CLEMENT et al., 2009) is a replication library that implements distinct checkpoint approaches apart from the traditional Stop-and-Copy. It uses the *Copy-on-Write* technique, which optimistically reads the application's state during execution without any synchronization, applying new write operations received in a different storage address. In (BESSANI et al., 2013), the authors present the *Sequential Checkpointing* approach, which explores the execution of checkpointing processes in a sequential and asynchronous manner among the group of service replicas. Their goal is to minimize the periods of service unavailability while saving the application's state.

2.4 DISCUSSION

In this chapter, we exposed different related contributions regarding log compaction and efficient logging mechanisms. Different compaction approaches, such as Kafka's compaction algorithm and RocksDB LSM, intend to reduce logged information by exploring the semantics of a read/write application. RocksDB for instance, is an active open-source project, with many call for contributions being made very recently.[4] On RocksDB LSM, our approach could be implemented to reduce the number of operations logged, or even identify and remove unnecessary operations before they are written into disk. This could impact the periodicity on which compaction is needed and, consequently, the database's performance. We could also couple our approach with its transaction WAL, providing a faster recovery of its in-memory mutable memtable.

[4] https://rocksdb.org/blog/2021/04/12/universal-improvements.html

Regarding log optimizations and recovery protocols mentioned on Section 2.2, most of them could be combined with our technique in one way or another. Distributed shared logs could be considered as the more feasible extensions for our technique, once they are inherently transparent to any application detail. Coupling our strategy on a distributed logger process could reduce an already diminished overhead for a reliable application, and also allow these shared loggers to scale even more.

Checkpointing protocols are also utilized to reduce storage footprint for logs on message passing systems (ELNOZAHY et al., 2002). A checkpoint protocol enables log truncation to keep logs in a sizable manner, allowing a faster recovery procedure due to skipping the sequential process of log entries during recovery. Although orthogonal to the utilized logging strategy, the utilization of a checkpoint protocol comes with the cost of actually taking the state snapshot, which incurs an overhead on the application's normal execution. Being transparent as it is, checkpoint/restore protocols, and all related optimizations, are complementary to our approach. On a recovery protocol, checkpointing could be easily coupled with our approach, allowing a log truncation of an already compacted log.

Table 1 – Related work with similar log compaction techniques.

Name	Application	Compaction	Activation	Periodicity	Recovery	Highlights	Caveats
(KREPS et al., 2011)	stream processing, compaction for long-term storage	discard intermediate updates with the same id	offline	size of a "dirty" portion	last value per key	allows long term storage	async. execution of cleaner threads can incur in overhead
(ROCKSDB, 2013)	non-distributed storage, write-intense loads	LSM algorithms: tiered, leveled and universal	offline	when the tree level exceeds a config. value	WAL for inmem state + LSM	extremely fast writes	read amplification and lack of temporal knowledge during recovery
Our technique	plug-and-play logging library	discard reads and intermediate updates within a batch	online	once a batch is filled	set of min. logs, one p/ batch	low overhead, batching, parallel I/O	latency increase on writes, proportional to batch size

Source: The Author.

Table 2 – Related contributions that provide efficient logging and recovery.

Name	Application	Optimization	Recovery	Highlights	Caveats
(BALAKRISHNAN et al., 2013)	general purpose shared logging service	shared log, batching and parallel IO	parallel recovery of different log shards	scalable shared log to attend several apps	
(XAVIER et al., 2020)	shared logging service for SMR applications	shared log	concurrent recovery for different application clients	scalable shared log, minimizes costs on IaaS	addition of loggers as passive members on consensus
(MOHAN et al., 1992)	ACID storage database	only the last value per key is logged	parallel recov. for different WAL records	extremely fast recovery, parallelism for all records	overhead during execution caused by data logging
(YU et al., 2016)	parallel recovery protocol for log-based applications	batching	parallel recov. for unconflicted entries, parallelism logging/recov.	unconflicted entries can be logged on diff. devices in arbitrary orders	conflict detection is done through a DAG, results in an overhead proportional to the structure size
(MENDIZABAL; DOTTI; PEDONE, 2017)	parallel recovery protocol for P-SMR	batching	cp. + parallel recovery of unconflicted entries, parallelism on logging/recovery	dependencies within batches are evaluated through bloom filter comparisons	bloom filters can inccur in false-positives dependencies within batches
Our technique	plug-and-play logging library	batching, parallel I/O	set of min. logs, seq. recovery for files, paral. execution of inner commands	low overhead, batching, parallel I/O	latency increase on writes, proportional to batch size

Source: The Author.

3 PROPOSED LOGGING APPROACH

Log recovery is a major design concern on dependable systems, and an efficient compaction approach directly impacts application's availability. Different compaction strategies exist in the literature, such as Apache Kafka's and RocksDB LSM algorithms. In our approach, we implement a different compaction technique, which exploits the application's command semantics and executes a compaction procedure in an *online* manner. Although shrinking the log at run-time might seem expensive at first glance, the proposed logging implements some optimizations, minimizing the following costs:

- **Synchronization costs:** By enabling concurrency between command execution, compaction, and persistence;
- **I/O costs:** By exploring batching on the compaction and persistence of commands, and utilizing parallel I/O when multiple storage devices are available.

This chapter presents a more in-depth reference of our technique, covering the fundamental and implementation aspects of these optimizations.

3.1 SYSTEM MODEL

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 (OLIVEIRA; GUERRAOUI; SCHIPER, 1997; AGUILERA; CHEN; TOUEG, 2000) 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 code 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.

3.2 FUNDAMENTALS

The key idea for shrinking the log is to consciously avoid the logging of commands considered unnecessary to reach a consistent state. To demonstrate the potential of safely discarding unnecessary log entries, we adopted a key-value store data model advocating its representativeness for a variety of applications. We assume the data model of a key-value storage application executing single-variable read and write commands over an address space, where a read $r(k)$ returns the value associated with the key k and a write $w(k, v)$ updates the variable given by the key k with the value v. In this case, read operations and writes to values that are subsequently overwritten are examples of unnecessary log entries, where only the last write command needs to be logged to reach a consistent state.

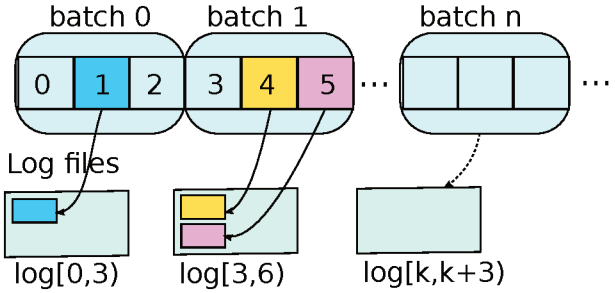
When recovering from a compacted state, the omission of unnecessary commands transferred to recovering replicas should not modify the state obtained from recovery. For logging purposes, incoming commands are seen as batches, and the compaction is performed per batch. When executed, the compaction procedure discards all unnecessary commands in the batch, so only the necessary ones are recorded to the log. The motivation behind the usage of batches is to allow the detection of overwrites on a limited interval of command indexes.

Figure 4 illustrates the compaction and persistence of commands into the log. The sequence of incoming commands is evaluated in the form of successive batches, given by *batch* 0, *batch* 1, and *batch n*. Each batch is compacted so that only the necessary commands are kept. Arrows indicate 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 t commands, where t is the batch size. As illustrated 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 for recovery are present in the batch n. Notice that the elimination of write operations to outdated values is possible only for successive writes in the same batch. As an example, let us assume that a client c_1 executes $w(x, 10)$ and a client c_2 the $w(x, 20)$ command, where both are associated with indexes 0 and 1 in the figure, respectively. In this case, since both executed within the same batch, only c_2's command is recorded in the log. A different case would occur if c_1 and c_2 were associated with indexes 1 and 4, where although c_1 and c_2 updated the same variable, both would be appended to the log files.

3.3 LOG COMPACTION

The compaction procedure delimits the commands in a virtual batch by establishing successive intervals every *batch size* commands. That means the system does not need to pack commands in a batch explicitly. After batch size commands have been compacted, the logging procedure stores the resulting commands in the log and starts reducing the next batch size commands.

Figure 4 – Compaction of command batches and split of the log.



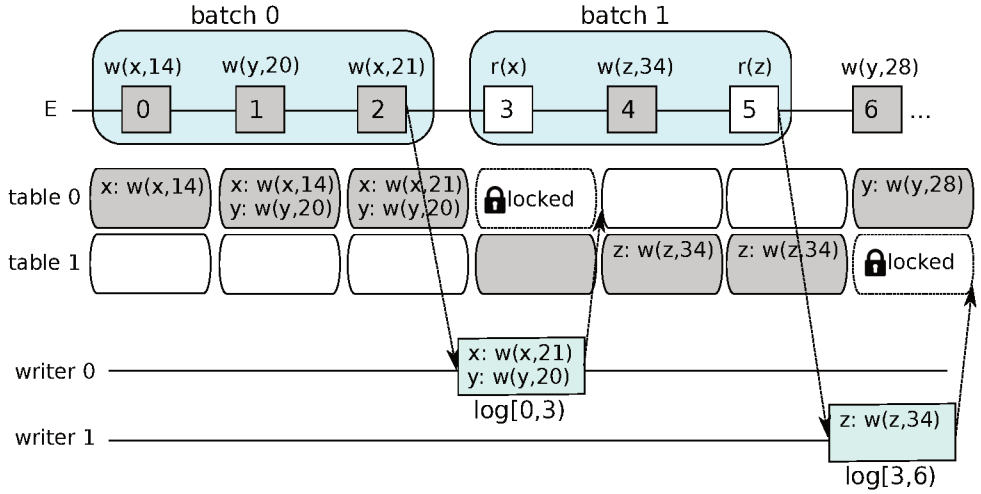
Source: The Author.

During the compaction of a batch, a hash table is used to keep information of which keys are updated by the commands. Each record on the hash table store the serialized command data indexed to its corresponding key, providing $O(1)$ costs for simple insertions. A write operation issued to an already populated key overwrites the current command associated with that key. The proposed strategy separates the compaction procedure from the logging in stable storage, and allows the usage of multiple tables to explore concurrency between compaction and persistence of commands. Thus, while a set of commands resulting from a compaction is being persisted, other batches can be reduced in parallel. When multiple storage devices are available, persistent writings can also occur in parallel, boosting I/O throughput.

The concurrent execution between log compaction and log persistence adopts multiple compaction tables and log writers. When the compaction procedure is fulfilling one table, other tables' content can be flushed to the persistent storage by log writers. Tables are accessed in mutual exclusion by the compaction procedure and log writers. A mutex enforces the synchronization over a cursor variable. The cursor indicates which table is being used by the compaction procedure. Besides, each table has a lock to ensure that only one log writer is persisting the table data to the log. If the compaction procedure tries to acquire the lock to this table to perform a batch compaction, it will wait until the table's contents are fully flushed into stable storage. Thus, the synchronization costs are perceived only on two scenarios: (i) when accessing the value of the cursor variable, that indicates the table index in which the operation must be logged; (ii) when the number of available tables is exhausted, so that the application must wait until the contents of the current table 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 compaction procedure acquires the mutex, signals a log writer to record the table 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 persists all serialized values from the table to the stable storage with its corresponding $[first, last]$ command index interval as metadata. Finally, it resets the table t_n, unlocks it, and signals the application process to send client replies to all the batched commands, even to the safely discard ones.

Figure 5 – Concurrent execution between log compaction and log persistence.



Source: The Author.

Figure 5 depicts an execution scenario with two hash tables, 2 log writers, and a batch size of 3 commands. As shown, the compaction procedure only modifies the current table for each command execution if the processed command is an update. If the 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 table cursor for the next table (*i.e.*, *table* 1). When signaled, log writer 0 immediately starts the persistence of *table* 0, interpreting the table state and aggregating its latest commands into a single log file. 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 completely persisted to the secondary storage, *table* 0 is reset, and it is now free to receive commands from another batch. The executor task E is notified of this reset, and can now send client replies related to the commands 0, 1 and 2.

In parallel to the *table* 0 logging, the compaction procedure executes commands from the next batch (*i.e.*, *batch* 1) by using *table* 1 to the commands that must be recorded in the log. 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 table's lock.

It is worth mentioning that 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 compaction procedure to fulfill 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 its job, the compaction procedure will 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.

3.4 RECOVERY PROTOCOL

The recovery protocol allows a replica to restore a consistent state and catch up with other replicas in the system. Recovering a failed replica requires retrieving the commands the replica missed while it was down, which can be obtained from other replicas' logs.

Our recovery protocol is based on the interpretation of a series of log files periodically generated during normal execution. The first two lines of a log file indicate the *first* and *last* indexes of commands stored in the file. These indexes, given by i and j, define the sequence containing only the necessary commands in the interval $[i,n]$. The following line indicates the number of commands written on the log, which can be less than the $n-i$ due to the removal of certain operations. The log is followed by a sequence of serialized commands, where the size of each command is binary encoded before its raw byte stream. An *End-of-Log* (EOL) flag is appended at the end, indicating the correct termination of the concurrent logging procedure.

At initialization, a recovering replica checks the last instance k stored in its local log, if any, and sends a request to other replicas asking for more up-to-date logs. A request informing index k is sent to other replicas, and those ahead in the execution reply with their highest index number h. 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. By returning a log with an upper index h, the replica not only implies that the informed log maps the application's state until the h^{th} command, but also that the informed log is *compacted* until the h^{th} command, disposed of reads and *later-overwritten* writes. The recovering replica then requests the log represented by all files within the $[k,h]$ 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.

After retrieving the log, we allow the configuration of one of two different strategies to process log files and execute its commands. These strategies are called *Naive* and *Descending*, and defined as:

- **Naive:** Process the retrieved log files on **ascending order**, ensuring that the commands' execution follows the same order of persisted batch intervals.
- **Descending:** Process the retrieved log files on **descending order** utilizing an auxiliary hash map data structure to ensure that only the last command per key is processed for the given log. This strategies delivers an optimal minimal set of independent commands.

A high-level pseudo-code for Naive and Descending strategies are shown in Algorithms 1 and 2, respectively. Considering f as the number of log files being processed, Algorithm 1 has a time complexity of $O(f\ lg\ f)$, being $O(f\ lg\ f)$ the analyzed complexity for the

Algorithm 1 Naive log processing strategy

```
1: procedure NAIVE([ ] files)
2:     SORTASCENDING(files)
3:     cmds ← NEWARRAY()
4:     for f in files do
5:         cmds.APPEND(LOADCOMMANDSFROMFILE(f))
6:     return cmds
```

Algorithm 2 Descending log processing strategy

```
1: procedure DESCENDING([ ] files)
2:     SORTDESCENDING(files)
3:     cmds ← NEWARRAY()
4:     for f in files do
5:         cmds.APPEND(LOADCOMMANDSFROMFILE(f))
6:
7:     table ← NEWHASHTABLE()
8:     for c in cmds do
9:         if not table.CONTAINS(c.Key) then
10:            table[c.Key] ← c
11:    return table.VALUES()
```

utilized `SortAscending` procedure. Besides the space complexity of `SortAscending` defined as $O(f)$, and the $O(n)$ to store n commands in memory, no extra space is consumed for Naive. On Algorithm 2, a time complexity of $O(f\ lg\ f)+O(n)$ is perceived for the Descending strategy due to the process of each individual command loaded from the log files (line 8), considering f as the number of log files and n as the number of loaded commands in total. The algorithm presents a space complexity of $O(f)+O(n)$, since at most n keys will be stored in the auxiliary hash table when processing the n commands loaded into memory.

If, after recovery, a replica receives any command with an index w, where $w \leq h$, it only has to ignore and not execute it, since its effects would have already been applied to its state. If $w > h+1$, 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 w, and the gap of commands would be fulfilled.

3.5 OVERALL REMARKS

The presented logging strategy relies on the application's command semantics to safely discard unnecessary entries from logs, diminishing storage costs, log transferring, and execution time from a recovering replica. To mitigate performance overheads, the technique explores a concurrent execution between the compaction and persistence tasks, and offers cheap I/O operations by writing batches of commands at once, that can also be parallelized if multiple I/O devices are available. In terms of recovery, one could choose between two different log process-

ing strategies. If executing over a memory limited environment, or the run-time compaction is already enough for the submitted workload, the Naive strategy can offer a lower recovery time due to simply recovering log files in ascending order in a lower time complexity. If an optimal compaction is needed, the Descending approach can further eliminate unnecessary commands that were not previously identified during run-time due to discarding overwritten commands between different batches.

Next, we evaluate the proposed logging approach in detail on Chapter 4, and analyze its performance when coupled to a commercial key-value store database in Chapter 5.

4 LOG COMPACTION EVALUATION

This chapter describes a first evaluation analyzing initial aspects of our approach, considering the baseline model of a standard logging scheme. Part of the results depicted in this chapter were published in (XAVIER et al., 2021).

We seek to answer two main questions with this evaluation:

- **Does our protocol allow a faster recovery procedure?** We evaluate the generated log files on different batch size 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.
- **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 under different workloads.

To evaluate the impacts of our approach, we designed the implementation of our log strategy in the form of an independent software library,[1] imported and utilized by a key-value store prototype[2] written in Go. Read and write operations are illustrated 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. Each incoming command c is issued to a $log(c)$ procedure, where its managed by the logging library with different configurations, such as number of concurrent tables and log interval. The prototype is configurable to execute the standard or the proposed logging approach. Commands are logged in batches, and replies to the command batches are sent to clients only after logging and execution. All persisted commands are serialized with protocol buffers.

Our evaluation aims to avoid costs and performance bottlenecks unrelated to the logging itself. Towards this end, 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. The analysis of our technique embedded in a commercial key-value store database appears in Chapter 5.

4.1 WORKLOADS AND CONFIGURATION

At initialization, the load generation produces read and write commands according to an input file following a specific workload pattern. We utilized a subset of the standard Yahoo!

[1] https://github.com/Lz-Gustavo/beemport
[2] https://github.com/Lz-Gustavo/beexecutor

Cloud Serving Benchmark (YCSB) (COOPER et al., 2010) workloads with two variations of YCSB-A, named YCSB-AW and YCSB-AWL.

The standard workloads YCSB-A, YCSB-B, and YCSB-C, vary in the percentage of read and write commands, but all of them follow an 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 an unfavorable 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. The subsequent access to recent keys, and a high read/write ratio, is emphasized on (ATIKOGLU et al., 2012).

In our load generation, inserts and updates are mapped into write operations. 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;
- YCSB-AW: 100% of writes, following a uniform distribution;
- YCSB-AWL: 100% of writes, following a latest distribution.

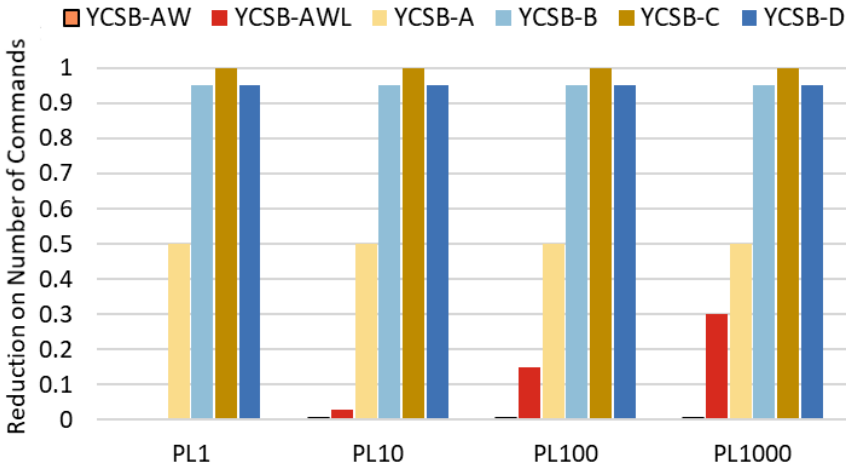
All workloads consider one million (10^6) distinct keys during load distribution. Generated log analyses were conducted with one million (10^6) commands workloads, whereas throughput and latency impact evaluations were executed with ten million (10^7) inputs to allow a longer execution. A ten minutes execution timeout is set for the throughput and latency experiment scenarios, which means that each execution for each interval configuration and workload executes up to a maximum of ten minutes processing the 10^7 input before finishing.

Experiments were executed on the *Emulab Utah* research cluster (WHITE et al., 2002), utilizing a Dell Poweredge R430 node equipped with two 2.4 GHz, 64-bit, 20MB cache, 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.

4.2 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 6 – Number of commands reduction on generated recovery logs with 10^6 commands, normalized to SL values.

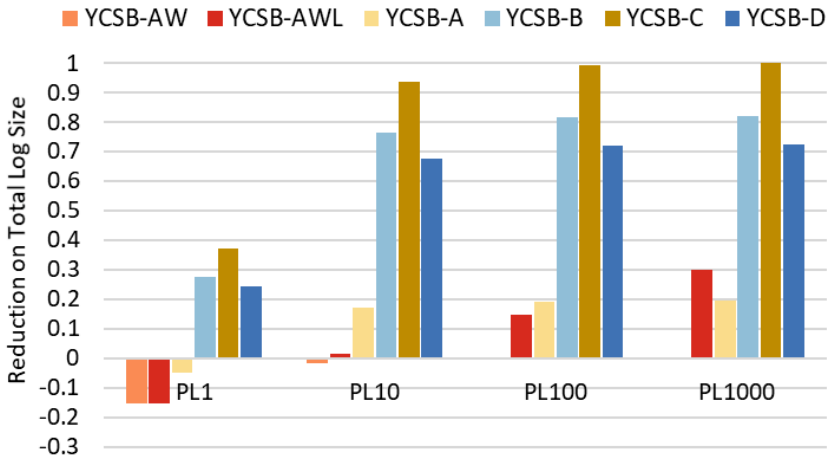


Source: The Author.

Figure 6 shows the reduction achieved on the number of commands for each combination of workload and batch size configuration, 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, which is explained by the write-only workload with a high number of distinct keys evenly distributed by the load generator. For YCSB-AWL, the latest distribution shows an interesting scenario by stimulation the removal of write operations over recent out-dated 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 PL1000. 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 7 shows the total log size reductions compared to SL. On the standard approach, the total log size observed were: 130.64 MB for YCSB-AW and AWL; 80.99 MB for YCSB-A; 36.42 MB for YCSB-B; 31.46 MB for YCSB-C; and 41.26 MB for YCSB-D. On the PL-1 scenarios, where a logging procedure is triggered after each command, our approach presents a penalty of ≈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 (*i.e.*, *first* and *last* indexes) 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 significant 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-

Figure 7 – Total size reduction on generated recovery logs with 10^6 commands, normalized to SL values.



Source: The Author.

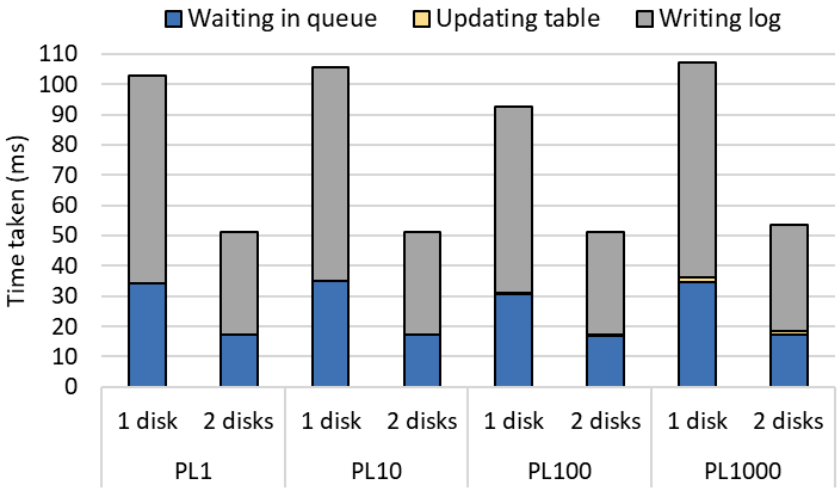
A maintains a ≈ 20% total size reduction. YCSB-AWL shows incremental benefits on larger batches, with a 30% decrease in log sizes for PL1000. Substantial differences can be seen for YCSB-C, B, and D, where a nearly 100% decrease is shown for the former and ≈ 80% for the other two for batch sizes of 1000 commands.

4.3 LATENCY AND THROUGHPUT ASSESSMENT

Figure 8 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 ≈ 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.

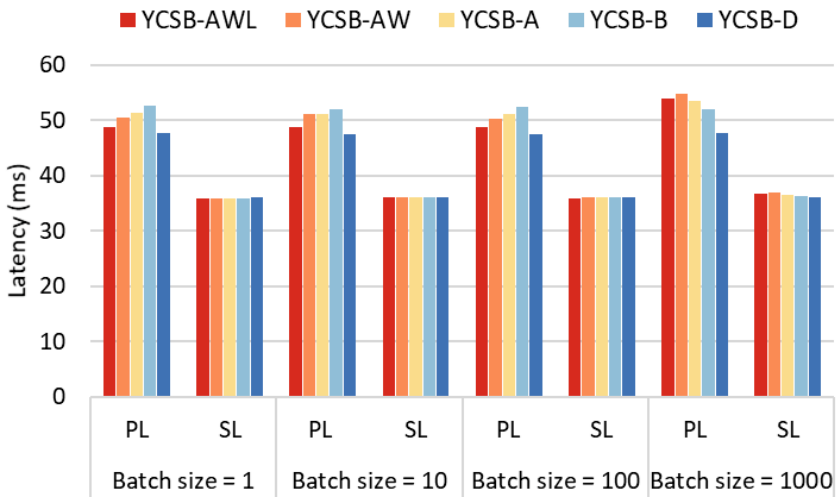
In Figure 9, we analyze the average latency measured for log persistence, comparing PL with its two disks configuration against SL on different workloads and batch sizes. At first glance, both strategies manifest low variations for latency values as batch size increases. This result is explained by the 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 stimulated workload since this strategy logs every command independently of its operation or accessed key. That is different for PL because of

Figure 8 – Latency breakdown for the proposed logging analyzed for YCSB-A.



Source: The Author.

Figure 9 – Average latency of SL and PL strategies for each workload, considering different batch sizes.

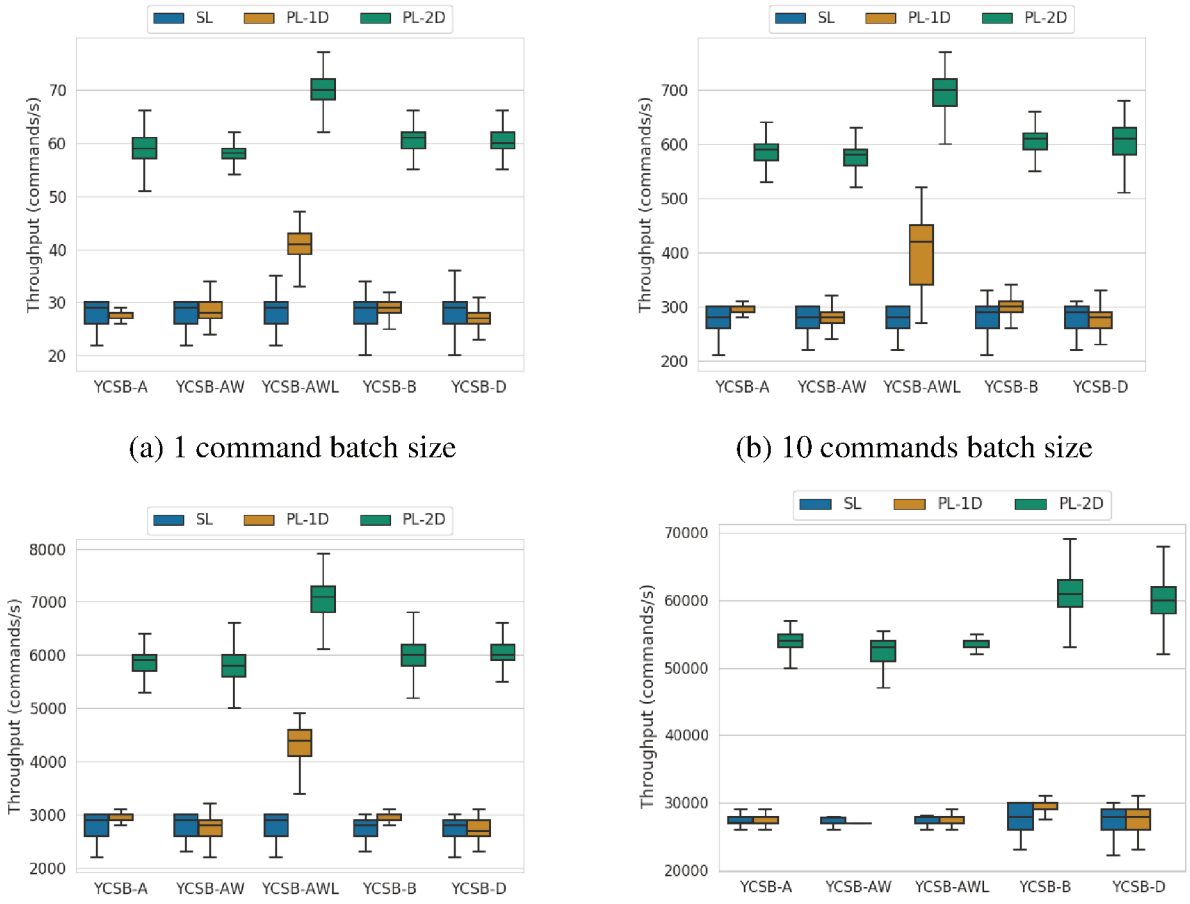


Source: The Author.

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.

Figure 10 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 10c. This effect is a consequence of eliminating subsequent

Figure 10 – Throughput analysis for different workloads and batch sizes with SL, PL-1D and PL-2D configurations.



(a) 1 command batch size

(b) 10 commands batch size

(c) 100 commands batch size

(d) 1000 commands batch size

Source: The Author.

write operations, where only the latest update of a particular key is kept for each batch of commands. The removal of writes incurs a 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 the I/O bottleneck's mitigation perceived during log flush to stable storage. PL-2D median values show a $\approx 100\%$ increase on throughput for most workloads on 1, 10, and 100 batch size configurations.

5 LOG COMPACTION IN A COMMERCIAL KEY-VALUE STORE

In this chapter, we extend the previous evaluation conducted on Chapter 4 to utilize a commercial logging strategy as a baseline comparison. This decision aims to demonstrate and verify the behavior of our technique considering a more realistic and competitive standard logging procedure (SL) than the first prototype. Some optimizations implemented on our approach, such as batching, and parallel I/O, are typically implemented by the logging procedures from commercial database systems. In this sense, instead of trying to mimic more realistic baselines on the prototype, we opted to follow the opposite path: start from a fork of an open-source production-grade database and implement our logging approach as a different configuration. By following this strategy and choosing a worthy baseline for our approach, we improve our analysis by two aspects: (i) by presenting evaluation results that are, under some circumstances, comparable to previous benchmarks results of the same database in the literature; and (ii) by minimizing possible biases that could be questioned regarding the implementation of realistic models on a key-store prototype.

After studying the repositories and documentations of different options, we opted to utilize etcd[1] as the commercial open source database on our evaluations. Etcd is a strongly consistent key-value storage, backed by a state machine using Raft consensus algorithm (ONGARO; OUSTERHOUT, 2014), and it is utilized by major projects in the industry, such as Kubernetes, CoreDNS, and Uber's M3. On Kubernetes, etcd is utilized to store all the cluster configuration and membership, because of its strong consistency guarantees and high availability. Besides its relevance, we opted for etcd due to its extend documentation and easy-to-follow Github repository.

Due to its relatively simple NoSQL key-value data model, etcd exposes a minimal API supporting 5 kinds of requests:

1. `put`, a single-variable write operation;
2. `range`, a read operation which can read from a single or a range of keys;
3. `deleteRange`, a delete operation which can delete a single or a range of keys;
4. `txn`, a request which can encapsulate different `put`, `range` and `deleteRange` operations to be processed atomically; and
5. `watch`, a request that signs up the caller to be later notified on every modifications on it's informed key.

Considering etcd's operation API, we revisit our previous definition of *unnecessary commands* to be now constituted by all `range` operations and the `put` operations over the same keys that are overwritten within a batch.

[1] https://etcd.io/

5.1 ETCD'S INTERNALS AND WRITE-AHEAD LOG

By utilizing its own implementation of the Raft algorithm, etcd design is strongly coupled with the protocol internals. For instance, on Raft proposed commands are always forwarded to leaders, where they are immediately appended as new entries on its log. From time to time, the leader replicates its log to other members. Once an entry is appended to a majority of members logs, that entry is considered committed, and it is now assured to be presented on every future leaders log. This kind of distinction between standard entries and committed ones is known by etcd, and is utilized to explore the concurrent execution between the persistence of new entries, *i.e.*, before they are considered committed, and the execution of already committed ones.

Algorithm 3 illustrates this procedure and the delivery of Raft states through a `Start()` function, shown in pseudo code. On the function new states are always received from the protocol abstraction through a channel (line 2). Since etcd is written in Go, it heavily utilizes a channel abstraction to implement any kind of communication between concurrent coroutines. For each received state, the `Start()` function sends its committed entries to `applyChannel` (line 3), where it will be handled by a different function concurrently. After those commands are received from `applyChannel`, but not after they are processed, `Start()` continues execution by persisting new entries on the recovery WAL (line 4) and by calling `Advance()`, which simply tells the protocol that the received state was fully executed, and a new state can now be received.

Algorithm 3 Etcd Raft handling function

```
1: procedure START
2:     while receive from raftChannel as state do
3:         send state.CommittedEntries to applyChannel
4:         SAVEONWAL(state.Entries)
5:         state.ADVANCE()
```

Being called at etcd's initialization, `Start()` is continuously executed concurrently with all the other application's procedures, and continues throughout all the process lifetime. Algorithm 4 illustrates some of these concurrently executed processes, and shows how these interact with the procedure `Start()` from Algorithm 3. As can be seen, a `Put()` procedure is depicted, and it represents a function that handles `put` requests received from the API. As the first step on `Put()`, it creates a Raft proposal request from the received request data and proposes it to the underlying protocol on the following line (lines 2-3). This proposal results in the delivery of a new state to the `raftChannel`, serving as an input to `Start()` (Algorithm 3, line 2). After proposal, it invokes the `Wait()` function for its corresponding proposal request on line 4, which blocks this procedure until that protocol request is fulfilled, *i.e.*, is considered committed and executed for a majority of nodes; or is *aborted*, if rejected by the leader or not processed within a timeout. After processed, the procedure continues to execute by assigning `Wait()`'s returned value to a `resp` variable, which is then forwarded as a client response on the

Algorithm 4 Etcd request handling and command processing functions

```
 1: procedure PUT(Request req)
 2:     r ← CREATERAFTREQUEST(CMDTYPE_PUT, req.Data)
 3:     PROPOSERAFTREQUEST(r)
 4:     resp ← WAIT(r.ID)
 5:     SENDCLIENTRESPONSE(resp)
 6:
 7: procedure APPLY
 8:     while receive from applyChannel as ap do
 9:         resp ← EXECUTECOMMAND(ap.Command)
10:         NOTIFY(ap.ID, resp)
```

following line. With a similar behavior to `Start()`, `Apply()` continuously executes by reading new apply requests through an `applyChannel` (line 8). For each received requests, it executes its underlying command and sends its result through the `Notify()` procedure (line 10), which also unblocks its correlated `Wait()` (line 4) with the same identifier.

As an important aspect for our approach, etcd's Raft protocol implements batching of commands during delivery. In practical terms, it basically delivers a set of *Entries* and *CommitedEntries* for each `state` received on `Start()`, as shown in Algorithm 3 (line 3). This batching approach is intended to hasten consensus rounds and achieve a higher throughput.

Durability is ensured in etcd by logging Raft's committed entries into a *Write-Ahead Log* (WAL) structure, before its corresponding commands are executed. To clarify terms, it is important to mention that WALs are different from the Raft log. WALs store only committed entries, are saved in persistent storage, and are solely utilized for recovery purposes; whereas Raft logs can momentarily contain non-committed entries, are stored in memory, and are utilized by the underlying protocol to coordinate state and ensure consistency among participants. For the sake of simplicity, when describing etcd, we will only refer to WALs when mentioning logs, log files, or any similar term, except when its explicitly mentioned other kinds of logs.

During execution, a WAL file is kept for every single replica within an etcd cluster, where each starts with a pre-allocated size of 64 MB. As for its format, WALs store for each new record: (i) a length field, which is a 64-bit structure holding the length of the actual record in its lower 56 bits, and its physical padding in the first three bits of the most significant byte; and (ii) a binary representation of the WAL record, which is the Protocol Buffer serialization of a structure holding the command type, its data, and a *cyclic redundancy check* (CRC) for error detection. Each record is 8-byte aligned so that the length field is never torn. To ensure state consistency, WALs are necessary written by synchronous I/O calls. In terms of storage, by default etcd can store only 2 GB of data, configured up to 8 GB.

As an optimization, etcd does not log `range` commands on its WAL, since they do not incur a state modification and are unnecessary to achieve a consistency state during recovery. We implement the same optimization on our technique by not logging read operations, with the addition of detecting write overwriting within a batch, which is not implemented by etcd.

5.2 ETCD'S RECOVERY PROTOCOL

The standard recovery procedure is triggered during etcd's initialization in case it detects stored WAL files or snapshots within a configured directory. Snapshot is a synonym for a *checkpoint*, referring to Checkpoint-restore protocols as discussed on Section 2.3. If a snapshot exists, the recovering node first identifies the highest entry index i contained in the newest snapshot. The replica then starts retrieving entries stored in WAL files within its disk, informing the i index to the WAL's read procedure, which indicates that only entries starting at index $i+1$ can be loaded. The WAL could have been already truncated, but its not ensured since the truncation is executed by a concurrent routine, and not sequentially after a snapshot is taken. If none snapshot was perceived, all entries are retrieved regardless of their indexes. After this procedure, an in-memory list of entries sorted by their indexes is returned.

From a sorted list of entries, etcd starts the initialization of its Raft node, loading the retrieved entries into the protocol's log. By doing this, the recovering replica knows three important informations: (i) its current *commit index*, representing the index of the latest entry present on its log; (ii) its current *term* (*i.e.*, the protocol state), also interpreted by the latest entry; and (iii) its *apply index*, representing the index of the latest command executed, which during recovery is always zero. If starting in an existing Raft cluster, with more than one node, a recovering replica r_n will necessarily inform these three attributes to the current leader r_l, which will then decide to: (i) fill the gap on r_n's log, if it detects that its current *commit index* and *term* are greater; or (ii) do nothing, if detects that r_n's *commit index* is updated to its own. A recovering replica to an existing cluster will never have a more updated log than the current leader (ONGARO; OUSTERHOUT, 2014). If starting in a single node cluster, the recovering replica will immediately consider its retrieved state, acquired from snapshot plus WAL, as the most updated one. Then it declares itself the leader and continues the recovery protocol.

The last step remaining to start processing new inputs is to re-create the application state by applying the latest snapshot and executing the remaining entries' commands. Once the recovering replica joins the consensus protocol, it eventually receives all the commands between its *commit index* and *apply index* by the exact same way as it receives regular proposed entries: through its `raftChannel` from `Start()` procedure (Algorithm 3). Both Raft decisions and `Start()` procedures are executed concurrently within the same process. One important aspect to mention is that instead of sending all of the retrieved entries at once, Raft will batch the retrieved entries while sending them to the running application through the `raftChannel`, as it does to regular proposed commands.

5.3 CUSTOMIZING ETCD

In order to evaluate our technique's performance on etcd, we had to implement some modifications on the code base. At first glance, we identified the existence of a `Storage` interface, initially implemented by the WAL abstraction, but easily interchangeable to a different

structure at run time. Following this path, we then defined a different structure implementing the `Storage` interface, whereas its `Save()` method would simply act as a wrapper to our library's `Log()` procedure. To easily switch between this new implementation and the standard one during experimentation, we defined an environment variable `ETCD_LOG_CONFIG` that was parsed during initialization to identify which structure should be initialized and assigned to the interface. Unfortunately, following this path did not end up on a working prototype. By implementing only the interface invoked during command persistence, we were not able to assure two main aspects of our approach: (i) hold commands' execution until a batch is filled, while still allowing new commands to be delivered; and (ii) hold client responses until batches are fully persisted. Nonetheless, it served as a practical exercise to understand the source code, and gave us a better understanding of the database internals.

We later opted to modify etcd's `Start()` function behavior in order to have a better control over command delivery by the protocol. In this sense, we implemented three different variations of `Start()` functions: (i) a regular `Start()` with the original etcd implementation; (ii) a batch variation, that served to compare etcd's approach with the same batching procedure as ours; and (iii) our technique's variation, which not only batches commands but also implements our compaction strategy during logging. The decision to which function to invoke is done by parsing the same `ETCD_LOG_CONFIG` environment variable.

Implementing batching of commands on top of etcd was not a trivial task. When delivering new entries to the application's routine through the `raftChannel`, Raft already batches commands to maintain latency levels on scenarios of higher throughput. Because of that, we had to come up with an implementation that was sufficiently generic to the delivery of any quantity of commands by Raft at a time. The implemented batching procedure on `StartBatch()` is shown in Algorithm 5. This procedures utilizes two new abstractions: (i) a *batch*, used to retain *Entries* and *CommittedEntries* received from the protocol, delaying logging and execution of commands; and (ii) a *timer*, utilized to specify a maximum retention timeout on batching, avoiding command execution to be hold indefinitely on scenarios without a continuous client throughput.

An important aspect to be mentioned for this implementation is that although we configure an intended batch size for logging purposes, their actual persisted size may differ. In scenarios with a continuous delivery of new states from the *raftChannel* within a `BATCH_TIMEOUT`, the size of generated logs would be of at least `BATCH_SIZE` commands (Algorithm 5, lines 27-28). However, if the `BATCH_SIZE` value is too high, or a sparse delivery of new states is perceived and a batch is not filled until the timer expires, logs with less than `BATCH_SIZE` could be generated (Algorithm 5, lines 8-13).

Figure 11 depicts an analysis over the batch sizes of generated log files on our infrastructure. The horizontal axis represents the configured values for the `BATCH_SIZE` parameter on our proposed logging approach (PL), while the vertical axis depicts boxplots of the actual sizes of persisted log files. In Figure 11a, we analyzed the batch sizes of generated log files when etcd was executed with the same `BATCH_SIZE` values of 1, 10, 100, and 1000 as studied on the

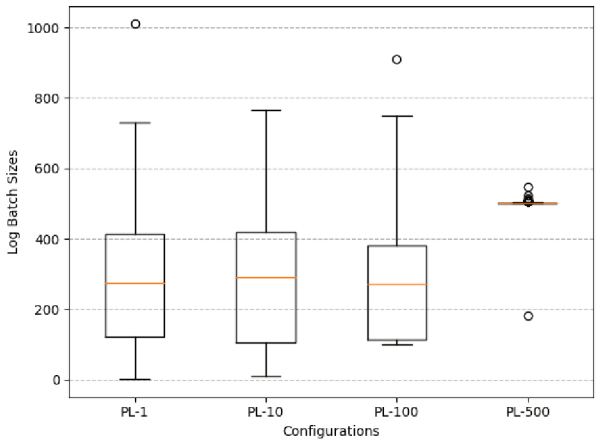
Algorithm 5 Etcd Raft handling procedure with batching

```
1:  procedure STARTBATCH
2:      count ← 0
3:      entryBatch ← NEWBATCH()
4:      applyBatch ← NEWBATCH()
5:      timer ← NEWTIMER(BATCH_TIMEOUT)
6:      mustResetTimer ← True
7:      while True do
8:          if timer.EXPIRED() then
9:              if count = 0 then
10:                 mustResetTimer ← True
11:                 continue
12:
13:             SAVEONWAL(entryBatch)
14:             send applyBatch to applyChannel
15:             count ← 0
16:             mustResetTimer ← True
17:             entryBatch.CLEAR()
18:             applyBatch.CLEAR()
19:
20:         else if receive from raftChannel as state then
21:             entryBatch.APPEND(state.Entries)
22:             applyBatch.APPEND(state.CommittedEntries)
23:             if mustResetTimer then
24:                 timer.RESET()
25:                 mustResetTimer ← False
26:
27:             count ← count + LENGTH(state.Entries)
28:             if count < BATCH_SIZE then
29:                 state.ADVANCE()
30:                 continue
31:
32:             SAVEONWAL(entryBatch)
33:             send applyBatch to applyChannel
34:             count ← 0
35:             mustResetTimer ← True
36:             entryBatch.CLEAR()
37:             applyBatch.CLEAR()
38:             state.ADVANCE()
```

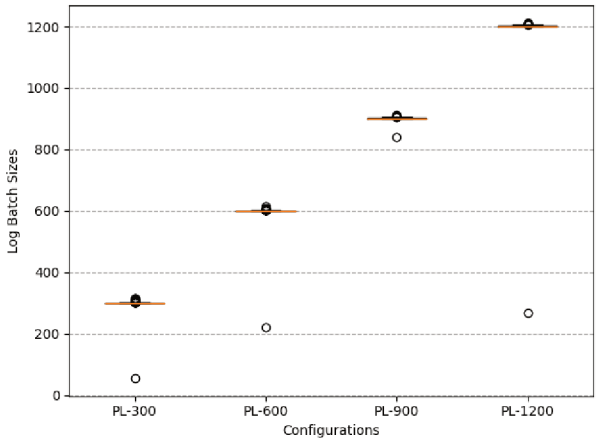
previous evaluation of Chapter 4. As can be seen, setting values below 500 did not necessarily incurred in the persistence of BATCH_SIZE files, mainly because the actual delivery rate of new commands by Raft was higher. After some iterations, we later defined 300 and its multiples as better values for BATCH_SIZE on our environment, on which the actual values of persisted file batch sizes can be seen on Figure 11b.

As for BATCH_TIMEOUT, we opted to configure it as 300 ms after analyzing average latency values of etcd under different configuration scenarios. The timeout necessarily had to

Figure 11 – Analysis over the actual persisted log files considering different `BATCH_SIZE` configurations.



(a) `BATCH_SIZE` values from Chapter 4's evaluation



(b) New `BATCH_SIZE` values

Source: The Author.

be higher than the average latency of our most expensive configuration, so that it would not interfere with etcd's execution and terminate on-going client requests.

As for etcd's recovery procedure, the most relevant adaptations made were (i) the disable of WAL snapshots for our evaluation, and (ii) the implementation of alternative methods for handling log files invoked only for our recovery technique. Our approach reads our own log file's format from disk instead of etcd's WAL, and loads recovered commands on Raft's log without requiring an ordered sequence of commands by their indexes, which is not assured, nor necessary, to our recovery protocol.

5.4 READ CONSISTENCY AND ISOLATION

Etcd allows clients to explicit request both *serializable* and *linearizable* consistency levels.[2] On a distributed environment, linearizable requests ensure the property of *linearizabil-*

[2] https://etcd.io/docs/v3.3/learning/api_guarantees/

ity (HERLIHY; WING, 1990), which makes the system appear as if there were only one copy of the data and all operations are atomic. In terms of performance, linearizable requests are more expensive since they rely on live consensus to retrieve a majority of replicas' state, whereas serializable read requests are attended only by reading the requested replica's state. No matter the consistency level requested, serializable isolation level is always ensured, which defines that read operations may not return intermediate data from other concurrent transactions.

Since read operations are not registered as entries in the log, both types of read operations are not accounted on the increment of the batch counter. If a batch is currently being filled and a linearizable read operation is requested, that read's response is delayed until that batch is filled, persisted, and executed, or the timeout is reached. Conversely, serializable reads are immediately attended, always returning the latest value for the request key. During experiments, we demonstrate results studying both consistency levels for read operations.

5.5 RELEVANT OPTIMIZATIONS

While developing the integration of our logging approach within etcd, we faced different optimization opportunities that impacted positively on the evaluation results. Many of these strategies arose at early stages of experiments, and have confirmed its gains after some iterations of experiment execution. To name a few, related to:

- **Synchronization:** The first implementation of our technique considered the invocation of the log procedure by different concurrent routines. As we coupled into etcd, we notice that although originated from concurrent client requests, the log procedure is always invoked by a single routine following the same delivered order of commands by the Raft protocol. With this in mind, we were able to remove the mutex needed to ensure mutual exclusion for the cursor variable, detailed in Section 3.3.

- **Memory allocation:** In instances where we needed to copy data and could not pass by reference (*e.g.* when loading log commands from disk into memory), we explored the extensive use of memory pre-allocation whenever possible, instead of relying on the automatic allocation offered by dynamic sized data structures. Also, when capturing performance metrics such as the server's throughput and recovery time, we always measured and saved data on in-memory buffers first instead of directly invoking I/O operations to secondary storage, and then flushed over the buffer contents to a file during application shutdown. On benchmark studies, this strategy alone resulted in an $\approx 18\%$ improvement on the application's recovery time.

- **Algorithm design:** The Descending recovery strategy relies on reading the log files on their descending order by log indexes to safely discard overwritten commands. When reading the names of the child files from a certain directory, we can never ensure that the names are returned on a specific order because this is OS-depended, so sorting the

list of returned names is always necessary. After analyzing the list of file names returned on numerous experimental iterations, we identified that many were returned on a "near-ascending" order, which constitutes the approximately worst-case scenario for a sort algorithm that intends to sort on descending order. By switching the sort algorithm from a QuickSort implementation on first versions of Descending to MergeSort, we were able to reduce the total recovery time taken on this strategy by ≈ 15 ms, when considering log files generated by the execution of one hundred thousand (10^5) commands.

- **Raft integration:** When proposing a list of new entries to etcd's Raft implementation, a verification procedure is run to ensure that the new entries proposed are sorted by their indexes. Some other procedures identify the last index of an entries list by counting the index of the first entry plus the actual list size. Both of these algorithms do not support the retrieved log of our technique, which is sorted only by their batch intervals (not within the batch) and can have missing indexes of safely discarded entries. Instead of sorting the entire sequence of commands and filling missing indexes with blank entries, we decided to simply append a dummy entry with the upper index of the last retrieved batch before proposing state to Raft, and modified the verification procedures to allow the proposal of an unsorted log. Also, we modified the procedures that identify the last index on the list to just look at the index of the last entry, that we ensure will always be an entry with the correct last index, preserving the $O(1)$ time complexity. Since the proposal procedure now accepted an unsorted sequence, we could also avoid sorting the entries retrieved with the Descending strategy into an ascending order before proposing, and just swap the first/last entries in-place to ensure the correct last index.

Also, we identified other optimization opportunities that we did not implement, but could be evaluated on future work:

- **Serialization:** On etcd's implementation, received commands are kept serialized on an `Entry` structure throughout most execution flow and are only deserialized when they are executed, which happens solely after they are logged. Since our compaction approach relies on identifying a command's key during execution, we must pay the costs of deserializing commands at runtime. This deserialization procedure is done from the Protocol Buffers binary encoding. In this sense, a possible optimization would be to modify the `Entry` proto definition to include the corresponding command's key as an extra attribute, avoiding the entire command deserialization to interpret its key.

- **WAL format:** When reading a log file's content from persisted storage, we do not implement any logic to decide whether a command must be retrieved or not, even if a command is later discarded from a log process strategy, such as Descending. In this sense, we could combine the idea of the previous optimization to include a command's key and its index on the actual WAL format before the actual command's representation. This way we

could first parse the key/index pair for each command to decide if it must be read of not, avoiding I/Os for unnecessary keys.

5.6 BENCHMARK AND WORKLOADS

For the evaluation scenarios, we utilized the YCSB benchmark and workloads previously defined in Section 4.1 with a few adaptations. In YCSB-A, B and C, we switched from the uniform distribution to zipfian, following the same configuration established in (COOPER et al., 2010). This decision was made with the intention to improve the accuracy and comparability of our results with other usages of YCSB in the literature. Differently from uniform in which an item is chosen uniformly at random, the zipfian distribution divides its value set into a head and a tail, in which some values have a high chance of being draw, *i.e.*, the head of the distribution, while most values will have a lesser chance, *i.e.*, the tail. Also, we ran experiments with two space sizes composed by 10^6 distinct keys, the same range of keys adopted in the previous experiments, and 10^4 distinct keys. We argue that the latter configuration is representative for practical usages of etcd in production environments, such as to keep track of cluster membership changes on Kubernetes (JEFFERY; HOWARD; MORTIER, 2021).

Another difference from the previous evaluation is that instead of processing an input file following a specific workload on our prototype, we now directly send API requests to etcd from the benchmark tool. Initially the utilized port of YCSB did not have support for etcd as it had for other databases, but we easily implemented an adapter utilizing etcd's *Software Developer Kit* (SDK), publicly available on a Github repository.[3]

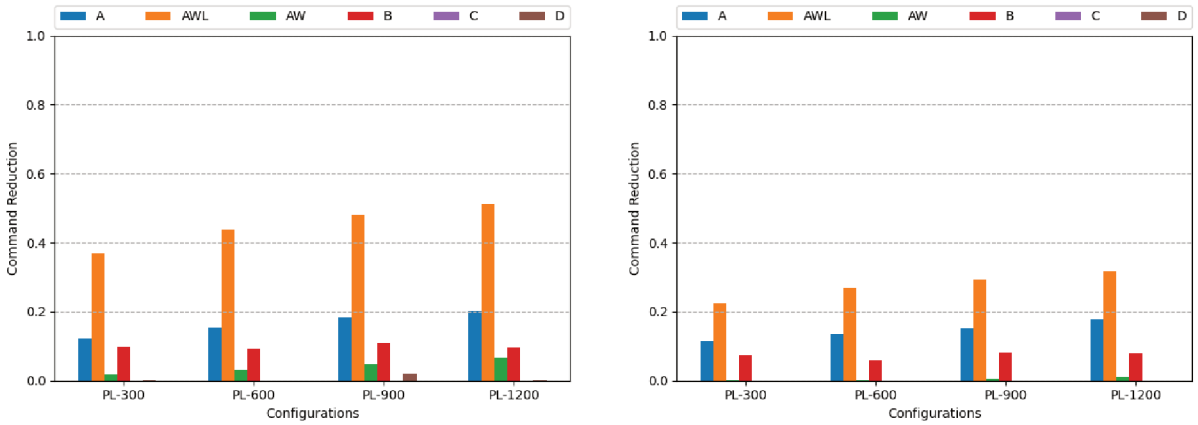
Since it now constitutes of a distributed environment, experiments were executed on the *Emulab Utah* research cluster (WHITE et al., 2002) utilizing two different nodes: (i) a Dell Poweredge R430 node for the benchmark tool, equipped with two 2.4 GHz, 64-bit, 20MB cache, 8-Core Xeon E5-2630v3 processors; 64 GB 2133 MT/s DDR4 random access memory; and two 1 TB HDD with 7,200RPM; and (ii) a Dell Poweredge R820 node running etcd, equipped with four 2.2 GHz, 64-bit, 16 MB cache, 8-Core Xeon "Sandy Bridge" processors; 128 GB 1333 MHz DDR3 RAM; and six 600 GB HDD with 10,000RPM. The decision for a different node on the server was driven by its availability of up to six I/O devices, which allowed a better evaluation for our parallel I/O strategy. Both nodes operate under a Ubuntu 18.04LTS image, and Go binaries were compiled with go-1.15. All modifications were done over etcd v3.4.14,[4] utilizing default values of 1s election timeout and 100ms heartbeat on experiments.

5.7 GENERATED LOG FILES ANALYSIS

The first analysis compares the log files generated after the execution of etcd configured with its standard logging scheme (SL) against etcd running our proposed logging approach

[3] https://github.com/Lz-Gustavo/go-ycsb/tree/etcd-v2
[4] https://github.com/Lz-Gustavo/etcd/tree/release-3.4

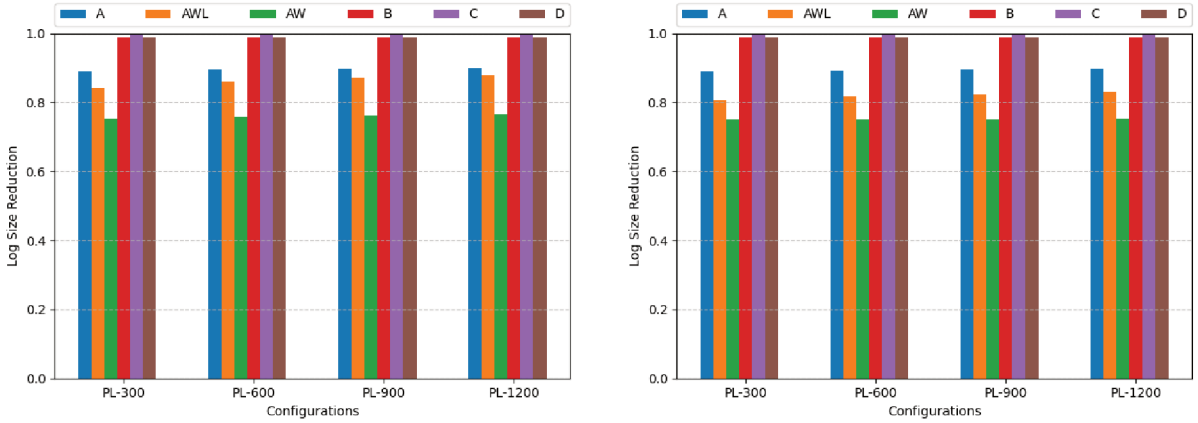
Figure 12 – Number of commands reduction on generated recovery logs for etcd, normalized to SL values.



(a) 10^4 distinct keys (b) 10^6 distinct keys

Source: The Author.

Figure 13 – Total size reduction on generated recovery logs for etcd, normalized to SL values.



(a) 10^4 distinct keys (b) 10^6 distinct keys

Source: The Author.

(PL) under different workloads and batch size configurations, and data normalized to SL values. For all workloads, the number of commands submitted by YCSB was 10^5.

The command reduction from generated logs with PL when compared to SL are presented in Figure 12, where we perceive an improvement of $\approx 36\%$ for YCSB-AWL compared to SL with a batch size of 300 commands. The reduction increases to $\approx 50\%$ for batches of 1200. This is explained by the latest distribution of command keys on YCSB-AWL, where it stimulates overwrites on recent updated keys. The greater the batch size, the greater the odds of such operations being identified and discarded by our technique. YCSB-A also exhibits a similar effect due to the zipfian distribution, but with a lesser reduction increase in each configuration. When comparing Figures 12a and 12b, we can notice the greater reduction on the former due to the smaller key space, being $\approx 15\%$ better than 12b in all configurations of YCSB-AWL. The execution with the remaining workloads exhibited no significant gains of PL over SL.

Figure 13 shows the total file size reductions of our strategy. On this study, a reduction above 70% can be perceived for all workloads and batch sizes, being nearly 99% for most read-intensive workloads of YCSB-B, C and D. This is explained by the fact that although not logging read commands, etcd log files are all pre-allocated with an initial value of 64 MB, no matter the submitted workload. Once a file exceeds the 64 MB size, a new file of the same size is allocated. Differently, our logging strategy does not pre-allocate data blocks to log files.

5.8 RECOVERY TIME IMPACT

The recovery time analysis aims to compare the time taken to fully restore an etcd's replica state after reading log files persisted by its default implementation (SL) against logs persisted by our log compaction strategy (PL). On all configurations, we measured the total time to restore the application's state by interpreting log files generated after the execution of 10^5 commands, being an unique set of log files for each workload. For PL executions, we evaluate the performance of both recovery strategies described in Section 3.4: *Naive* and *Descending*, and we also present a breakdown analysis of the time taken by the recovery procedure steps. Every combination of workload, batch, and recovery strategy was executed 30 times, and the average values of these iterations are presented.

Table 3 depicts the values obtained for the state recovery of etcd on its standard log configuration (SL) while submitted to all studied YCSB workloads, with 10^4 distinct keys. As expected, the slowest trace was observed for YCSB-AW, a fully write and uniformly distributed workload, whereas the fastest was YCSB-C, a read-only workload. This can be explained by the large difference of $\approx 10^5$ in the number of commands, and raw data size between the log files generated from both workloads.

In Table 4 we present the values of the recovery duration of PL for YCSB workloads with 10^4 distinct keys. All different combinations of workload, batch size, and recovery strategy are shown. Gray colored cells represent the values obtained utilizing the *Descending* recovery strategy, whereas the blank ones show the values from *Naive*. Read (i), Receive (ii), and Apply (iii) columns present a breakdown of the duration on different steps of the recovery procedure,

Table 3 – SL recovery results with a 10^4 distinct keys configuration.

Workload	Total (ms)
YCSB-A	506.53
YCSB-AWL	654.22
YCSB-AW	663.09
YCSB-B	281.45
YCSB-C	99.23
YCSB-D	285.07

Source: The Author.

Table 4 – PL recovery results with 10^4 distinct keys. Gray colored cells represent the values obtained utilizing the *Descending* recovery strategy, whereas the blank ones show the values from *Naive*.

Batch	YCSB	Read (ms)	Receive (ms)	Apply (ms)	Total (ms)	% SL
300	A	96.64	66.28	103.96	266.89	47.31
		147.56	76.21	18.01	241.77	52.27
300	AWL	151.49	68.64	151.24	371.37	43.23
		211.27	82.03	13.07	306.37	53.17
300	AW	191.25	72.93	188.11	452.29	31.79
		292.66	73.33	13.94	379.93	42.70
300	B	30.17	63.91	19.09	113.17	59.79
		37.97	60.41	8.53	106.91	62.02
300	C	17.61	61.05	1.76	80.42	18.95
		17.60	61.11	1.76	80.48	18.90
300	D	31.81	64.40	23.80	120.01	57.90
		40.99	60.38	8.34	109.70	61.52
600	A	87.87	70.78	63.64	222.28	56.12
		135.02	76.20	17.76	228.98	54.79
600	AWL	111.17	71.38	128.22	310.77	52.50
		161.14	69.25	16.91	247.31	62.20
600	AW	160.48	74.00	181.64	416.12	37.24
		255.61	71.83	17.42	344.86	47.99
600	B	29.94	60.44	17.08	107.46	61.82
		35.76	63.40	9.46	108.63	61.41
600	C	17.72	59.64	1.63	78.99	20.40
		17.22	61.31	1.72	80.25	19.13
600	D	30.54	61.76	16.83	109.14	61.71
		38.88	60.51	12.61	112.00	60.71
900	A	86.04	68.05	53.78	207.87	58.96
		130.36	67.83	17.67	215.86	57.38
900	AWL	100.81	65.46	96.57	262.84	59.82
		149.44	62.86	19.34	231.63	64.59
900	AW	154.81	74.05	148.88	377.73	43.03
		245.46	66.64	19.62	331.72	49.97
900	B	30.25	60.92	17.83	109.00	61.27
		36.86	61.50	7.67	106.03	62.33
900	C	17.86	62.81	1.63	82.29	17.07
		16.91	61.65	1.68	80.24	19.14
900	D	31.20	62.58	19.02	112.80	60.43
		37.65	61.17	11.75	110.57	61.21
1200	A	84.71	69.29	54.02	208.01	58.93
		129.79	65.74	18.56	214.09	57.73
1200	AWL	103.55	70.72	84.33	258.59	60.47
		141.99	64.30	18.97	225.27	65.57
1200	AW	152.50	88.51	137.17	378.18	42.97
		233.09	94.85	15.12	343.06	48.26
1200	B	29.78	60.34	17.26	107.39	61.85
		35.76	66.70	6.99	109.45	61.11
1200	C	23.08	61.45	1.58	86.11	13.22
		17.94	62.72	1.65	82.31	17.05
1200	D	31.60	62.11	17.62	111.33	60.95
		39.56	61.26	9.32	110.13	61.37

Source: The Author.

being respectively: (i) the time taken to read the log contents from disk and propose them to the underlying protocol; (ii) the time to receive the new command entries from the protocol routine; and (iii) the duration to execute the received commands and catch up with the protocol state. The %SL column shows percentage time reductions over SL for the same workload, considering the values of Table 3 as the baseline.

Every scenario demonstrated a time reduction when compared to SL for the same workload. The highest reduction was seen for YCSB-AWL, 1200 batch size, with descending recovery strategy, with a 65.57% improvement over SL. As expected, higher batch size values positively influenced the recovery time, since they directly improve command compaction by increasing the odds of identifying command overwrite during normal execution. When comparing results obtained by the Naive recovery strategy (white background cells) against the ones by Descending (gray colored cells) under the same workloads, we can perceive a significant improvement for write-intensive workloads such as YCSB-AW and AWL, where for 300 and 600 batch size incurred $\approx 10\%$ gains by using Descending. As the batch size increases, the performance improvements of Descending from Naive tends to decrease, which is explained by the fact that log files are indeed more compacted under higher batch sizes, diminishing gains of the post-compaction optimization done by Descending. Specially in write-intensive workloads, Descending configurations also shown higher duration values on the Read step due to the mentioned optimization, but compensated on the Apply step by greatly reducing the number of commands necessary to execute. As an example, on YCSB-AW, 600 batch size, although increasing the Read step by 95 ms, the Descending strategy was able to reduce the Apply step by 164 ms, resulting in a 10% improvement over Naive.

Table 5 extends the analysis shown on Table 3 by increasing the number of distinct keys from 10^4 to 10^6. With a wider key space, it is expected larger log files during recovery due to the lesser keys overwrite, resulting in a recovery time increase. When comparing both tables, this indeed happened for the write-intensive workloads YCSB-A, AW, and AWL, being the latter the highest increase with 28 ms. The same effect did not impacted remaining workloads, mainly due to their high read proportion and small log size.

The same analysis of higher distinct key values is shown for PL on Table 6. When

Table 5 – SL recovery results with a 10^6 distinct keys configuration.

Workload	Total (ms)
YCSB-A	511.28
YCSB-AWL	682.13
YCSB-AW	677.88
YCSB-B	278.85
YCSB-C	112.66
YCSB-D	283.27

Source: The Author.

comparing its results with Table 4, we perceive that the increase in the number of distinct values incurred an overall increase in the recovery time for all configurations and workloads. Also, the improvements by choosing Descending over the Naive strategy on write-intensive workloads were diminished, being the highest gain of $\approx 6\%$ observed for YCSB-AWL, and batch size 300. These effects are related to the lesser command overwrite under a higher key space, which impacts both the online compaction done by PL during normal execution, and the post-optimization of the Descending recovery strategy.

5.9 EXECUTION OVERHEAD

We evaluate the impact of our proposed logging approach (PL) on etcd's performance by executing series of experiments comparing the application's saturation with PL, under different configurations, against the application's saturation with etcd's standard logging scheme (SL). The different configurations evaluated for PL include the variation in the batch size and the increase in the number of disk devices, exploring the parallel execution of I/O operations. For SL, we also demonstrate results from using the same implementation of batching and command retention from PL, but without the command compaction and synchronization costs related to our technique. This decision aims to allow an evaluation of our compaction strategy without the interference of batching alone, since both scenarios would be implementing the same technique. All results demonstrated here utilize a linearizable consistency level (Section 5.4) for read operations. Experiments considering the weaker consistency level of serializability are presented on Section 5.10.

As a baseline result for all of our technique's evaluation, Figure 14 shows a saturation analysis of a single-node deployment of etcd running on our infrastructure. The study was conducted over the default implementation of etcd without any interference of our batching implementation. This experiment is based on the consecutive executions of the load generator with an increase in the number of concurrent clients served by the application. The left-most sample is taken with 1 client, followed by 150, 300, and all remaining points have a linear increase of 150. The y-axis shows the 90^{th} percentile latency values for each execution and x-axis represents the average throughput. Workloads with a mixed read/write ratio such as YCSB-A, B and D presented an earlier saturation and a significant lower throughput than YCSB-AW, AWL, and C workloads. This same effect of an earlier saturation is perceived for these workloads on a related study (SONBOL et al., 2020), where the authors attributed the effect to the increase in the number of write operations. We argue that it may not be the case since a better performance was observed for YCSB-AW and AWL, our custom fully write workloads. It may be related to a possible interference that a proportion of read operations may have on the protocol's batching, but the in-depth analysis of this effect runs out of scope for this study. On average, latency values for most workloads was kept at $\approx$ 90ms at 80% maximum workload, with the exception of YCSB-C due to being a read-only profile.

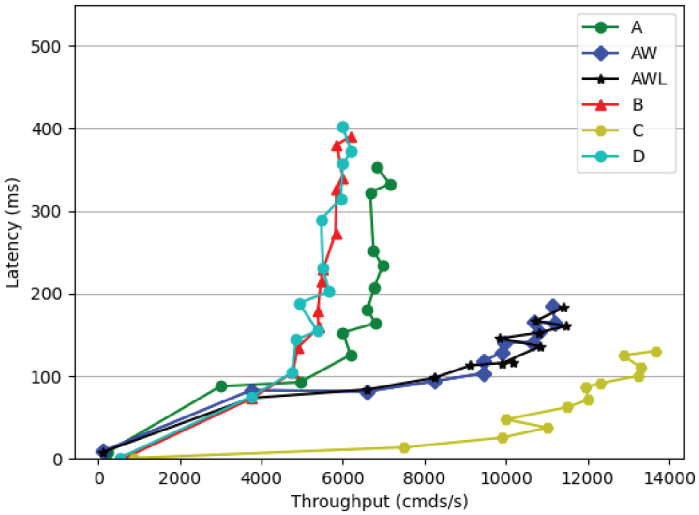
Figure 15 illustrates boxplots for the latency values observed for different workloads,

Table 6 – PL recovery results for 10^6 distinct keys. Gray colored cells represent the values obtained utilizing the *Descending* recovery strategy, whereas the blank ones show the values from *Naive*.

Batch	YCSB	Read (ms)	Receive (ms)	Apply (ms)	Total (ms)	% SL
300	A	105.70	85.74	119.67	311.11	39.15
		159.99	85.65	38.66	284.31	44.39
300	AWL	181.06	101.88	177.59	460.53	32.49
		274.76	96.36	50.20	421.32	38.23
300	AW	221.67	88.66	222.79	533.12	21.36
		335.33	110.77	96.93	543.03	19.89
300	B	34.33	75.51	38.94	148.77	46.65
		47.45	68.95	16.69	133.10	52.27
300	C	21.25	68.95	1.90	92.10	18.25
		20.84	70.46	1.96	93.26	17.22
300	D	35.54	70.52	50.34	156.40	44.79
		43.77	71.64	14.13	129.55	54.27
600	A	97.92	78.51	71.51	247.94	51.51
		150.14	82.94	39.35	272.42	46.72
600	AWL	143.22	78.83	143.65	365.70	46.39
		218.38	89.64	50.72	358.74	47.41
600	AW	181.45	92.38	189.81	463.64	31.60
		312.20	97.43	93.58	503.22	25.77
600	B	34.75	71.35	33.24	139.34	50.03
		41.66	73.58	16.98	132.22	52.58
600	C	20.93	70.47	1.89	93.29	17.20
		20.76	69.71	2.00	92.47	17.93
600	D	34.86	69.66	37.22	141.74	49.96
		44.69	72.75	16.72	134.16	52.64
900	A	93.18	82.23	63.66	239.07	53.24
		145.21	71.42	40.94	257.57	49.62
900	AWL	133.59	82.64	129.38	345.61	49.33
		213.57	71.93	48.86	334.37	50.98
900	AW	176.06	87.23	169.72	433.01	36.12
		297.11	77.82	97.23	472.16	30.35
900	B	34.88	69.16	42.80	146.83	47.34
		41.73	70.56	14.96	127.25	54.37
900	C	20.68	70.24	1.87	92.79	17.64
		20.46	71.44	1.81	93.72	16.82
900	D	34.23	69.87	47.22	151.32	46.58
		41.48	70.21	17.26	128.95	54.48
1200	A	91.02	75.73	57.19	223.94	56.20
		142.13	70.83	43.83	256.80	49.77
1200	AWL	124.46	112.25	120.37	357.07	47.65
		210.66	71.94	46.43	329.03	51.76
1200	AW	168.67	111.99	176.28	456.93	32.59
		296.95	77.88	95.97	470.79	30.55
1200	B	35.63	70.18	48.48	154.29	44.67
		40.43	70.15	10.43	121.01	56.60
1200	C	20.12	72.21	1.70	94.03	16.54
		21.91	71.46	1.83	95.20	15.50
1200	D	36.14	69.04	42.71	147.89	47.79
		42.38	68.52	12.67	123.58	56.37

Source: The Author.

Figure 14 – Saturation analysis of SL for different YCSB workloads.

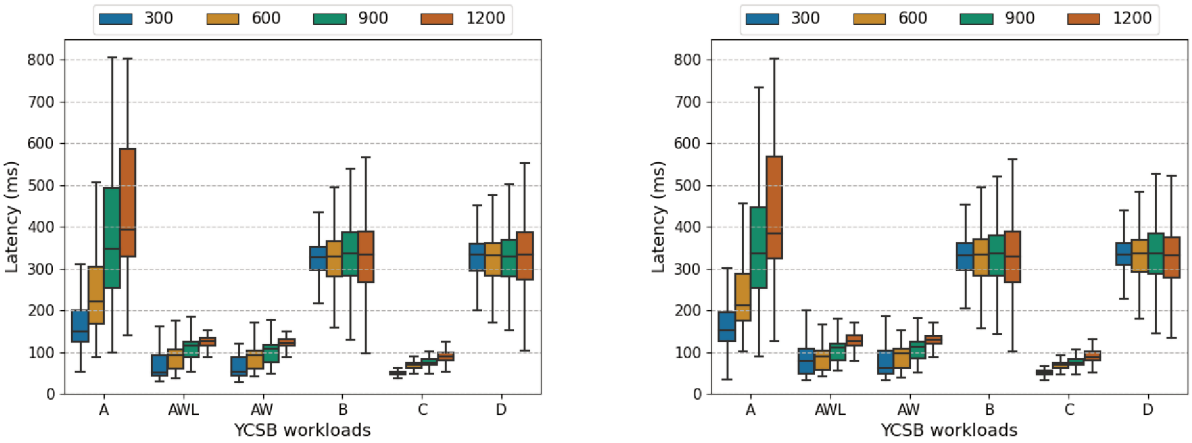


Source: The Author.

demonstrating an overview of the performance impact imposed by PL. From prior experimental results, we knew beforehand the maximum workload supported for each batch size configuration, *i.e.*, the maximum throughput once application starts to saturate. Thus, we depict the latency values while the application was submitted to $\approx 75\%$ of its maximum capacity. As seen, PL did not exhibit a significant performance impact over SL under different workload and batch configurations. The mixed workloads YCSB-A, B and D demonstrated a huge standard deviation and consistently higher latency values when compared to other workloads. This is not only related to their earlier saturation seen on Figure 14, but also to the following behavior on our approach when submitted to mixed workloads: since we do not account read operations as an increment to the batch counter, only the workload's write throughput influences the rate on which batches are filled, and, consequently, the rate on which write and linearizable read operations are responded. However, this is not the case for a fully-read workload like YCSB-C, as batches are only initialized by write operations.

Figure 16 depicts a performance analysis considering the same YCSB-AW workload and batch sizes of 300 and 1200 commands, on which the saturation curves for SL and PL with different numbers of disk devices are shown. As seen on Figure 16a, the first two executions with a number of clients $<$ `BATCH_SIZE` resulted in the observation of latency values $\geq$ `BATCH_TIMEOUT`. This is an important aspect of our approach, and happens because these number of clients do not incur the necessary throughput in order to fill a batch of 300 commands within the timeout. It is important to mention that the throughput is directly influenced by Raft's delivery rate of commands, and not only by client request rate. The saturation its reached for PL configuration at $\approx 8,000$ commands/s and 750 clients, whereas for SL it starts later at $\approx 10,000$ commands/s and 1050 clients. By increasing the number of I/O devices on PL-2disk,

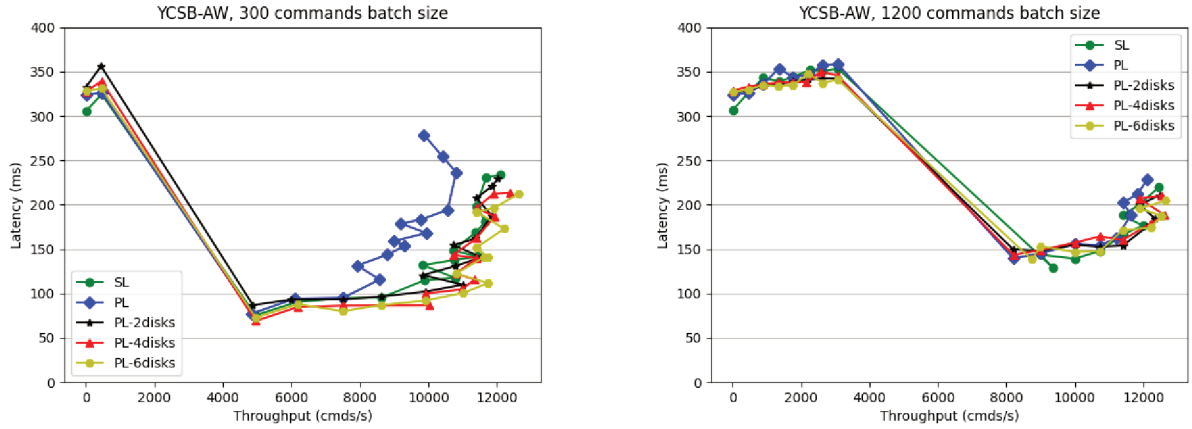
Figure 15 – Latency comparison of SL and PL with a linearizable read consistency, considering different workloads and batch size configurations.



(a) SL (b) PL

Source: The Author.

Figure 16 – Saturation analysis of batch sizes 300 and 1200 for YCSB-AW.



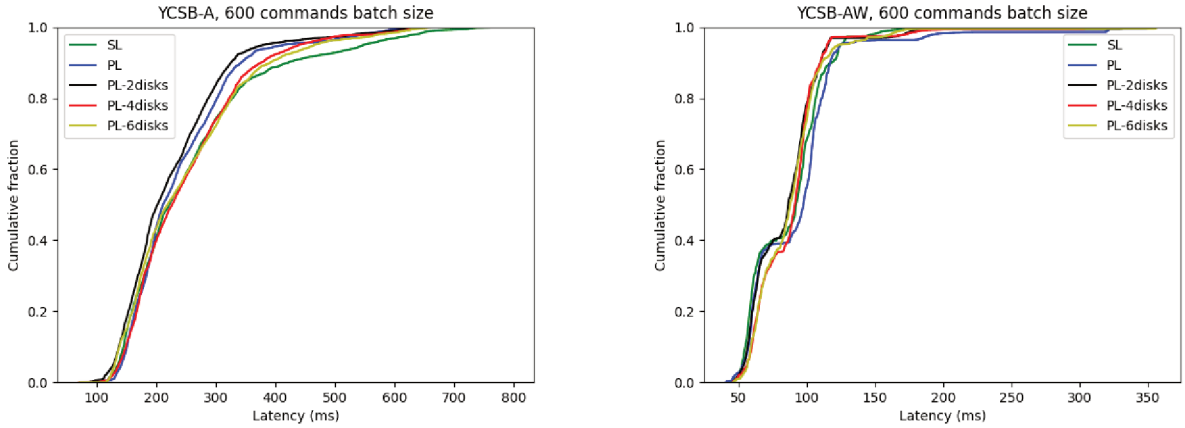
(a) batch size of 300 commands (b) batch size of 1200 commands

Source: The Author.

PL-4disk and PL-6disk configurations, we identify a slightly latency reduction and the reach of greater throughput values when compared to PL, depicting a similar saturation as SL. We could not conclude any advantage regarding the increase of I/O devices when comparing the application's performance between PL-2disk, PL-4disk and PL-6disk. Figure 16b depicts the saturation analysis of the same workload, but considering a batch of size of 1200 commands. Under this configuration, we identify the increase in the number of points with latency values $\geq$ `BATCH_TIMEOUT`, and an increase in the average latency observed for all remaining scenarios by $\approx$ 50 ms. Despite the significant increase in latency due to the greater retention of commands before persistence, no relevant effect was observed on the throughput. Similarly, the usage of more than two I/O devices did not result in significant performance gains.

Figure 17 shows a cumulative distribution fraction (CDF) of the latency values from

Figure 17 – Cumulative fraction of latency values on different workloads.



(a) YCSB-A (b) YCSB-AW

Source: The Author.

YCSB-A and YCSB-AW workloads with a batch size of 600 commands. As seen, while executing with the mixed workload of a 50/50 write/read ratio in Figure 17a, we observe higher latency values when compared to a 100% write workload in Figure 17b. This is related to saturation effect seen in Figure 14 and the fact that reads are not accounted on the increment of batches. Also, when analyzing YCSB-A's distribution, we cannot perceive any noticeable difference at a 0.5 fraction, which would presume a difference for the latency values between the two operation types.

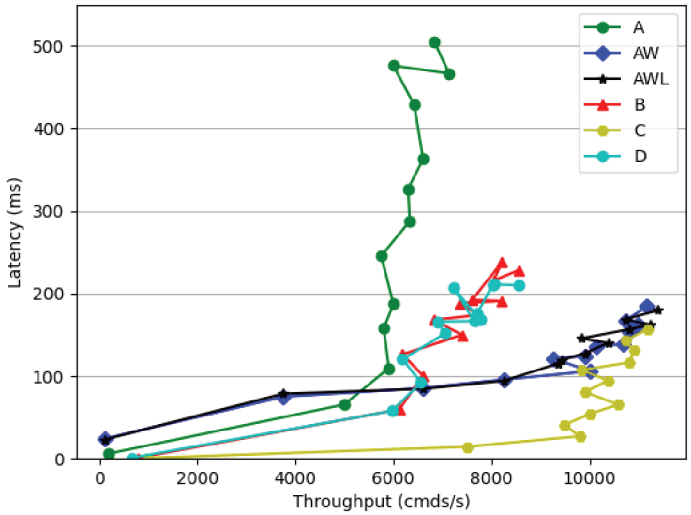
5.10 EXPLORING SERIALIZABLE CONSISTENCY

On this section, we evaluate PL and SL's performance considering a serializable read consistency model. Under this configuration, our compaction approach avoids the retention of read operations while batches are being filled, and immediately answers read requests with the operation's key last durable state.

In Figure 18, we analyze the default etcd's saturation under different YCSB workloads. As expected, when compared to Figure 14, the weaker consistency level had a positive impact on the saturation curve of mixed read intensive workloads of YCSB-B and D, since read operation's response are no more delayed until batch persistence. YCSB-A had no significant impact on its 90^{th} percentile latency value, but we discuss the impact for its read operations later. Other workloads observed no significant changes either because of its fully write percentage, *e.g.* YCSB-AW, AWL, and the read intensive YCSB-C.

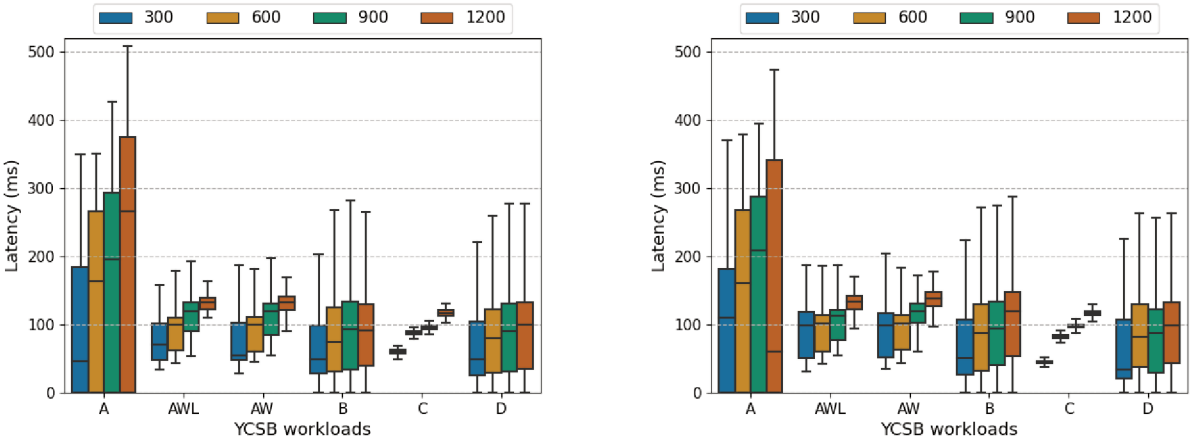
In Figure 19, we show the same experiment demonstrated on Figure 15, but with a serializable consistency level for read requests. This configuration imposes a latency decrease on all mixed workloads of YCSB-A, B and D (lower y-axis limit). This is related to the fact that linearizable reads are only attended after a batch persisted and execution, and their mere

Figure 18 – Saturation analysis of SL for different YCSB workloads with serializable read consistency.



Source: The Author.

Figure 19 – Latency comparison of SL and PL with a serializable read consistency, considering different workloads and batch size configurations.



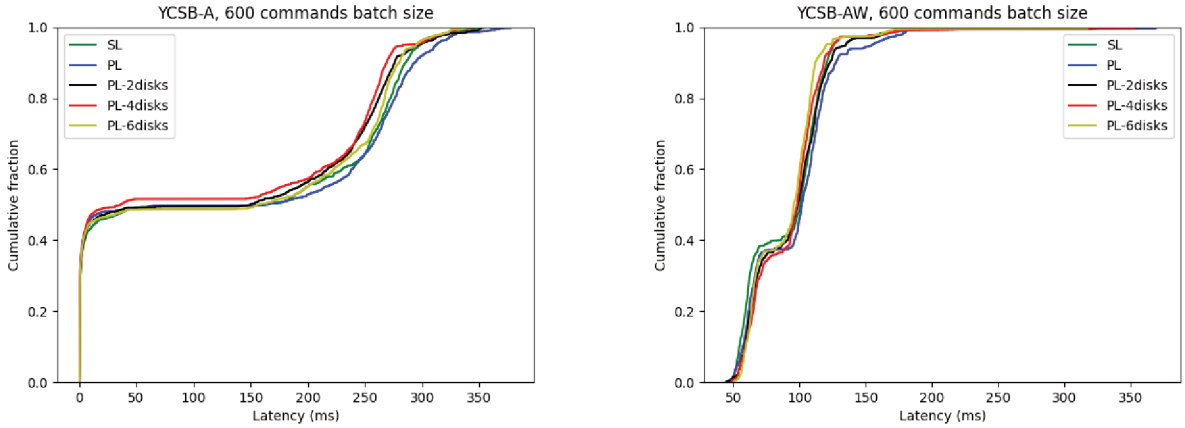
(a) SL (b) PL

Source: The Author.

presence lowers the write operation throughput that accelerates batch filling. Although being a 100% read workload, YCSB-C exhibit similar values with the weaker consistency level, because batches are only initialized by write operations. Similarly from the last experiment, we can also conclude by comparing Figure 19a and Figure 19b that PL show no significant performance loss over SL.

Figure 20 shows a cumulative distribution fraction (CDF) of the latency values from YCSB-A and YCSB-AW workloads with a 600 commands batch size and a serializable read consistency. The same higher latency values of Figure 20a when compared to Figure 20b is also present, being related to same effects of earlier saturation and that reads are not accounted

Figure 20 – Cumulative fraction of latency values on different workloads with serializable read consistency.



(a) YCSB-A (b) YCSB-AW

Source: The Author.

on the increment of batches. From the distribution shown for YCSB-A, we notice that a ≈ 0.5 fraction depicts latency values below 100 ms, being roughly the same as the read/write ratio of YCSB-A. By comparing against the CDF for YCSB-A with linearizable consistency on Figure 17a, we perceive that operations with such lower latency values were not seen with the stronger consistency level, concluding that the first half of the distribution seen in Figure 20a is indeed serializable consistent reads.

6 CONCLUSION

Our work presents a logging approach that exploits application semantics to safely discard entries from command logs, delivering compacted log files that permit a faster state recovery by benefiting both log transferring and 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 reduce logging overhead and alleviate I/O bottlenecks, the proposed technique explores the concurrent execution between compaction and persistence tasks, and implements other optimizations regarding log management, such as batching and parallel I/O.

Two different recovery strategies, named Naive and Descending, were also presented, each performing better depending on the submitted workload. Descending performs better on write-intensive workloads with a high chance of command overwrite, since it implements a post compaction of overwritten keys during the recovery phase to return an optimal minimal state. In the contrary, Naive outperforms Descending on every other workload profile, specially read-intensive workloads with a low chances of command overwrite, as it avoids any other process over logged commands besides simply retrieving them from persisted storage.

By evaluating over a key-value store prototype, we demonstrated 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 concurrent execution. For instance, in a balanced workload composed of 50% reads and 50% writes, our approach delivers a 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 also demonstrated to scale-up with the addition of more storage devices.

Also, to provide a more accurate analysis of our technique we opted to evaluate it in a more realistic environment. In this sense, we created a fork of etcd, an open-source database known in the cloud computing industry, and customized it to utilize our log compaction strategy at run-time and during recovery. This implementation effort brought a few challenges, such as to understand the vast code base in order to implement the new compaction features without compromising the system's correctness. The most challenging feature to implement was definitely the idea to hold commands' execution while still delivering new ones while batches are being filled, mainly because etcd is strongly coupled to its Raft implementation, which required modifications on the protocol's internals.

By evaluating the implementation of our log compaction technique on etcd, we shown that our approach can significantly reduce the time taken to recover from a state of 10^5 commands, and reach up to a 65.57% improvement under some workloads. These benefits arise from the online compaction approach and by the Descending log processing strategy, that eliminates commands' overwrite not previously identified during application's execution. In terms

of performance, we identified that our approach imposes an increase on etcd's latency for most studied workloads. This latency increase is linked to our implemented batching approach that enforces the retention of client responses until logging and execution, which happens only after batches are filled or a timeout is reached. We confirm this behavior by comparing our approach against etcd implementing the same batching strategy, where we could not identify any other performance loss, specially when parallel I/O was utilized.

6.1 FUTURE WORK

Our contribution opens up opportunities for future optimizations and research regarding log compaction. For instance, one could extend the same idea of an online log compaction and evaluate it on other database solutions, distributed or not, under the influence of a broad variety of workloads. Relaxing the proposed model and contributing with a compaction approach for more complex operations, such as multi-variable update commands, would be of great interest in the research area.

As a follow up contribution from our approach, one could apply the idea of the optimal compaction done by the Descending strategy during recovery to propose a second compaction phase done by asynchronous tasks over log files previously persisted by our technique. This way, the costs in discarding overwritten commands not previously identified during the online compaction would fall on concurrent routines, and not during recovery, which could be beneficial under some workloads and when executed over parallel architectures. By considering the combination of these asynchronous compaction tasks with the proposed online compaction, the `BATCH_SIZE` and `BATCH_TIMEOUT` parameters could be re-evaluated to diminish latency overhead while not sacrificing its compaction efficiency, and they could be dynamically tuned depending on the submitted workload. Since logs would be optimal, with at most one command per key, a parallel execution of recovered commands could also be explored.

BIBLIOGRAPHY

AGUILERA, M. K.; CHEN, W.; TOUEG, S. Failure detection and consensus in the crash-recovery model. **Distributed computing**, Springer, v. 13, n. 2, p. 99–125, 2000.

ASSUNÇÃO, M. D. et al. Big data computing and clouds: Trends and future directions. **Journal of Parallel and Distributed Computing**, Elsevier, v. 79, p. 3–15, 2015.

ATIKOGLU, B. et al. Workload analysis of a large-scale key-value store. In: **Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems**. [S.l.: s.n.], 2012. p. 53–64.

BALAKRISHNAN, M. et al. Corfu: A distributed shared log. **ACM Transactions on Computer Systems (TOCS)**, ACM New York, NY, USA, v. 31, n. 4, p. 1–24, 2013.

BESSANI, A. et al. On the efficiency of durable state machine replication. In: **2013 USENIX Annual Technical Conference (USENIX ATC 13)**. [S.l.: s.n.], 2013.

CADONNA, D. B. B. **Performance Tuning RocksDB for Your Kafka Streams Application**. [S.l.]: Confluent, 2021. Https://www.confluent.io/blog/how-to-tune-rocksdb-kafka-streams-state-stores-performance/.

CHACZKO, Z. et al. Availability and load balancing in cloud computing. In: **ICCSM, Singapore**. [S.l.: s.n.], 2011.

CLAY, K. **Amazon.com Goes Down, Loses $66,240 Per Minute**. [S.l.]: Forbes, 2013. Https://www.forbes.com/sites/ kellyclay/2013/08/19/amazon-com-goes-down-loses-66240-per-minute/.

CLEMENT, A. et al. Upright cluster services. In: ACM. **Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles**. [S.l.], 2009. p. 277–290.

COOPER, B. F. et al. Benchmarking cloud serving systems with ycsb. In: **Proceedings of the 1st ACM symposium on Cloud computing**. [S.l.: s.n.], 2010.

DAI, Y. et al. From wisckey to bourbon: A learned index for log-structured merge trees. In: **14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)**. [S.l.: s.n.], 2020. p. 155–171.

DONG, S. et al. Optimizing space amplification in rocksdb. In: **CIDR**. [S.l.: s.n.], 2017. v. 3, p. 3.

ELNOZAHY, E. N. et al. A survey of rollback-recovery protocols in message-passing systems. **ACM Computing Surveys (CSUR)**, ACM New York, NY, USA, v. 34, n. 3, p. 375–408, 2002.

HERLIHY, M. P.; WING, J. M. Linearizability: A correctness condition for concurrent objects. **ACM Transactions on Programming Languages and Systems (TOPLAS)**, p. 463–492, 1990.

JEFFERY, A.; HOWARD, H.; MORTIER, R. Rearchitecting kubernetes for the edge. In: **Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking**. [S.l.: s.n.], 2021. p. 7–12.

JUNIOR, G.; AVILA, E. de. **Redução do custo da durabilidade em Replicação Máquina de Estados através de checkpoints particionados**. Dissertação (Mestrado) — Universidade Federal do Rio Grande, 2020.

JUNQUEIRA, F. P.; REED, B. C.; SERAFINI, M. Zab: High-performance broadcast for primary-backup systems. In: IEEE. **2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)**. [S.l.], 2011. p. 245–256.

KREPS, J. **I Heart Logs: Event Data, Stream Processing, and Data Integration**. [S.l.]: " O'Reilly Media, Inc.", 2014.

KREPS, J. et al. Kafka: A distributed messaging system for log processing. In: **Proceedings of the NetDB**. [S.l.: s.n.], 2011. v. 11, p. 1–7.

LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. **Communications of the ACM**, ACM, v. 21, n. 7, p. 558–565, 1978.

LIU, X.; IFTIKHAR, N.; XIE, X. Survey of real-time processing systems for big data. In: **Proceedings of the 18th International Database Engineering & Applications Symposium**. [S.l.: s.n.], 2014. p. 356–361.

LUO, C.; CAREY, M. J. Lsm-based storage techniques: a survey. **The VLDB Journal**, Springer, v. 29, n. 1, p. 393–418, 2020.

MALVIYA, N. et al. Rethinking main memory oltp recovery. In: IEEE. **2014 IEEE 30th International Conference on Data Engineering**. [S.l.], 2014. p. 604–615.

MARANDI, P. J. et al. Filo: consolidated consensus as a cloud service. In: **2016 USENIX Annual Technical Conference (USENIX ATC 16)**. [S.l.: s.n.], 2016. p. 237–249.

MENDIZABAL, O. M.; DOTTI, F. L.; PEDONE, F. Analysis of checkpointing overhead in parallel state machine replication. In: **Proceedings of the 31st Annual ACM Symposium on Applied Computing**. [S.l.: s.n.], 2016. p. 534–537.

MENDIZABAL, O. M.; DOTTI, F. L.; PEDONE, F. High performance recovery for parallel state machine replication. In: IEEE. **2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)**. [S.l.], 2017. p. 34–44.

MENDIZABAL, O. M. et al. Checkpointing in parallel state-machine replication. In: AGUILERA, M. K.; QUERZONI, L.; SHAPIRO, M. (Ed.). **Principles of Distributed Systems**. Cham: Springer International Publishing, 2014. p. 123–138. ISBN 978-3-319-14472-6.

MOHAN, C. et al. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. **ACM Transactions on Database Systems (TODS)**, ACM New York, NY, USA, v. 17, n. 1, p. 94–162, 1992.

NARKHEDE, N.; SHAPIRA, G.; PALINO, T. **Kafka: the definitive guide: real-time data and stream processing at scale**. [S.l.]: " O'Reilly Media, Inc.", 2017.

OLIVEIRA, R.; GUERRAOUI, R.; SCHIPER, A. Consensus in the crash-recover model. **EPFL, Dept. d'Informatique, Tech. rep**, p. 97–239, 1997.

ONGARO, D.; OUSTERHOUT, J. In search of an understandable consensus algorithm. In: **2014 USENIX Annual Technical Conference (USENIX ATC 14)**. [S.l.: s.n.], 2014. p. 305–319.

O'NEIL, P. et al. The log-structured merge-tree (lsm-tree). **Acta Informatica**, Springer, v. 33, n. 4, p. 351–385, 1996.

PAN, F.; YUE, Y.; XIONG, J. dcompaction: Delayed compaction for the lsm-tree. **International Journal of Parallel Programming**, Springer, v. 45, n. 6, p. 1310–1325, 2017.

ROCKSDB. 2013. Https://rocksdb.org/.

SONBOL, K. et al. Edgekv: Distributed key-value store for the network edge. In: IEEE. **2020 IEEE Symposium on Computers and Communications (ISCC)**. [S.l.], 2020. p. 1–6.

WHITE, B. et al. An integrated experimental environment for distributed systems and networks. **ACM SIGOPS Operating Systems Review**, ACM, 2002.

XAVIER, L. G. C. et al. Scalable and decoupled logging for state machine replication. In: SBC. **38th Brazilian Symposium on Computer Networks and Distributed Systems (SBRC)**. [S.l.], 2020. p. 267–280.

XAVIER, L. G. C. et al. Shrinking logs by safely discarding commands. In: SBC. **39th Brazilian Symposium on Computer Networks and Distributed Systems (SBRC)**. [S.l.], 2021. p. 588–601.

YAO, C. et al. Adaptive logging: Optimizing logging and recovery costs in distributed in-memory databases. In: **Proceedings of the 2016 International Conference on Management of Data**. [S.l.: s.n.], 2016. p. 1119–1134.

YU, X. et al. Taurus: A Parallel Transaction Recovery Method Based on Fine-Granularity Dependency Tracking. **CoRR**, 2016. Disponível em: https://math.mit.edu/research/highschool/primes/materials/2016/Zhu-Kaashoek.pdf.

ZHANG, H. et al. In-memory big data management and processing: A survey. **IEEE Transactions on Knowledge and Data Engineering**, IEEE, v. 27, n. 7, p. 1920–1948, 2015.