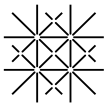

University of Basel

Memory-Bounded Replication of Mutable Data Structures over Immutable Append-Only Logs

MSc Thesis

Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
Computer Networks Research Group
https://cn.dmi.unibas.ch

Examiner: Prof. Dr. Christian Tschudin
Supervisor: Prof. Dr. Christian Tschudin

Sebastian Lukas Philipp
s********.p******@unibas.ch
****-***-***

2022-12-31

Acknowledgments

I would like to express my gratitude to Prof. Dr. Christian Tschudin for the opportunity to write this thesis in his research group, and for the guidance and valuable input and feedback he provided during the course of this thesis.

Additional thanks go to Angela Hunziker for proofreading this thesis report and providing valuable feedback.

Abstract

Append-only logs are data structures which permit random-access read operations, but write operations are limited to appending to the end of the log. Nevertheless, arbitrarily modifications of data can be represented by creating an ever-growing stream of update operations appended to such a log. However, if a new consumer of this update stream wishes to recover the state represented by the log, often the entire log must be kept in storage and be replicated again.

In this thesis report, we present *PREDSL*, a framework which facilitates the implementation of data structures by producing such a sequence of modification operations, and provide implementations for commonly used data structures.

Further on, we designed, implemented and evaluated different strategies by which only a small, contiguous portion of the log – a "sliding window" of the log's latest entries – must be kept in storage and replicated to new consumers.

The results of our evaluation show that these strategies indeed manage to maintain a small sliding window in which all information relevant to reconstruct the entire state of the encoded data structure is represented in a very compact form, rather than spread over the entire log.

Table of Contents

1 Introduction

In this chapter we give a cursory introduction into append only logs, and into our motivation for pursuing this topic. We outline the goals of this thesis and which challenges we need to overcome.

The remainder of this thesis report is structured as follows: In Chapter 2 we highlight the history of append-only logs up to recent developments. In Chapter 3 we present *PREDSL*, the framework we implemented. We describes its overall architecture, the data structures we implemented as well as the strategies we designed for keeping the size of the log manageable. In Chapter 4 we describe the test cases we subjected *PREDSL* to for evaluation, and present and discuss the results of this evaluation. In Chapter 5, we provide a summary of our work and draw a conclusion. Finally, in Chapter 6 we describe which areas we did not cover as part of this thesis, and into which future research may yield further results.

1.1 Motivation

Append-only logs have been a topic of interest in multiple fields of computer science research for at least 40 years. Such a linear data structure, where the only permitted type of modification is appending additional data at its end, seems primitive and simple at first. However, this simplicity makes append-only logs an universal foundation for almost all uses cases where data is or can be modeled as a sequence of updates.

Early research has been using append-only logs as a basis for file systems, accelerating write-heavy workloads which could simply append one block of data to the other in a linear fashion. While these kinds of file systems have been replaced by further innovation, we still see log-based components being used in modern data storage in the form of file system journals or database write-ahead logs: When database records are being modified, the modifications are recorded in a log, which is persisted to disk. For example, if the database system were to crash, a consistent database state can be recovered by "replaying" the log entries one after another.

In recent years, append-only logs have been experiencing a revival in the field of data replication between network peers. Using an append-only log as the foundation for replicated data structures allows us to describe the state of each peer by a single quantity: The latest

log entry it has successfully received. This makes it almost trivial for a network to determine which data needs to be replicated to which peer.

Recent iterations of this design paradigm have begun to combine append-only logs with cryptography principles: By signing log entries with cryptographic keys, authorship of a log entry can be verified. Similarly, by embedding cryptographic hashes computed from other entries in a log, a total temporal order among all entries in a log can not only be claimed by the author of the log, but can also be independently verified by all consumers of such a log.

However, ensuring these properties come at a price: Trivial implementations require a network participant to replicate an entire log in order to verify their security guarantees. Recent research has explored how some security properties can be maintained even when only small parts of a log are replicated.

1.2 Challenges

An append-only log can be used as an abstract state machine where each log entry describes a state transition. By relying on the previously discussed properties of distributed append-only logs, we can utilize these logs to create a *replicated state machine*. An observer of this log can simply consume the individual log entries (describing state transitions) in the correct order to reconstruct the state machine's state.

We can utilize this state machine to describe (arbitrary) mutable data structures, which are being modified by state transitions. Let us, for example consider the simple example of a list, which supports three modification operations:

- **insert**$(\mathbf{l}, \mathbf{i}, \mathbf{v})$ inserts item v at position i in the list l. All entries in l at indices $>= i$ have their index incremented by 1 as a consequence.
- **delete**$(\mathbf{l}, \mathbf{i})$ removes the item at position i in the list l. All entries in l at indices $= i$ have their index decremented by 1.
- **replace**$(\mathbf{l}, \mathbf{i}, \mathbf{v})$ replaces the item at position i in the list l with a new item v. Other entries remain unaffected by this operation.

We can use these three primitives to construct a log sequence describing the modifications performed on such a list. Take for example Figure 1.1, where we first add 3 integer values to the list, then delete one value, and finally replace one entry with another value.

While this simple example already suffices to show how a sequence of state transitions in an append-only log can be used to describe a mutable data structure, it requires a consumer to retrieve and consume the entire log, which grows indefinitely as the data structure is modified by the producer.

This can quickly lead to trouble when consumers with a constrained memory size attempt to consume this log. Such a memory-constrained consumer could be accommodated if only a part of the log needed to be consumed in order to reconstruct the current state of the data structure.

A naïve approach to accommodate such partial replication would be to simply eliminate the log entries corresponding to items that have been replaced or removed from the list, and

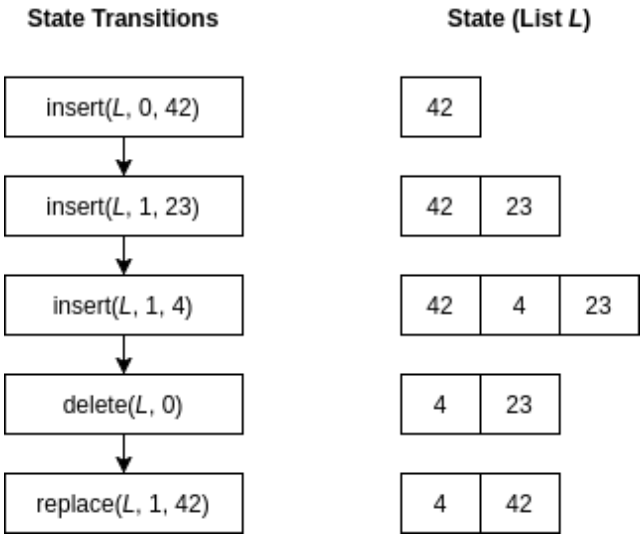

Figure 1.1: Left: A log of transitions of a state machine representing a mutable list. Right: The state of the list after applying the corresponding state transition to the left.

only maintain those entries referring to items still present in the current state. However, we can quickly see that this approach introduces a problem: The transitions in this example are focused around list index numbers, which are required for all three types of modifications. When we begin eliminating older log entries of removed items, we lose the context required for these index based operations:

Let us assume we eliminate the first and fourth log entries, which inserted and deleted the value 42 at index 0, respectively. Consuming the second and third entry would already introduce the first issue: Our list now starts at index 1 rather than 0. While this is usually not supported in most list implementations, let us for the moment assume we are dealing with an implementation capable of supporting arbitrary list starts. Even so, when we consume the last log entry, we run into another issue: Instead of replacing the value 23 with 42, as shown in Figure 1.1, we would now replace the preceding value 4 with 42 – remember that our modified list starts at 1 –, and end up with a different state than we would when consuming the entire log; we have lost the change of context caused by the *delete* operations, and our list indexes are still off by one.

This simple example shows that it is not trivial to define a state machine which can tolerate removal of "stale" transitions without changing the represented current state. Suitable candidates must be found which can tolerate certain kinds of elimination in order to reduce the set of log entries required to reconstruct the current state. Since the exact definition of the state machine depends on the type of data structure represented by it, and the set of operations required to modify this data structure, such candidates should be found for multiple types of data structures – starting with simple lists, but progressing to trees or graphs and potentially other data structure.

1.3 Goals

In this thesis we wanted to pick up on the recent developments in partially replicated logs and apply them to some of the basic data structures used every day in computer science and engineering, exploring strategies for describing these data structures in a way suitable for partially replicated logs.

We aimed to design a framework within which these data structures can be implemented in a way such that all modifications to these data structures are represented as entries appended to a log.

With these data structure implementations, we wanted to understand how data is distributed on the log as time progresses, i.e. which portions of the log predominantly contain entries required to represent the current state, and which portions contain mostly stale entries. Based on these insights we planned to design strategies to optimize said distribution of log entries so that older portions of the log can be truncated without losing still relevant information.

Finally, we planned to put these strategies to the test in different scenarios against the data structures we implemented, and evaluate how the strategies performed.

2
Related Work

In this chapter we give an introduction into append-only logs. We first highlight some historical uses of append-only logs, followed by recent developments.

2.1 File Systems

One of the origins of append-only logs lies in early attempts to increase performance of file systems in certain use cases.

Early hard disks executed both read and write operations orders of magnitudes slower than nowadays. Due to the "spinning disk" nature of these storage devices, contiguous data can be accessed much faster because fewer and shorter seek operations (moving the disk arm and waiting for the correct sector to pass by) are required. Log based file systems utilized this characteristic by keeping write accesses to files cached in volatile memory, and then committing them asynchronously in one single, sequential write operation.

This causes "a log structured file system [to achieve] *temporal locality*; information that is created or modified at the same time will be grouped closely on disk." In contrast, "[a] traditional file system achieves *logical locality* by assuming certain access patterns [such as] sequential reading of files" [14]. While this *temporal locality* of append-only file systems allows for more performant write operations, the reading performance suffers from it: Whenever a file is modified, the modified sections are written to an entirely new location, possibly far away from their "neighbors". This fragmentation makes reading and reconstructing an entire file more and more time-intensive, the more it is being modified. To manage this reading overhead, these log-based file systems utilize volatile memory caches; files that had recently been read from or written to would be cached in memory and could be accessed faster. In additions, these file systems usually come with their own, optimized on-disk index structures to increase read performance.

2.1.1 Write-Once Log Service

In 1987, Finlayson and Cheriton [4] introduced *Clio*, a log file service, in which files, once created, can only be read from and appended to; deleting or rewriting files is not possible.

Clio was designed with write-once optical disks in mind.

Back then, these predecessors of writable CDs were more cost-effective than magnetic hard disks. The authors argue that "regular" file systems can not make use of this cheaper storage, since they assume that data – especially metadata such as inodes – can be rewritten.

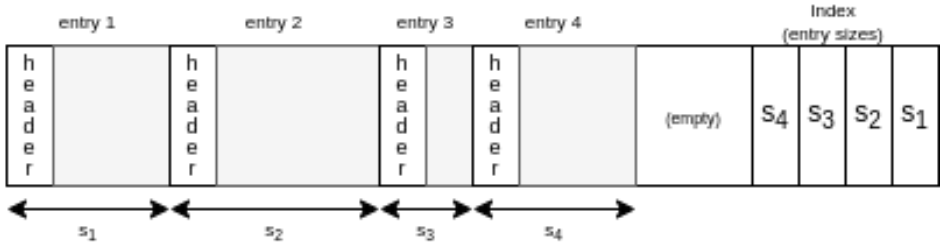

Figure 2.1: "Layout of log entries within a disk block". Colorized digital reproduction **based on an original figure from Finlayson and Cheriton [4]**.

Clio is implemented as an extension to a network storage server, where it exposes log files as a special type of files which can only be appended to. When an entry is appended to a log file, it is stored on disk with a small header describing which log file it belongs to, as well as a timestamp. Log entries are collected and stored contiguously within a disk block; this layout is shown in Figure 2.1.

Note that Clio does not feature the typical *inode* blocks utilized by regular file systems for describing where on the disk the blocks belonging to a file are stored. Since there can be multiple log entries in a block, a block generally does not belong to a single file; instead it can be shared by the entries of multiple logs. Instead of inodes, Clio uses a N-ary bitmap tree called *entrymap*, which is implemented internally as yet another log file stored in the same manner: An entrymap log entry is written once every N blocks. For every file (except the entrymap itself), the entrymap entry contains a bitmap describing which of the N preceding blocks contains at least one entry of this log. These entries occurring once every N blocks form the leafs of the entrymap tree. For every N leaves that have been written to the log, a 2nd level inner tree node is added once every N^2 blocks, a 3rd level node every N^3 nodes, and so on.

2.1.2 Log-Structured File System

In 1992, Rosenblum and Ousterhout [14] introduced *Sprite LFS*, a *log-structured file system*: "A log-structured file system writes all modifications to disk sequentially in a log-like structure, thereby speeding up both file writing and crash recovery." Unlike Clio, *Sprite* is a full-featured file system not limited to log files. Neither does it prohibit deletion or rewriting of files, nor does it utilize write-once storage. Instead, conventional magnetic disk storage is used.

Similar to conventional file systems, Sprite files are not mapped directly between the file system hierarchy and the physical location on disk: Instead an *inode* is created per file. This data structure maintains file metadata along with the location of the actual data blocks on disk.

In conventional file systems, once an inode has been allocated, it's location never changes. While data blocks may be rearranged when files are modified, their new location is always written back to the same inode, which is always identified by the same unique number. From this number, the fixed position of the inode on disk can be computed.

In contrast, *Sprite LFS* does not rewrite inodes to the same location. Instead, they are appended to the on-disk log structure along with the changed data blocks. In order to find these constantly moving inodes, an additional data structure called the *inode map* is utilized. This inode map maintains the location of the most recent inode for each file. And even this additional data structure, which is also divided into blocks, is appended to the log. Finally every once in a while, *Sprite* writes out a *checkpoint* to this dedicated region on disk, containing among other things the location of all inode map blocks. This *checkpoint region* is the only component of *Sprite* that is repeatedly written to a fixed location.

These additional levels of indirection would usually incur a decent overhead when reading a file from disk. However, the inode map is cached entirely in memory, so that this overhead is only incurred once after mounting the file system. Combined with the memory caching of recently accessed files, the authors show that under most usage patterns, *Sprite LFS* achieves read performance only insubstantially worse than that of conventional file systems, while exhibiting a notably increased write performance than other file systems.

This performance gain does however come with a price: For one thing, in order to maintain performance, a lot more volatile memory is required than with conventional systems. For another thing, the write performance is only improved, if enough contiguous unused space is available at the end of the log. This is the case on a freshly formatted device, but due to the appending nature of write operations, the disk would be filled a lot quicker than with conventional file systems.

When the end of the disk is reached, the log wraps around and starts over at the beginning. *Sprite LFS* recovers space to be reused from the blocks previously allocated to files that have been deleted, or blocks than have been obsoleted by a newer version.

When not enough contiguous free storage is available, *Sprite* employs a mechanism named "threading": The log is written to recovered segments and is "woven" in between already allocated segments. This requires the long, sequential write operations to be split into multiple smaller operations, completely nullifying the performance benefits.

To combat the need for threading, *Sprite* implements a "copy-and-compact" garbage collector: The contents of multiple contiguous partially-populated segments are compacted into a sequence of contiguous blocks, which is then appended to the end of the log. After this, the involved segments don't contain any live data anymore, and can be reused as a long, contiguous range of free storage. This garbage collection ensures that *Sprite* can maintain its write performance. However, it already requires some amount of contiguous free space at the of the log, where it can append the compacted data. When the file system utilization is close to its capacity limit, there is not enough free space to work with, and *Sprite* can not uphold its write performance. These two mechanisms, threading and copy-and-compact are shown in Figure 2.2

Thus, if the write performance gains are to be utilized, more storage needs to be provisioned than is actually expected to be used, in order to provide enough reserve for the

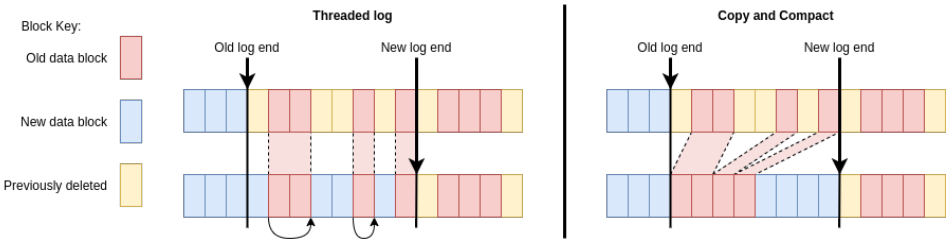

Figure 2.2: Visualization of *Sprite LFS*' mechanisms for storage reclamation: *Threading* (left) is used to continue writing the log when no contiguous free storage of sufficient size is available; the log is "woven" in between older, but still active blocks. *Copy-and-Compact* garbage collection is used to reclaim larger ranges of contiguous storage; partially used segments are compacted and re-appended to the end of the log. Colorized digital reproduction **based on an original figure from Rosenblum and Ousterhout [14]**.

garbage collector to work with.

2.2 Single-Writer Data Replication

Append-only logs are utilized heavily in distributed systems where data produced by a single producer needs to be replicated to one or more consumers. When the producer performs any modification on the data it produces, it emits a log entry. This log is then replicated to all consumers of the log through an appropriate mode of transport. Due to the linear fashion of the log, consumers receive the new data in order and can synchronize their local state to that of the producer simply by consuming one log entry after the other in the correct order.

This concept can be applied to a wide variety of use cases. Replication solutions relying on logs can vary e.g. in the way the log is transferred, the way the total order of log entries is ensured, and the way the consuming application interprets the data written to the log by the producers.

In the following, we will highlight some data replication solutions relying on append-only logs.

2.2.1 Consensus Algorithms

Raft is a recently proposed "consensus algorithm for managing a replicated log" [12]. It was introduced in 2014 by Ongaro and Ousterhout as an alternative to the *Paxos* [7] consensus algorithm, aiming to be less complicated and easier to understand.

When a Raft cluster – consisting of a predetermined number of member nodes n – starts up, the cluster members will first elect a leader node. Leadership phases are called "terms". A term is identified by a monotonically increasing number, begins with an election and lasts until the next election. When a new term starts, each node waits for a short, but random amount of time, and if has not yet heard from another node by then, it will announce its "candidacy" for the current term's leadership to all other members. If a node receives other nodes' candidacies before broadcasting its own candidacy it will vote for the first node whose candidacy has been received first. A node becomes the elected leader for the current term

once it has received a simple majority vote, or $\lfloor \frac{n}{2} + 1 \rfloor$ votes. In case of a stalemate, the voting cycle simply starts again with newly generated random wait times. Once a node has been elected, it has to broadcast keep-alive packets on a regular basis to let the other cluster members know that it is still available in order to maintain its leader status.

If the leader node is not heard from for a certain amount of time, the remaining nodes will initiate a new term and re-elect a new leader. Raft can tolerate non-Byzantine failure of up to $\lceil \frac{n}{2} - 1 \rceil$ members. If half or more nodes fail, the cluster will not be able to elect a new leader, because it can not agree on a majority vote.

The members of a Raft cluster are maintaining an append-only log. Each node keeps a local copy of this log, but at any time the only node permitted to introduce new entries to the log is the currently elected leader. When a new entry is added to the log, the leader will only *tentatively* accept this new entry. It will then notify the other members of the cluster of this new log entry, incrementing a serial number attached to the log entry. When another node receives this new entry, at it meets certain acceptance criteria (e.g. having the correct term number), it confirms the acceptance of the entry back to the leader. The leader only permanently commits the new entry to the log if it has been confirmed by the majority of members. This mechanism works similar to the leader election mechanism and assures that all nodes will eventually agree on one global state of the log.

Raft has become a popular foundation for consensus systems and distributed database systems. For example, it forms the basis of the *etcd*[1] distributed key-value store, which in turn is one of the core components of the *Kubernetes*[2] container orchestration platform.

Nevertheless, Raft comes with one great weakness: All nodes are trusted to behave correctly. During leader election, a rogue node could easily become the leader simply by broadcasting a candidacy with a higher term number than that of the other candidacies. Once elected, full trust is placed in the leader node to replicate the same log state to all cluster members. In particular, a Byzantine leader could introduce different log entries to each follower's local log, creating an inconsistent state. Raft is able to return to a consistent state, however it can not be predicted, what this state will be, as this depends on which node wins the next leader election.

Another noteworthy property of Raft is its approach on the *single writer* property of the log: While Raft ensures that at every given time, only the current leader appends to the log, over time multiple nodes can have been leaders, and the log will have been written by multiple nodes, though never simultaneously. This approach (which we will call *single writer at a time*) is different from e.g. *Secure Scuttlebutt* (Section 2.2.2), where each log is only ever written by the same writer (hereafter called *single writer at all times*.

2.2.2 Peer-to-Peer Communication

Secure Scuttlebutt is a peer-to-peer communications and network protocol first released in 2014. It was first described in academia by Tarr et al. in 2019 [16]. Secure Scuttlebutt (*SSB* for short) is built around cryptographically signed and potentially encrypted single-writer

[1] https://etcd.io/
[2] https://kubernetes.io/

append-only logs, which can be subscribed to by other network participants.

A log in SSB is uniquely identified by a Ed25519 [1] public key. Every log entry must be signed with the corresponding private key in order to be considered valid. Furthermore, each log in SSB forms a *hash chain* of its log entries: The signed portion of each entry in a log – except the first one – contains the SHA256 hash value of the previous log entry, cryptographically proofing the relative order between two adjacent log entries.

A log entry in SSB is a JSON[3] document. In addition to the already mentioned predecessor hash, this document contains the Ed25519 public key, a sequence number (its position in the log), a timestamp, and finally the entry's content, which is yet another JSON document. The log entry JSON document is serialized into a canonical string representation, which is then signed by the private key of the log author, and the resulting signature is appended to the entry. Finally, this signed entry is then appended to the log. This structure is visualized in Figure 2.3. The requirement for signed entries causes SSB to be a *single writer at all times* system, since only the holder of this key can ever append to the log.

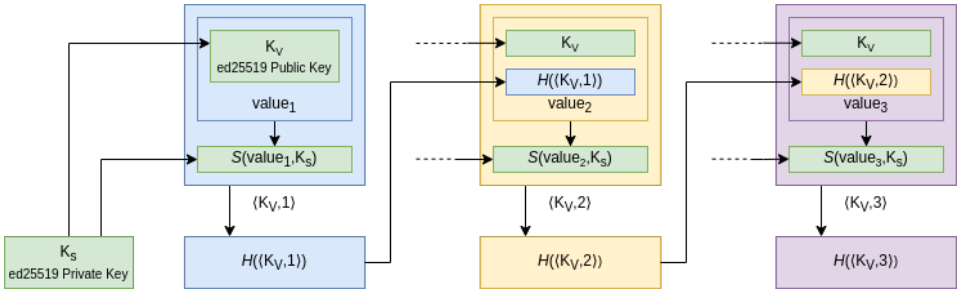

Figure 2.3: The structure of a SSB log. The root of trust is the Ed25519 private key K_S, from which the corresponding public key K_V is produced. The i-th log entry is identified by the public key and its sequence number $\langle K_V, i\rangle$, or its unique hash, $\mathcal{H}(\langle K_V, i\rangle)$. The content of a log entry consists of a JSON document value$_i$, which is signed by the corresponding private key to form the cryptographic signature $\mathcal{S}(\text{value}_i, K_S)$. Except for the first log entry, each entry's value object contains the SHA256 hash of its predecessor $\mathcal{H}(\langle K_V, i-1\rangle)$.

A log entry in SSB can be uniquely identified in two ways: Either by the SHA256 hash value of the entry (including the signature), or by a combination of the Ed25519 public key (identifying the log) and the entry's sequence number (identifying its position in the log).

Both approaches have in common that they can be used as names for addressing a log or log entry by content, rather than location: A network participant subscribes to a log by specifying its public key. Similarly, when participants are exchanging or comparing log entries, they refer to them by one of these two naming schemes. It is up to the network to actually find and retrieve the requested data. This approach to networking is called *Content Centric Networking* or *Information Centric Networking* and was initially introduced by Jacobson et al. [5] in 2009. Since then it has been an active topic of research, with multiple research projects forming around it.

The network participants, called *relays* in SSB, maintain one or more peer-to-peer con-

[3] JavaScript Object Notation, https://www.json.org/json-en.html

nections to other peers. The relays in a network implement a gossip protocol where each relay regularly notifies its immediate neighbors about the logs it has subscribed to, thereby signaling both an interest in updates to these logs, as well as the number of log entries already present locally. When a relay notices that it has new entries for a log that one of its peers is following as well, it will replicate these new log entries to said peer.

This causes Secure Scuttlebutt to act as a kind of a social network; initially, each relay can only see the logs known by itself and by its immediate peers. The basic architecture of SSB only permits this single-hop replication. However, there is a lot of room for building more sophisticated solutions on top of this basic network architecture. One of the most important improvements is the "friends plugin", which "scans the relay's log for specific messages that indicate which other identities the author *follows*". By repeating this process for the *follow* messages retrieved from its peers, a network participant can choose to expand its social circle hop by hop.

2.2.3 File Synchronization

Dat, introduced in 2017 by Ogden et al. [11], is another protocol for peer-to-peer data exchange. Unlike SSB, which is built around a log of small entries, Dat focuses on "syncing folders of data, even if they are large or changing constantly".

Dat consists of multiple modules, out of which we will only look at *Hypercore*, which consists of Dat's core components "storage, content integrity and networking protocols". The storage primitives of Hypercore are "binary append-only streams" called *registers*. Similar to a SSB log, a Hypercore register is identified by an Ed25519 keypair, which is used to sign content added to the register. This data is organized block-wise in *chunks*. In addition a Hypercore register also maintains a Merkle tree of the chunk hashes.

This Merkle tree is maintained in a way so that only new nodes are ever added to the tree, nodes are never removed or modified. Nodes are never appended to leafs of the tree; instead, new independent subtrees are built "bottom-up", starting from the leaves, and adding new inner nodes, each of which joins two equal-sized subtrees to a new, larger subtree. However, since a Merkle tree (or a tree in general), by definition, always has to have a single root node, a "virtual" root node is maintained, which simply forms a hash over all subtree roots. In addition, this virtual root node is signed by the authors cryptographic key, implicitly signing the entire tree and thereby all data chunks. This procedure is visualized in Figure 2.4.

The node numbers used in Figure 2.4 are a "binary in-order interval numbering" scheme used by Hypercore to deterministically and predictably identify each node in the tree. This numbering scheme causes every node in this tree to always only grow to the right; node 0 will always be the left-most leaf.

It is worth noting that enforcing this structure of the Merkle tree implies that new data chunks can only ever be appended to the end of the data stream. Combined with the required signature in the virtual root node, this turns Hypercore into a single-writer append-only log similar to Secure Scuttlebutt, even though its structure is entirely different from that of SSB.

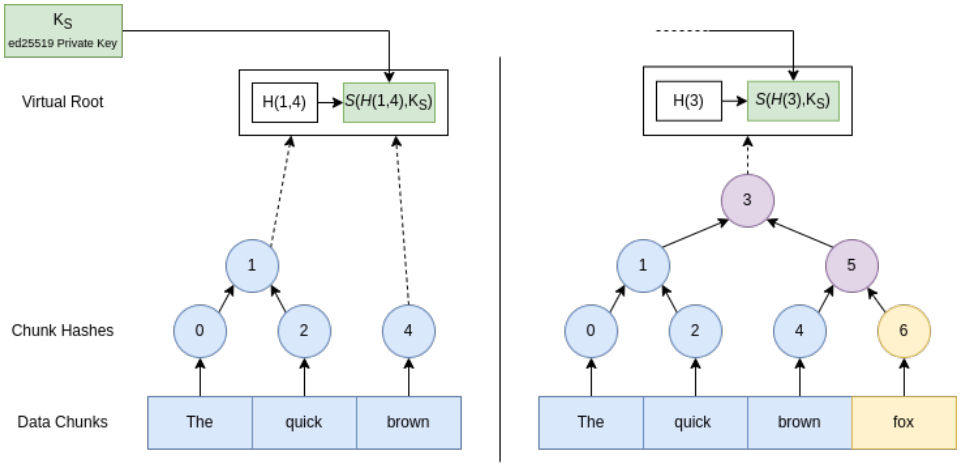

Figure 2.4: On the left side is a Hypercore register containing 3 data chunks. Each chunk is hashed, forming the leaf nodes 0, 2 and 4 of the Merkle tree. Inner nodes are inserted wherever their subtrees form *perfect* binary trees. This is only the case for node 1 in this case. The virtual root node contains the hash over all subtree roots, 1 and 4 in this case. On the right side, we see a new chunk of data being added. A new leaf node, 6, containing the hash of the new chunk is added, forming a new subtree. Afterwards, additional inner nodes are added where a perfect binary tree is formed, node 5 and 3 in this example. Finally, the new virtual root is created using the hash of node 3, and signed using the author's private key.

While this Merkle tree based approach appears to incur a storage and computation overhead, when compared to SSB, we can quickly see an advantage when we look at Hypercore's replication protocol:

Let us assume a consumer called Alice wants to subscribe to a Hypercore register produced by Bob, and that Alice and Bob are directly connected network peers.

To subscribe to the register, Alice and Bob perform a subscription handshake, where Alice notifies Bob of its subscription. Bob will respond with the number of data chunks it is currently aware of. After this handshake, it is up to Alice to decide which content chunks she wants to retrieve. Bob responds to each data chunk requested by Alice with said chuck, as well as all $\mathcal{O}(\log(n))$ nodes of the Merkle tree required to verify the hash chain from the root to this chunk.

This approach to log replication is noteworthy because it permits Alice to *sparsely* replicate the log, while still being able to cryptographically verify the log authorship and a data chunk's position in the log.

2.3 Classification of Append-Only Logs

In an unpublished manuscript from 2020, Meyer [8] introduces a schema for the classification of replicated append-only logs. The author introduces three general classes of logs, which primarily differ in the extent in which trust and consistency guarantees are provided by the log service, rather than being implicitly assumed.

The first class, named *Naïve Logs*, assumes a network of mutually trusting peers. In this class, the only measure that is applied for providing consistency of the log is a monotonically

increasing sequence number attached to each log entry. This mutual trust implies that network participants will only ever append to logs they are authorized to append to, and will do so using a correct sequence number; neither the single-writer, nor the append-only property of such a log are cryptographically enforced.

One example of such a naïve log implementation is *Raft* (Section 2.2.1), where all cluster members trust that only the elected leader appends to the (single) log, and that this is done using the correct sequence numbers. The example of Raft also shows us that naïve logs can be sufficient to maintain a consistent replicated log.

The second class introduced by Meyer is called *Signed Naïve Logs*. This class differs from the first class in that a cryptographic signature is attached to each log entry, and all entries in a log must be signed by the same key. By verifying these signatures and ensuring they were all created by the same key, the single-writer property can be enforced, since only the intended writer holds the key required to sign a log's entries. On the other hand, the append-only property is not enforced. For example, a producer can create a signed log entry with a sequence number of 2, and then only afterwards create and publish the entry 1. Assuming that single log entries are replicated individually, such a behavior can not be differentiated from replication delays.

The third class of replicated logs is called *Signed Hash Chains*. In addition to the signatures introduced in the previous class, each log entry is augmented with a hash value of its predecessor, computed using a cryptographically secure hash function. Consumers can verify that two adjacent log entries have been appended to the log in order by computing the hash value of the first entry and comparing it to the hash embedded in the following log entry. If the two values match, the consumer has confirmed that these log entries have been appended in order, since the second entry can only have been produced at a time where the hash of the first entry was already known.

An example of signed hash chains is *Secure Scuttlebutt* (Section 2.2.2). See esp. Figure 2.3, where we have visualized the structure of a SSB log and how the signed hash chain is formed.

Note that none of the measures introduced in these classes can prevent the practice of *forking* a log. A log is considered forked, if its author has published multiple log entries sharing a sequence number, but differing in content. While forks can not be prevented, they can be detected, though the effort required to do so differs between these three classes. See Figure 2.5 for examples of forked logs of each class and how they can be detected.

2.4 Security Properties for Sparse Replication

In another unpublished manuscript from 2020, Buldas et al. [9] analyzed how the following security properties can be maintained when a log is only partially replicated:

Authenticity "Validate authorship of log entries and/or the whole log".

Integrity (Immutability and Inclusion) "Once added, a log entry cannot be modified or removed, an for a given log entry, it can be efficiently decided whether it is part of a log or not."

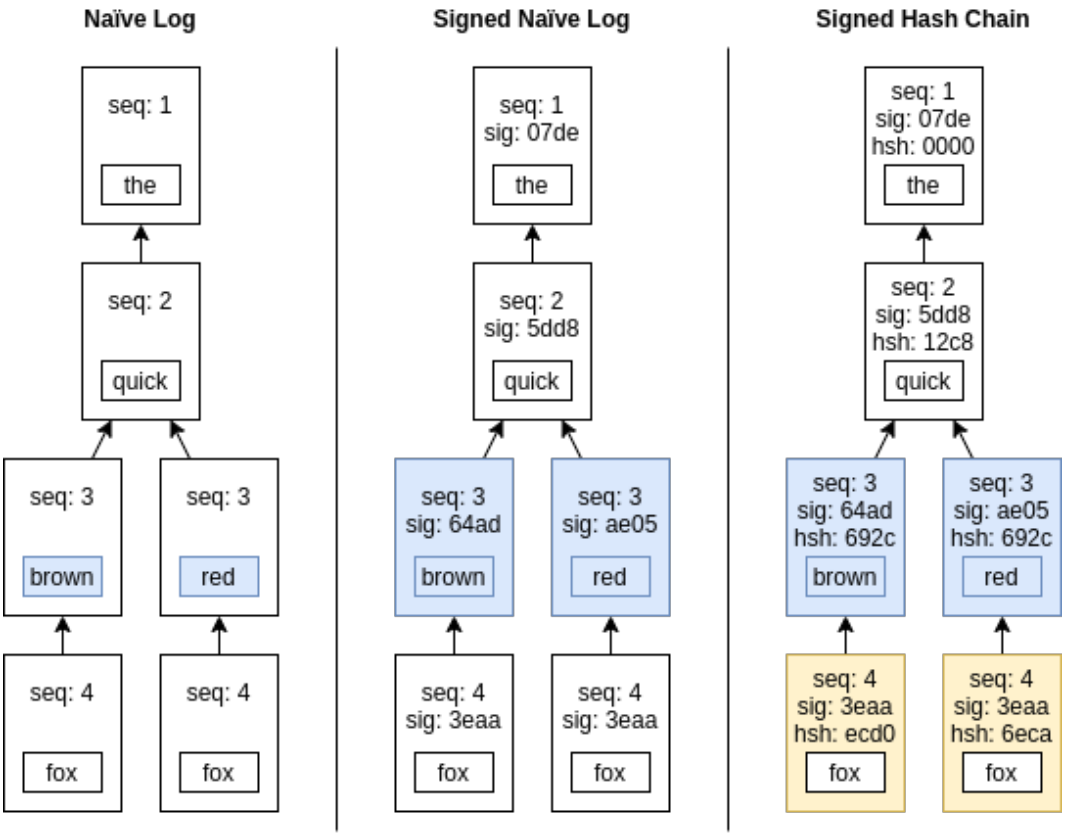

Figure 2.5: Forked logs and the log entries the fork can be detected in. With Naïve Logs, the fork can only be detected by comparing the contents of conflicting log entries. With Signed Naïve logs, the signature of the conflicting entries will not match, therefore it is sufficient to compare the metadata of conflicting entries. With signed hash chains, the hashes and signatures of all following entries will not match as well, simplifying detection of a fork even further.

Coherence "Given two log instances, decide whether one is a sublog of the other, and in the negative case find the longest common prefix, i.e. the point of divergence."

The authors examine how existing log service implementations, namely Secure Scuttlebutt (Section 2.2.2) and Hypercore (Section 2.2.3), would fulfil these properties under partial replication. The authors assert that the signed Merkle tree used by Hypercore already enforces the first two properties, and propose algorithms for verifying all three properties.

Secure Scuttlebutt, on the other hand, uses a linear signed hash chain. This structure prevents an "efficient" (not specified any closer) verification of the last two properties, since due to the linear nature of the chain, "the whole chain [has to be] fetch[ed] and traverse[d] [...] in the worst case".

In order to improve the efficiency of verifying these properties in SSB, an augmentation to the log structure of Secure Scuttlebutt is proposed, which is based on a binary linking scheme first described by Buldas et al. in 1998 [2]. The proposed solution adds additional predecessor hashes to the headers of some log entries. Rather than only containing the hash

of the immediately preceding log entry, the hashes of some select entries further "up" in the log are added. The distance to these additional predecessors increases exponentially. By traversing back through those additional links, where present, the number of log entries required for traversing to the log root can be reduced to $\mathcal{O}(\log(n))$, while also incurring a $\mathcal{O}(\log(n))$ overhead in memory and storage space required for the additional hashes.

2.5 Complex Data Structures in Conflict-Free Replicated Data Types

The concept of *Conflict-Free Replicated Data Types* (*CRDTs*), introduced by Shapiro et al. in 2011 [15], is an approach to real-time collaborative editing of documents by multiple users over the internet, providing strong eventual consistency. In the last few years, there has been further research into combining multiple such CRDTs in order to represent and manage more complex, composed data structures such as JSON documents.

YATA [10] is a CRDT framework published in 2016 by Nicolaescu et al.. It features the representation of JSON and XML documents as nested CRDTs. "YATA represents linear data (e.g. text) as a doubly linked list" [10]. When a new item is inserted into such a list, identifiers of its immediate predecessor and successor are provided in order to insert the item at the correct position.

DSON [13], another CRDT framework capable of representing JSON-like data structures, introduced by Rinberg et al. in early 2022, follows a different approach for maintaining the order of items in a linear data structure: Rather than using integers for positioning elements in an array, the domain of real numbers is used. "An insertion of an element between an element at position p_1 and one at p_2, inserts an element at position $(p_1 + p_2)/2$. The array is the sequence of values sorted by the corresponding position in ascending order." [13, p. 1058]

3
Design & Implementation

In this chapter we describe *PREDSL* (short for "**Pr**uned, **Re**plicated **D**ata **S**tructure **L**ogs"), the framework we designed and implemented during the course of this thesis, which provides management and abstraction layers to both log and data structure implementations. Most importantly our framework describes the interface between logs and data structures serialized on these logs. The overall architecture of *PREDSL* is visualized in Figure 3.1. *PREDSL* was implemented using the Python[4] programming language and is publicly available on gitlab.com[5].

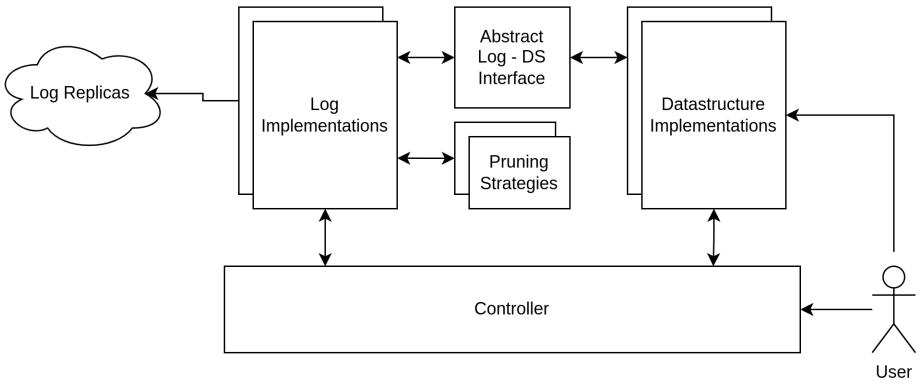

Figure 3.1: The overall architecture of the framework used for implementing the log-serialized data structures.

Within *PREDSL*, multiple implementations of both logs and data structures are supported, all adhering to a common interface. Different log implementations differ in the way their log is stored or replicated. For example, a log instance could be implemented which performs replication using the *Secure Scuttlebutt* protocol (see Section 2.2.2). During this thesis only a single log instance has been implemented, in which the log is only stored in memory, and replication happens through shared memory. While this ephemeral log im-

[4] https://www.python.org/
[5] https://gitlab.com/s3lph/predsl

plementation may not serve any real-world purpose, it was sufficient for implementing and evaluating different data structures and pruning strategies.

In *PREDSL*, a single log can contain multiple data structures, and these data structures may be arbitrarily nested within each other. By doing so, we can encode e.g. entire JSON[6]-like documents on a single log, similar to *DSON* by Rinberg et al. [13] (see Section 2.5).

In the remainder of this chapter, the individual components of *PREDSL* are described in detail: In Section 3.1, we describe the common interface between logs and data structures. In Section 3.2, we describe our approach to log pruning and the different pruning strategies we employed. In Section 3.3 we describe the data structures we implemented in *PREDSL*, along with the set of low-level modification operations of each data structure, and how they are serialized to a generic log.

3.1 Log – Data Structure Interface

As shown in Figure 3.1, the log and data structure implementations interact with each other via an abstract interface provided by *PREDSL*. Thanks to this common interface, a new data structure can be implemented in a generic way, without targeting a specific log implementation. The same holds true for the other way around. In this section, we go into detail about this interface and how logs and data structures interact with each other.

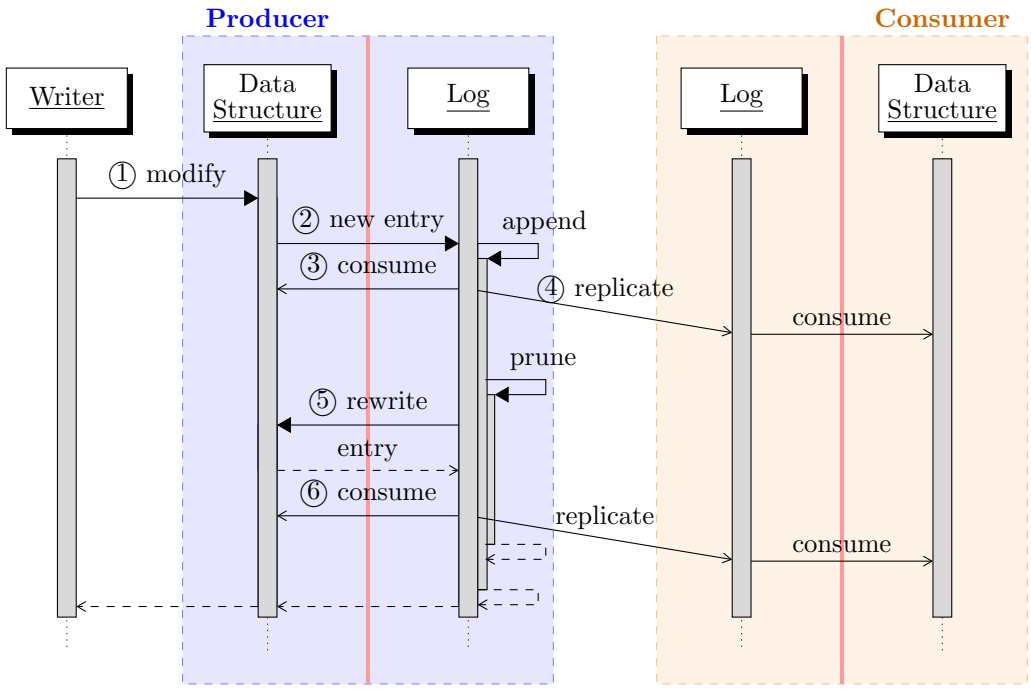

Figure 3.2: Sequence diagram of the interaction between data structures and logs. In this example, the writer performs a single modification on the data structure, which is replicated to a single consumer. The red lines between the logs and data structures represent the *Log - Data Structure Interface* discussed in Section 3.1.

[6] https://www.json.org/

First, let us consider the interface on the *log producer side*, depicted on the left-hand side in Figure 3.2: On the producer side, the data structures are writable by external writers, as opposed to the consumer side, where they can only consume updates from replicated log entries. In the following we will be using circled numbers ⓪ referring to the corresponding markers in Figure 3.2 for better orientation.

When such an external writer performs a modifying operation ① on a data structure, the modification is not applied directly to the data structure itself. Instead, the data structure implementation encodes this modification in a new log entry, which it then appends to its backing log ②. Only when the log has successfully persisted the new entry, will it be passed back to the data structure for consumption ③, meaning that the modification encoded in the log entry is finally applied to the data structures internal state. At this time, the new log entry is also replicated to read-only replicas of the log ④, where it is in turn consumed by the read-only copies of the data structure.

After appending each log entry, the log implementation will decide whether the log can and should be rewritten and/or pruned. Log pruning is discussed in detail in Section 3.2. This decision depends on the rewriting strategy chosen by the log. The rewriting strategies we implemented are discussed in Section 3.4. If it is decided that the oldest entry should be rewritten at the log head, the log looks up the data structure referred to by this oldest entry, and requests from it to generate a new log entry to replace the old one ⑤. Again, this entry is appended to the log, consumed ⑥ and replicated. Depending on the rewriting strategy, this process may be repeated.

Finally, after the log has finished rewriting older entries, the old entries that have been rewritten are pruned from the log, and control is returned to the external log writer.

On the consumer side, the interface between log and data structure is a subset of the producer-side interface: The only interaction on this side is the consumption of log entries replicated to the log.

This architecture has the advantage that producer and consumer side data structure implementations can share a large portion of their code. The append-then-consume approach on the producer-side furthermore ensures that the producer log remains the "single source of truth" of the data structures' state, because no modifications are applied to the producer-side data structure before they have been committed to the log.

3.2 Log Pruning

As a log based data structure is modified over time, the log continues to grow with each single modification. However, storage space is limited, and so we need to find a way to reduce the space required.

Our main requirement is to be able, at any point in time, to reconstruct the current state of the data structure at that point in time. Fulfilling this requirement enables fast bootstrapping of new log consumers: New consumers can immediately reconstruct the data structure, and from that point in time subscribe to future modifications.

In order to keep the size of the log manageable, we maintain a *sliding window* over the log, starting at the oldest log entry required for reconstructing the data structure, and ending

at the log head. When an older log entry becomes obsolete, the start of the sliding window progresses to the next entry that is still required. We then discard the log entries that are no longer within the sliding window, in a process called *log pruning*. This process is visualized in Figure 3.3.

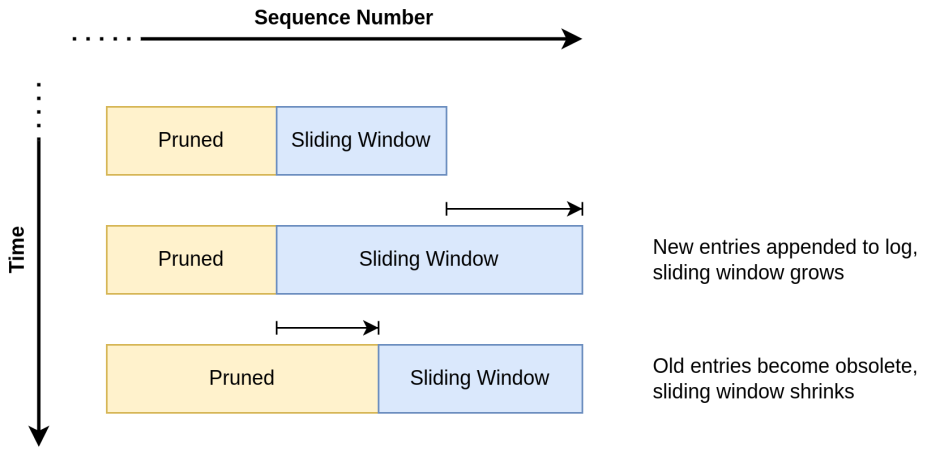

Figure 3.3: The sliding window over an ever growing log. The sliding window grows when new log entries are appended and shrinks when old log entries become obsolete. Obsolete entries are pruned from the log and can be deleted; only the entries within the sliding window need to be kept in storage.

For a better understanding of how pruning helps to keep the log size manageable, let us consider the example shown in Figure 3.4. This example shows how modifications to a linked list are appended to a log and consumed to produce the actual data structure. On the left-hand side, we see the log, growing downwards. Each log entry consists of a sequence number and one or more operations. These operations are:

create(v) Create a new node to be inserted into the linked list, representing the value v. The node is further on uniquely identified by the sequence number n of the log entry it was created in. From this follows that there can only be one such create operation per entry.

head(n) Set the head pointer of the list to the node with sequence number n. The head pointer defines the start of the list, pointing to the node at the first position.

link(a, b) Create a link between two nodes a and b that describes that node b follows directly after node a. If a already had a successor, the link from a to its successor is removed first. The same applies if b already had a predecessor. If the previous successor of a is not re-linked to some other position in the list within the same log entry, it is removed from the list, and must not be re-added at a later time.

These three operations are our basic building blocks for implementing a linked list. If we return to Figure 3.4, we can see how each log entry encodes an ordered sequence of one or

more of these operations. When they are applied to the data structure represented by the log, we obtain the linked lists as shown on the right-hand side of Figure 3.4:

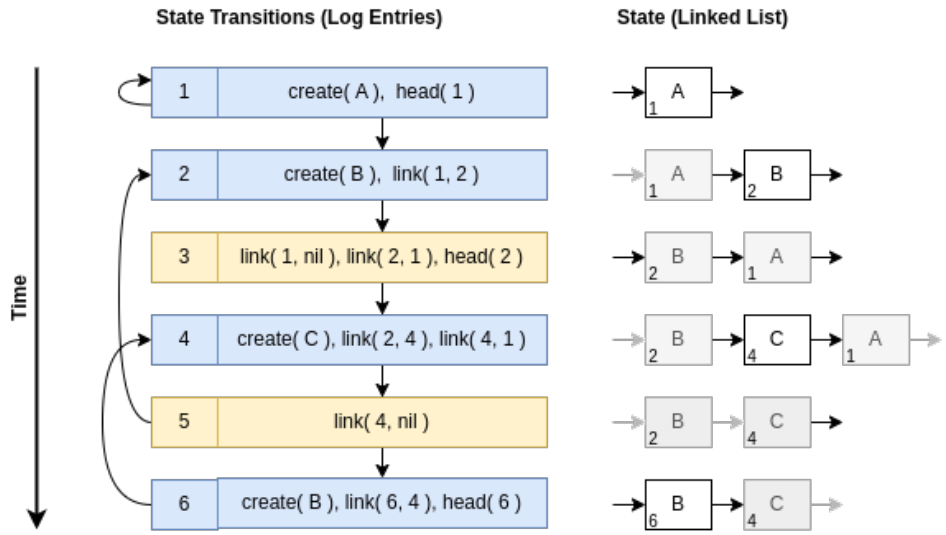

Figure 3.4: An example of a log encoding the modifications to a linked list. The left side shows the log, with each entry consisting of a sequence number and a ordered list of operations. The right side reflects the state of the linked list after applying the operations in the corresponding log entry, highlighting the actual changes at this step.

In log entries 1 and 2, we create one new node each, and append them to the list. In log entry 3, we swap the previously created nodes, without introducing a new node. In entry 4, we add a third node in between the two previous ones. In entry 5, we remove the link pointing to the the last node, implicitly removing it from the list entirely.

This is where the log can be pruned for the first time: The node that was just removed was introduced in log entry 1. Since we set the restriction that a node must not be re-added at a later time, this node will never reappear in a valid log. In conclusion, the information stored in log entry 1 will never be needed again to reconstruct the data structure, and it can safely be pruned; the start of the sliding window progresses to log entry 2. This is indicated by the arrows on the left-hand side; these are the *oldest pointers* and point to the current start of the sliding window.

However, simply waiting for older log entries to become obsolete is usually not sufficient; when or whether a log entry becomes obsolete depends on the behavior of the log producer, and there are no guarantees that this will eventually happen. This is why, in addition to this *passive pruning* approach, we *actively rewrite* old log entries.

We implemented multiple rewriting strategies, which are described in 3.4. They all have in common that they take the oldest node in a data structure, and replace it with a new node, with the same value as the old node and its current state. An example of this is shown in log entry 6 in Figure 3.4: The now oldest node 2 is replaced by a new node which takes its place at the head of the list. Since this implicitly removes node 2 from the list, the corresponding log entry can be pruned.

We can also make another observation here: We can distinguish between two different

types of log entries in this model: Some entries create a new node (highlighted in blue), while others only modify existing nodes (highlighted in yellow). When we rewrite older "blue" log entries, we also implicitly consolidate the changes introduced by "yellow" entries into the rewrite entry.

This means that when a "blue" entry can be pruned from the log, so can all "yellow" entries between this entry and the next "blue" entry. This is why, in the example above, the *oldest pointer* of log entry 6 points to entry 4, rather than 3. This is also the reason why pruning and rewriting not only helps maintain a log's size, but can even help reduce it.

3.3 Data Structures

In the following we describe the data structures implemented during the course of this thesis, and how modifications to these data structures are serialized into log entries. Internally, these log entries are represented as Python objects, and the actual serialization format depends on the specific log implementation, but in this thesis we use a JSON representation of these internal Python objects. We first cover the properties all log entries have in common:

Since *PREDSL* allows encoding of multiple data structures on a single log, log consumers need to have a way to determine which data structure a log entry refers to. For this a *data structure identifier* that is unique within a log, and the type of data structure are provided in each log entry. In addition, each log entry contains its sequence number, and, if the *oldest pointer* changes, its new value. Apart from these basic requirements, it is up to each data structure to encode their modifications, though all of our implementations follow a common convention when a new entry in a data structure is created: the entry's value is encoded in the "value" JSON key, and either refers to a nested data structure or to an immediate scalar value:

```
{
  # The unique identifier of the data structure within this log
  "duid": 1,
  # The type of data structure
  "type": "linkedlist",
  # The log entry's sequence number.
  "serial": 10,
  # Consuming this log entry permits some older entries to be
  # pruned, the sliding window progresses to [6, 10].
  "oldest": 6,
  # The value to insert.  This will create a node that is
  # uniquely identified by the log entry's sequence number (10)
  # and carries the numeric value 42.
  "value": {
    # Since nesting of data structures is supported, we need to
    # differentiate between other data structures (referred to
```

```
    # by their type and duid) and immediate scalar values.
    "type": "immediate",
    # As defined above, this is an immediate scalar value.
    "value": 42
  },
  # other keys are implementation-dependent.
  ...
}
```

If instead of an immediate value, another data structure is to be inserted, it is referred to by its type and unique identifier:

```
{
  ...,
  "value": {
    "type": "dictionary",
    "duid": 2
  },
  ...
}
```

From this convention of encoding the value of a new data structure entry follows that we can only create a single data structure entry per log entry. This approach serves two purposes: For one, we can use the sequence number of the log entry (the one a new element is created in) as a stable unique identifier without maintaining additional lookup tables. This identifier is used when modifying the element at a future point in time. For another, this makes it trivial to determine the *oldest pointer* $o(l)$ by computing the minimal identifier over all elements d in the log l's data structure D:

$$o(l) = \min\left(\{\forall d \in D : id\,(d)\}\right) \tag{3.1}$$

If a log encodes more than one data structures, the same can be achieved by computing the minimum over all data structures in the log:

$$o(l) = \min\Big(\bigcup_{D \in l} \{\forall d \in D : id\,(d)\}\Big) \tag{3.2}$$

In the remainder of this section, we will introduce the individual types of data structures implemented in the course of this thesis. For each data structure, we give a general description, and define the low-level operations implemented for modifying these data structures. We also provide examples of log entries for each data structures.

3.3.1 Arrays

Arrays are one of the (if not the most) fundamental data structures in computer science: An ordered sequence of data elements, where each element can be uniquely identified by its position in the list.

However, despite their simplicity, arrays pose a unique challenge when they are encoded by a partially replicated state machine: Assume an array implementation as proposed in Section 1.2, where operations on the array refer to its elements by their position in the array. As operations on elements that have since been deleted are pruned, we lose context required to recover the actual positions in the array, and the log producer and consumers may no longer agree on the state of the array.

Research in the field of *Conflict-free Replicated Data Types* (CRDTs, see Section 2.5) proposes different solutions for this problem, which can generally be classified in two different approaches: Either a *stable identifier* is defined from which the order in the array can be implicitly derived, or the order of elements is not defined through absolute positions, but through positions relative to other elements.

An example for stable identifiers are the identifiers used by Rinberg et al. [13] in their implementation of *DSON*: Rather than using integers for positioning elements in an array, the domain of real numbers is used. "An insertion of an element between an element at position p_1 and one at p_2, inserts an element at position $(p_1 + p_2)/2$. The array is the sequence of values sorted by the corresponding position in ascending order." [13, p. 1058]

A common way to implement the second approach of using relative positioning are linked lists, as is done in the *YATA* [10] framework. A linked list is defined by a set of nodes, directed links between these node, and a pointer to the first node in the list (the *head pointer*). The order of a linked list is obtained by iterating through all nodes, starting at the head pointer.

For our array implementation, we chose the approach of using a linked list. The operations our linked list implementation provides have already been explained in detail in the example in Section 3.2, so we will not reiterate them here.

An entry defining a node in a linked list could look like this:

```
{
  "duid": 1,
  "type": "linkedlist",
  "serial": 10,
  # Insert a scalar value into the linked list.
  "value": {
    "type": "immediate",
    "value": 23
  },
  # The new node will be the head of the list,
  # i.e. it is inserted at index 0.
  "head": 10,
  # Create a link between the new node 10 and node 5.
```

```
  # This places node 5 at index 1.
  "links": [
    [10, 5]
  ]
}
```

3.3.1.1 Sorting

In addition to the low-level operations described in Section 3.2, we also implement some higher-level operations. One of these operations is to sort the elements in a list.

Sorting is a special edge case for *PREDSL* because the low-level operations it involves only ever modify links between existing nodes in a linked list. This has an impact on log pruning, as discussed in Section 3.2, where we highlighted the differences between log entries that introduce new information and those that modify existing information.

The primitive operation used in sorting is the swapping of two nodes in a linked list. Such a SWAP operation for a log-based linked list, given two nodes, creates a log entry that modifies the links between each node and their predecessors and successors in such a way that they take each other's place. If one of the swapped nodes happened to be the head of the linked list, the SWAP operation also writes a new head pointer to the log.

Using a singly linked list, the SWAP would perform with a time complexity of $\mathcal{O}(n)$, as in order to obtain the predecessors of the nodes to be swapped, the list would have to be traversed. This is why our array implementation uses a doubly linked list to represent its internal state, so that each node has points to both its predecessor and its successor.

Algorithm 1 Pseudocode for a SWAP implementation of a log-based linked list.

function SWAP(L, a,b)
  if $a = b$ **then return**
  end if
  $e \leftarrow$ Initialize a new log entry
  if $\succ_a = b$ **then** ▷ $\succ_a$ is a's (possibly nil) successor node, $\prec_a$ is its predecessor
    $e.links \leftarrow \langle\langle \prec_a, b\rangle, \langle b, a\rangle, \langle a, \succ_b\rangle\rangle$
  else if $\succ_b = a$ **then**
    $e.links \leftarrow \langle\langle \prec_b, a\rangle, \langle a, b\rangle, \langle b, \succ_a\rangle\rangle$
  else
    $e.links \leftarrow \langle\langle \prec_b, a\rangle, \langle a, \succ_b\rangle, \langle \prec_a, b\rangle, \langle b, \succ_a\rangle\rangle$
  end if
  if head$(L) = a$ **then** ▷ head(L) yields the head node of list L
    $e.head \leftarrow b$
  else if head$(L) = b$ **then**
    $e.head \leftarrow a$
  end if
  Append e to the log ▷ Also triggers consumption of the same log entry
end function

A pseudocode implementation of such a SWAP operation is listed in Algorithm 1. This implementation exhibits typical characteristics of write operations in *PREDSL*: Even though SWAP is an operation that would modify the list, it does not actually alter the internal state

of the data structure. Instead, the changes are appended to the log. Only after the log implementation has successfully committed the new entry to storage, is it handed back to the data structure for consumption and modification of internal state.

Based on this SWAP primitive we build our sorting operation. We chose the *selection sort* algorithm for this implementation. While selection sort has a time complexity of $\Theta(n)$, it only requires $\mathcal{O}(1)$ memory and has the additional property that its guaranteed to perform the minimal number of SWAP operations in order to sort the list.

In our log-based approach to model data structure modifications, this last property carries even more weight, as it helps to keep the number of operations written to the log to a minimum. A pseudocode implementation of SELECTIONSORT is listed in Algorithm 2.

Algorithm 2 Pseudocode for a SELECTIONSORT implementation of a log-based linked list.

```
function SelectionSort(L)
    a ← head(L)
    while ≻_a ≠ nil do
        min ← a
        b ←≻_a
        while b ≠ nil do
            if b < min then
                min ← b
            end if
            b ←≻_b
        end while
        if min < a then
            Swap(L, a, min)
            a ← min
        end if
        a ←≻_a
    end while
end function
```

3.3.2 Dictionaries

Dictionaries, also called maps, are another basic data structure. They consist of mappings from discrete, unique keys to arbitrary values. In most implementations, dictionary keys have to be some kind of scalar type, whereas there usually is no restriction whatsoever on the type of values. For example, in our implementation only strings of text are permitted as keys, whereas the values these keys map to can even be other data structures.

Even though a dictionary may internally sort key-value pairs in order to increase lookup speed, their order is usually not exposed. Thus, for our implementation, while keys may be sorted internally, there is no relationship between any two keys encoded on the log. We implement the following low-level operations:

put(k, v) Create a new entry in the dictionary, consisting of a key k and a value v. If the dictionary already contains an entry with the same key k, this entry is replaced by the new one. k must be a scalar value , whereas v can be of any type, possibly another nested data structure.

delete(k) Remove the entry with key k from the dictionary, if it exists. If there is no such entry, do nothing.

clear() Remove all entries from the dictionary.

A entry defining a dictionary entry could look like this:

```
{
  "duid": 1,
  "type": "dictionary",
  "serial": 10,
  # The key to insert into the dictionary.  This implementation
  # restricts itself to strings.
  "key": "answer",
  # The value to map the key to.
  "value": {
    "type": "immediate",
    "value": 42
  },
  # Remove the key from the dictionary, rather than adding a new
  # key-value pair.  The "delete" option is mutually exclusive
  # with the "value" option.
  "delete": true
}
```

Same as with our linked list-operation, by convention there can only be a single key-value pair per log entry. Since all operations we define on dictionaries refer to at least a key, each log entry will always only contain a single operation. This also means that the *merging preemptive rewriting* strategy presented in Section 3.4.4 is not applicable for this data structure.

The implementation of a dictionary data structure, alongside with that of an array and the support for nesting of data structures, enable *PREDSL* to encode JSON-like documents in a similar fashion as is done by the CRDT implementations DSON [13] and YATA [10] introduced in Section 2.5.

3.3.3 Sets

A set is similar to a dictionary, except that it only describes a unique set of keys, without mapping them to values. Unlike with dictionaries, we permit any type of data structure in the “keys” of a set.Instead this implementation, the ”value” is a boolean flag indicating whether to add or remove a key from the set). This the operations on sets are similar to those in dictionaries:

add(k) Add a new key k to the set. If the set already contains the key k, this operation has no effect. Unlike in dictionaries, k may be of any type.

remove(k) Remove the key k from the set, if it exists. If there is no such key, do nothing.

clear() Remove all keys from the set.

Whether a key (encoded in the "value" property) is to be added or removed from the set is indicated by an additional "delete" flag. If the delete flag is set without providing a value, the set is emptied out. A log entry adding a key to a set could look like this:

```
{
  "duid": 1,
  "type": "set",
  "serial": 10,
  # The key to add to or remove from the set.
  "value": {
    "type": "immediate",
    "value": 42
  },
  # This key is being added, not removed.
  "delete": false
}
```

Same as with dictionaries, *merging preemptive rewriting* is not applicable for sets.

3.3.4 Graphs

A graph is defined by a set of vertices and edges between these vertices.
We define the following operations on graphs:

add(v) Create a new vertex in the graph, which holds the value v. Initially the vertex is not connected to any other vertex in the graph. From here on, the vertex is uniquely identified by the sequence number n of the log entry it was created in.

link(n, m, w) Create an edge $\langle n, m \rangle$ in the graph between the vertices with identifiers n and m. Depending on the specific type of graph data structure, this edge is either directed or undirected; for directed graphs it is oriented from m to n. For weighted graphs, the additional parameter w specifies the weight of the edge; for unweighted this parameter is omitted.

unlink(n, m) Remove the edge $\langle n, m \rangle$ between vertices n and m, if it exists. If such an edge does not exist, this operation does nothing. In directed graphs, an inverted edge $\langle m, n \rangle$ is not affected by this deletion.

remove(n) Remove vertex n from the graph, if it exists. If such a vertex does not exist, this operation does nothing. All incoming and outgoing edges are removed implicitly.

As with other data structures, multiple of these operations can be combined into a single log entry, with the exception that only one new vertex can be created per entry. The

following (rather theoretical) example of a log entry showcases all 4 operations: It inserts a vertex into a graph and immediately connects it to two other vertices, while simultaneously removing existing edges and vertices:

```
{
  "duid": 1,
  "type": "graph",
  "serial": 10,
  # The value assigned to a vertex in a graph
  "value": {
    "type": "immediate",
    "value": 42
  },
  # Connect the new vertex to existing ones 4 and 8.
  "edges": {
    "10": [4, 8],
  },
  # Remove the existing edge between 4 and 8.
  "unlink": [
    [4, 8]
  ],
  # Delete vertex 5.
  "delete": [5]
}
```

The graph implementation provides for different operation modes, encoded in log entries as four different data types:

- Unweighted and undirected, encoded as `graph`.
- Weighted and undirected, encoded as `wgraph`.
- Unweighted and directed, encoded as `dgraph`.
- Weighted and directed, encoded as `dwgraph`.

In the evaluation part of this thesis, only a single implementation is examined, since all four implementations are equivalent apart from their weightedness and directedness.

3.3.5 Trees

A tree data structure generally consists of nodes in a hierarchical relationship with a single node as the root of the tree. Each node has a value assigned to it, and an arbitrary number of ordered children. As before, nodes are uniquely identified by the sequence number of the log entry they were initially defined in. We define the following operations on *PREDSL*'s tree implementation:

create(v) Create a new node to be inserted into the tree. The node holds the value v and is uniquely identified by the log entry sequence number n.

children(p, [c]) Replace the sequence of children of the node identified by p by the sequence of node identifiers $[c]$. If the node p or any node in $[c]$ does not exist, the log entry is rejected. Nodes that are removed from the tree hierarchy by this operation are removed entirely unless they are *reparented*, i.e. reinserted as a child of another node in the tree, within the same log entry.

root(n) Set the tree's root node to be n. If the former root node is not reparented by means of the *children* operation, it (and any of its non-reparented children) are removed from the tree. As a tree can – by definition – only have a single root node, this operation can also only occur once in a log entry.

As an example, consider the following log entry inserting a new node at the root of a tree, replacing another node and inheriting its children.

```
{
  "duid": 1,
  "type": "tree",
  "serial": 10,
  # The value of the node to create.
  "value": {
    "type": "immediate",
    "value": 42
  },
  "children": {
    # Node 10 inherits the existing children
    # of the node it replaces.
    "10": [3, 5, 1]
  },
  # Node 10 is set as the new root of the tree.  As its
  # predecessor is not reparented, it is implicitly removed
  # from the tree.
  "root": 10
}
```

This way of implementing modifications in a tree may seem odd at first: For one thing, even trivial modifications such as adding a new node to the tree require multiple operations. For another thing it may seem wasteful to always provide the entire list of children even when only changing a single child.

The latter decision was made in order to be able to support an arbitrary and changing number of (ordered) child nodes: If only the modifications to the child sequence were encoded on the log, we would be facing the same challenges already discussed in Section 3.3.1. In

order to keep the tree implementation simpler, and making the assumption that most trees would on average have a small number of children per node, we decided on this compromise of implementation complexity versus log space usage.

The especially low-level approach to modification operations was chosen in order to be able to support a wider range of higher-level operations. For example, when a new node is inserted into a tree, it can, depending on the type of tree being implemented, either take another node's place or be inserted in between two nodes.

The implementation of a tree shown here can be considered too generic for most applications: Neither does it pose restriction on the number of children, nor does it define the exact semantics of operations such as "inserting a node". However, being this generic mainly serves the purpose of acting as a foundation on which more specialized types of trees can be implemented. In the following we present two types of trees implemented on this basis.

3.3.5.1 Binary Search Trees

A *binary tree* is a special type of tree where each node has exactly two (possibly *nil*) children, designated "left" and "right". It is commonly used as a binary *search* tree, where the "left child of" and "right child of" relationships between nodes are determined by the nodes' values being smaller or greater than their parents' values.

In our implementation, a binary search tree is modeled exactly the same as the generic tree discussed above, with the exception that it carries a different type designation (`2tree` instead of just `tree`). A *children* entry in the log will always contain exactly two items, referring to the left and right child nodes. Absent children are indicated by *nil* values, as shown in the following example:

```
{
  "type": "2tree",
  ...,
  "children": {
    # node 10 only has a left child
    "10": [4, null]
  },
  ...
}
```

On this binary search tree, we define the following, higher-level modifications, which make use of the more low-level operations introduced earlier:

insert(v) Create a new node with value v. Iterate the tree starting at the root, continuing left if v is smaller than the current node's value, or right if v is greater, until a *nil* entry is encountered. The new node is inserted at the position of the *nil* entry. Inserting a value already present in the search tree is no permitted.

delete(v) Locates the node with value v in the search tree. If such a node exists, it is removed and its children, if any, are reparented at appropriate locations (see Cormen

et al. [3, p. 295f] for the different cases of node deletion in a binary search tree). If such a node does not exist, this operation does nothing.

3.3.5.2 AVL Trees

An AVL tree is a specialization of a binary search tree, where after each modification a (possibly empty) sequence of *rotations* is applied in order to keep the tree balanced. A rotation changes the hierarchy of some nodes while maintaining the order of nodes as indicated by their "left child of" and "right child of" relationships. In total, there are four different rotations [6, p. 461ff], for all of which we provide our implementations in pseudocode in Algorithms 3 and 4.

Algorithm 3 Pseudocode implementation of the two single rotations by means of appending entries to the data structure's log.

```
function RotateLeft(T, n)
    p ← parent(n)                          ▷ parent(n) yields the parent of node n
    r ← right(n)                           ▷ left(n) and right(n) yield the children of node n.
    if r = nil then
        raise error                        ▷ right(n) must not be nil
    end if
    e ← Initialize a new log entry
    e.children[n] ← ⟨left(n), left(r)⟩     ▷ n adopts the left child of r
    e.children[r] ← ⟨n, right(r)⟩          ▷ n becomes the left child of r
    if p ≠ nil ∧ left(p) = n then          ▷ r replaces n as a child of p
        e.children[p] ← ⟨r, right(p)⟩
    else if p ≠ nil ∧ right(p) = n then
        e.children[p] ← ⟨left(p), n⟩
    else if root(T) = n then               ▷ If n is the tree's root, r becomes the new root
        e.root ← r
    end if
    Append e to the log                    ▷ Also triggers consumption of the same log entry
    return r
end function
```

```
function RotateRight(T, n)
    p ← parent(n)
    l ← left(n)
    if l = nil then
        raise error                        ▷ left(n) must not be nil
    end if
    e ← Initialize a new log entry
    e.children[n] ← ⟨right(n), right(l)⟩   ▷ n adopts the right child of l
    e.children[l] ← ⟨left(l), n⟩           ▷ n becomes the right child of l
    ▷ ...                                  ▷ Same reparenting as in RotateLeft
    Append e to the log
    return l
end function
```

These rotations are used by an AVL tree to *rebalance* a (sub-) tree that is no longer balanced, i.e. the heights of the left and right subtree differ by more than 1. In order to do so, the rebalancing routine shown in Algorithm 5 is applied to the **parent** of the node that has been modified after each *insert* or *remove* operation.

Algorithm 4 Pseudocode implementation of the two double rotations by means of appending entries to the data structure's log.

```
function RotateLeftRight(T, n)
    if left(n) = nil ∨ right(left(n)) = nil then
        raise error                          ▷ Both left(n) and right(left(n)) must not be nil
    end if
    p ← parent(n)
    l ← left(n)
    r ← right(l)
    e ← Initialize a new log entry
    e.children[r] ← ⟨l, n⟩                   ▷ r adopts its parent l and grandparent n
    e.children[l] ← ⟨left(l), left(r)⟩       ▷ l adopts r's former left child
    e.children[n] ← ⟨right(r), right(n)⟩     ▷ n adopts r's former right child
    if p ≠ nil ∧ left(p) = n then            ▷ r replaces n as a child of p
        e.children[p] ← ⟨r, right(p)⟩
    else if p ≠ nil ∧ right(p) = n then
        e.children[p] ← ⟨left(p), n⟩
    else if root(T) = n then                 ▷ If n is the tree's root, r becomes the new root
        e.root ← r
    end if
    Append e to the log                      ▷ Also triggers consumption of the same log entry
    return r
end function
```

```
function RotateRightLeft(T, n)
    if right(n) = nil ∨ left(right(n)) = nil then
        raise error
    end if
    p ← parent(n)
    r ← right(n)
    l ← left(r)
    e ← Initialize a new log entry
    e.children[l] ← ⟨n, r⟩                   ▷ l adopts its parent r and grandparent n
    e.children[r] ← ⟨right(l), right(r)⟩     ▷ r adopts l's former right child
    e.children[n] ← ⟨left(n), left(l)⟩       ▷ n adopts l's former left child
    ▷ ...                                    ▷ Same reparenting as in RotateLeftRight
    Append e to the log
    return l
end function
```

Note that rebalancing a node may reduce its subtree height, and thus bring its parent out of balance. This is why the rebalancing routine iterates up the tree until balance is restored.

Algorithm 5 Pseudocode implementation of the AVL tree rebalancing algorithm, as implemented in *PREDSL*.

```
function RebalanceNode(T, n)
    if bal(n) < −1 then                    ▷ bal(n) yields the difference of subtree heights
        if bal(left(n)) > 0 then
            return RotateLeftRight(n)
        else
            return RotateRight(n)
        end if
    else if bal(n) > 1 then
        if bal(right(n)) < 0 then
            return RotateRight(n)
        else
            return RotateLeft(n)
        end if
    end if
    return n
end function
```

```
function Rebalance(T, n)
    p ← n
    while p ≠ nil ∧ |bal(p)| > 1 do
        p ← RebalanceNode(T, p)()
        p ← parent(p)
    end while
end function
```

This implementation of the AVL tree rebalancing creates a log entry per rotation, so that per *insert* or *delete* operation up to $\mathcal{O}(\log_2 n)$ entries may be appended to the log. While it would theoretically be possible to encode a modification including all rebalancing operations inside a single log entry, the tree's state would have to maintained twice during the modification: Once as it represented on the log, and once as it is being modified by subsequent rebalancing operations.

Thus we had decided to implement the multi-entry solution discussed her. Even though a small storage overhead is incurred per log entry, implementation is greatly simplified, and the overhead can at least be partially compensated if the *merging preemptive rewriting* strategy is applied.

3.4 Log Rewriting Strategies

In this section we describe the strategies we employed to decide when and how to rewrite an old log entry at the head of the log. In order to describe and compare the different strategies, we introduce the metric of *overhead* on logs: The overhead $k(l, t)$ of a log l at the time that its head entry has sequence number t is the ratio between the length of its *current* sliding window and its *smallest possible* sliding window:

$$k(l,t) = \frac{t - o(l)}{\sum_{D \in l} |D|} \tag{3.3}$$

$t - o(l)$ describes the length of the current sliding window, which starts at the oldest relevant entry $o(l)$ (see Equations 3.1 and 3.2) and ends at the head t of the log.

$\sum_{D \in l} |D|$ describes the smallest possible length of the sliding window: As explained in Section 3.3, each item in a data structure must be defined in its own log entry due to the convention of using the log entry sequence number as an identifier for the data structure item created in such an entry. In addition, all operations that modify relationships between these items can be encoded in the same log entries the affected items are created in, there never needs to be more than one log entry for each item in a data structure. Thus, the most compact representation of a data structure in *PREDSL* is always equal to the number of items in a data structure, or in other words its length $|D|$.

The rewriting strategies implemented in *PREDSL* make use of this *overhead* metric when deciding whether log entries need to be rewritten. In the remainder of this section we introduce each of the rewriting strategies we implemented and evaluated.

3.4.1 Baseline: No Rewriting

The simplest and most trivial rewriting strategy is to never actively rewrite log entries at all. The oldest entry in a log is only removed once it is rendered obsolete through "natural" use, e.g. by being removed from the data structure. As discussed in the previous section, there is no guarantee that this will happen in all cases, so this strategy mainly serves as a baseline to compare our other strategies to.

3.4.2 Single Rewriting

After each modification of a data structure, the overhead $k(l,t)$ of the log is computed and compared against a preconfigured threshold k_{max}. If this threshold is exceeded, a single rewrite operation is appended as an additional log entry.

The actual creation of the rewrite log entry is delegated to the affected data structure (note that this is the data structure referred to by the oldest log entry, and not necessarily the data structure that was modified in the just-appended entry). This is done because each data structure has different information that has to be contained in the rewrite log entry, e.g. for the linked list, the links to an item's predecessor and successor need to be encoded in the rewrite entry, in addition to an item's value, whereas for example with a tree, the children of a node, as well as those of that node's parent must be provided. The rewriting algorithm for our linked list implementation is shown in Algorithm 6.

It is possible that a single rewrite is not sufficient to reduce the overhead to $k < k_{max}$. Under some usage scenarios, it may even occur that every single modification to a data structure makes the log overhead exceed k_{max}, and there is constant rewriting without reducing the sliding window length. This is where the following *batch rewriting* strategy comes in.

Algorithm 6 The log entry rewriting algorithm of our linked list implementation: A new log entry containing the old entry's value is created. In addition, the links to the predecessor and successor nodes are replaced, and if necessary, the list's head pointer is set to the new log entry.

```
function LinkedListRewrite(L, o)                    ▷ Linked List L, oldest entry o
    e ← Initialize a new log entry
    e.value ← o.value                               ▷ val(L, o) retrieves the value store in entry o
    e.links ← ⟨⟨≺_o, id(e)⟩, ⟨id(e), ≻_o⟩⟩
    if head(L) = o then
        e.head ← e
    end if
    return e
end function
```

Algorithm 7 The *single rewriting* strategy: After appending a log entry, if the overhead exceeds k_{max}, an additional rewrite entry is created and appended to the log.

```
function AppendRewriteSingle(l, e, k_max)           ▷ Log l, new entry e
    Append(l, e)
    if k(l, id(e)) >= k_max then
        D ← GetDatastructure(o(l))                  ▷ Data structure D of the oldest log entry
        r ← Rewrite(D, o(l))                        ▷ Instruct D to create rewrite entry r
        Append(l, r)
    end if
    Prune(l)                                        ▷ Remove the now obsolete old log entries
end function
```

3.4.3 Batch Rewriting

Same as with the *single rewriting* strategy, after each modification of a data structure, the overhead of the log is computed and compared against a preconfigured threshold k_{max}. If this threshold is exceeded, rewrite operations are applied until the overhead falls below a second, smaller threshold k_{min}. Pseudocode for this strategy is shown in Algorithm 8.

Algorithm 8 The *batch rewriting* strategy: After appending a log entry, if the overhead exceeds k_{max}, create and append rewriting log entries until the overhead reaches k_{min}.

```
function AppendRewriteBatch(l, e, k_max, k_min)     ▷ Log l, new entry e
    Append(l, e)
    if k(l, id(e)) >= k_max then
        while k(l, id(e)) > k_min do
            D ← GetDatastructure(o(l))              ▷ Data structure D of the oldest log entry
            r ← Rewrite(D, o(l))                    ▷ Instruct D to create rewrite entry r
            Append(l, r)
            Prune(l)                                ▷ Remove the now obsolete old log entries
        end while
    end if
    Prune(l)
end function
```

3.4.4 Merging Preemptive Rewriting

The previously discussed rewriting strategies all waited for some preconfigured threshold to be exceeded before applying rewrite operations. We also considered a strategy which would take advantage of any change they get to rewrite old log entries. We call this strategy *merging preemptive rewriting* strategy:

Before each modification is appended to the log, it is inspected whether the created log entry contains a *value* attribute. If a value is present, no rewriting is applied. If there is no value, the rewrite operation is not written to a separate log entry, but *merged* into the original modification entry, before they are appended to the log as a single entry. In order to support this, the rewrite procedure of the data structures must be adapted to handle the merge functionality, as shown in Algorithm 9.

Algorithm 9 The log entry rewriting algorithm of our linked list implementation, adapted for optionally merging the rewrite into an already prepared log entry.

```
function LinkedListRewriteMerge(L, o, e)          ▷ Linked List L, oldest entry o
    if e ≠ nil ∧ e.value ≠ nil then
        return nil          ▷ Cannot merge rewrite if there already is a value in the entry
    else if e = nil then          ▷ Only create a new log entry if not merging with another
        e ← Initialize a new log entry
        e.links ← ⟨⟩
    end if
    e.value ← o.value
    e.links ← e.links ‖ ⟨⟨≺_o, id(e)⟩, ⟨id(e), ≻_o⟩⟩ ▷ Append rewritten links to links in entry
    if head(L) = o ∧ e.head = nil then
        e.head ← e                                  ▷ Rewrite head if current head is o...
    else if e.head = o then
        e.head ← e                                  ▷ ... or if the log entry would set it to o
    end if
    return e
end function
```

With this adapted rewrite procedure, the *merging preemptive rewrite* strategy, as shown in Algorithm 10 can be applied. For best results, this strategy can be combined with our previous threshold-based strategies in order to maintain a manageable sliding window size in all usage situations.

Algorithm 10 The merging preemptive rewriting strategy: A new log entry is not append directly, but it is first attempted to merge a rewrite operation into the entry.

```
function AppendPreemptiveMerge(l, e)                    ▷ Log l, new entry e
    if e.value = nil then
        D ← GetDatastructure(o(l))
        r ← Rewrite(D, o(l), e)
        if r ≠ nil then
            e ← r
        end if
    end if
    Append(l, e)
end function
```

4 Evaluation

In this chapter we evaluate the performance of the log rewriting strategies we introduced in Section 3.4 when applied to the different data structures described in Section 3.3 and present our results.

The remainder of this chapter is structured as follows: In Section 4.1 we describe the design of our evaluation and which metrics we analyzed. In Section 4.2 we present the results of this evaluation. Finally, in Section 4.3 we discuss our results.

4.1 Performance Evaluation

At first, in order to evaluate performance, we need to get an understanding of how log entries are distributed in a log, and how this distribution evolves over time. In order to facilitate this understanding, we introduce *valley plots*, where we plot a log's sliding window and the significance of each log entry over time.

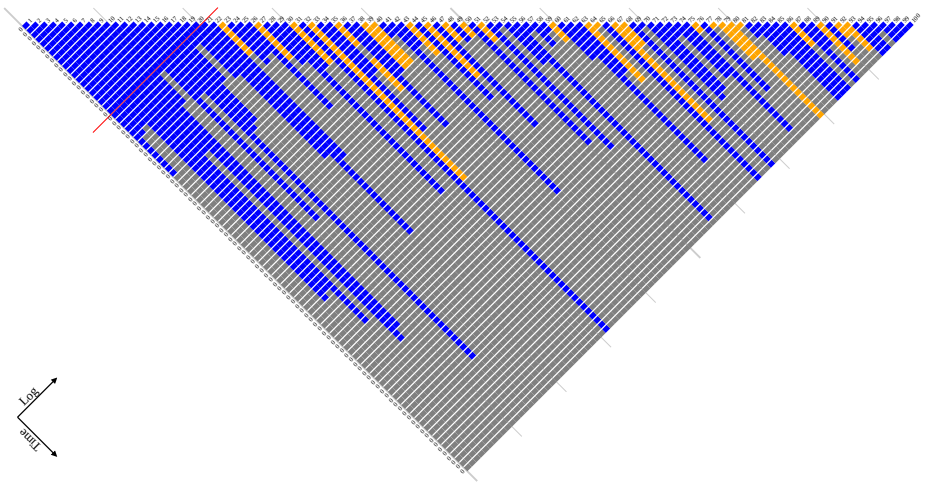

Figure 4.1: Valley plot of 100 randomized linked list insertions, deletions and replacements with all pruning and rewriting disabled. The first 20 entries (left of the red line) are insert operations only, so that there actually are entries which can be deleted.

In such a valley plot, as shown in Figure 4.1, we insert a new "line" (angled by 45 degrees for a more compact rendering) each time after a new entry is appended to a log. This line spans the sliding window of the log at the time the entry was appended, and annotates each log entry between the oldest entry (the "bottom-left-most" in this line) and the just appended entry (the "top-right-most") with a color indicating its relevance.

An entry is rendered in gray if it no longer contains relevant information, but has not been pruned yet. Otherwise, it is rendered in blue or yellow, depending on the type of information encoded in the log entry: An entry is rendered in blue, if if defines a new item in a data structure, or in yellow, if it only defines relationships between already existing items. The significance of this distinction is explained in detail in Section 3.2. Note that in some occasions (e.g. with merging preemptive rewriting) log entries may turn from blue to yellow over time. This happens if the item value defined in such an entry becomes obsolete, but if the same entry still contains valid information about relationships between items.

If we consider Figure 4.1 in more detail, we can see that at all times, the majority of valid log entries is concentrated at the newer end of the sliding window ("further up" on the vertical axis in a valley plot); the older a log entry becomes, the greater the change of it becoming obsolete (i.e. the further down we go in the plot, the larger the gray portion of the sliding window becomes). The exact distribution depends on how the data structure is used, but this observation holds true in general for all usage patterns where existing items in a data structure are modified or removed.

If we recall Section 3.2, the goal achieved by log pruning is to keep the size of the log sliding window as small as possible. More precisely, the *maximal* sliding window size over all time should be minimized. Transferred to the valley plot, this means that the (vertical) height of the plot should be minimized.

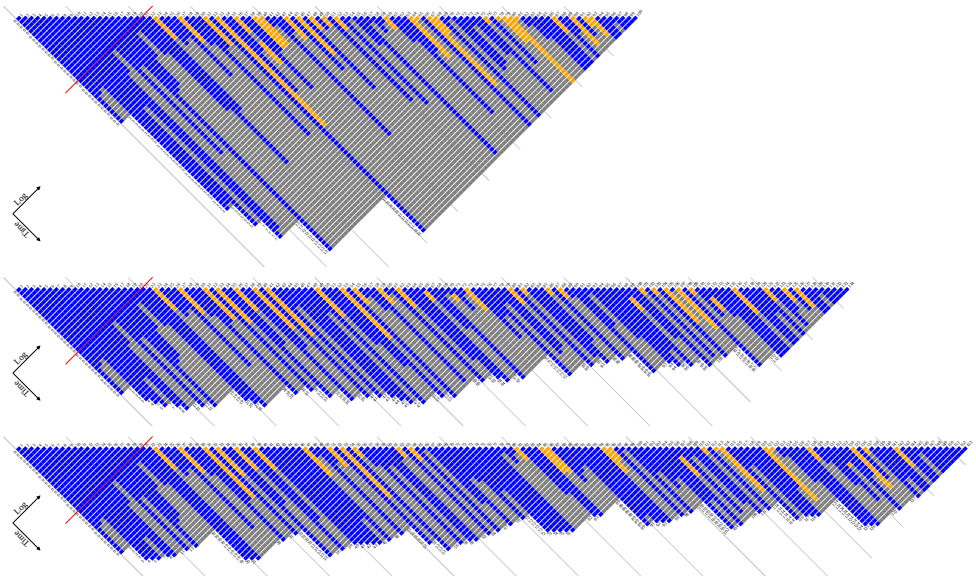

Figure 4.2: Valley plots of 100 randomized linked list insertions, deletions and replacements using different rewriting strategies: Pruning without rewriting (top), Single rewriting (center) and batch rewriting (bottom).

In Figure 4.2, we can see valley plots for the same 100 random operations as in Figure 4.1, but applying different rewrite strategies. We can see that all our strategies help in reducing the maximal sliding window size. This can be seen even more clearly in Figure 4.4. However, the same plots also reveal that, in doing so, rewriting increases the total number of written log entries. For example, in the bottom plot in Figure 4.2, a total of 153 log entries are written, despite encoding the same 100 modifications as in the examples above.

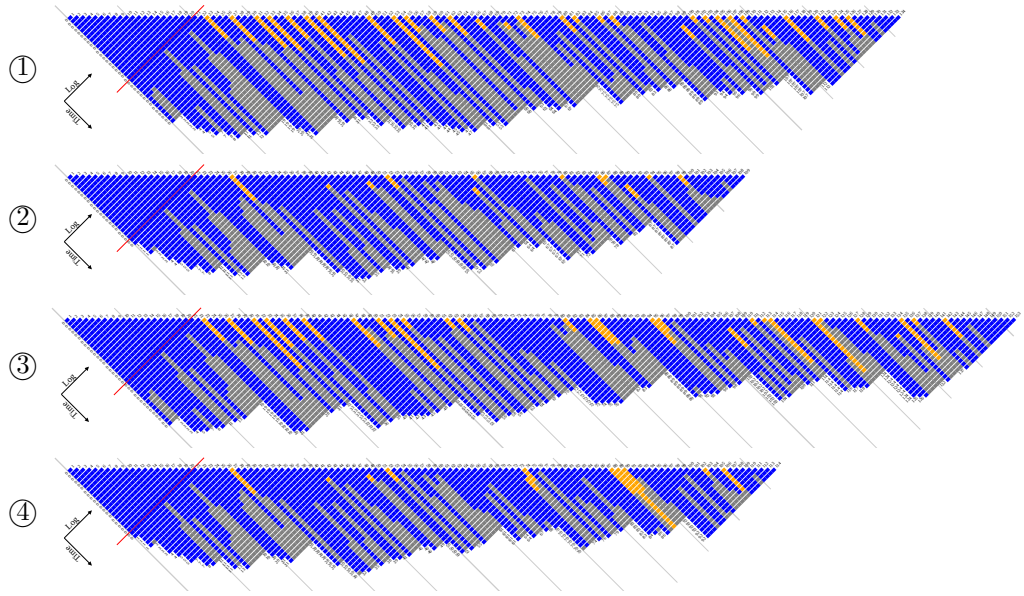

Figure 4.3: Comparison of valley plots for single and batch rewriting, with and without *merging preemptive rewrites* applied additionally. From top to bottom: ① single rewrites, no preemptive rewriting. ② single rewrites, with preemptive rewriting. ③ batch rewrites, no preemptive rewriting. ④ batch rewrites, with preemptive rewriting.

Since each additional log entry is replicated and processed by all of the log's consumers, we achieve this reduced storage consumption at the cost of increased processing time at both the producer and consumer sides. The number of written log entries can be reduced when we apply the *merging preemptive rewrite* strategy in addition to other strategies, as shown in Figure 4.3. Thus, we set a secondary goal to write as little additional log entries as necessary to achieve the primary goal of reduced lot storage.

While valley plots help to get an intuitive understanding of the distribution of entries in a log and of the general performance of a rewrite strategy for a small-scale test scenario, they do not scale well and become hard to read for larger, more extensive test cases. In addition, in such small test cases, the meaningfulness of valley plots is reduced, since the differences between different rewrite strategies are rather small, as becomes evident e.g. in Figure 4.4.

Thus, in order to properly evaluate and compare the number of additionally written log entries in terms of numbers, we define the *rewrite efficiency* of a data structure under a specific use case to be the relative amount of log entries that are used for actual modifications of the data structure while rewriting is not necessary, i.e. $k(l,t) < k_{max}$. A log entry is not counted towards this relative amount if $k(l,t)$ exceeds k_{max} (i.e. rewriting is necessary) or if the entry is purely a rewrite entry. In other words, if active rewriting is never necessary, the

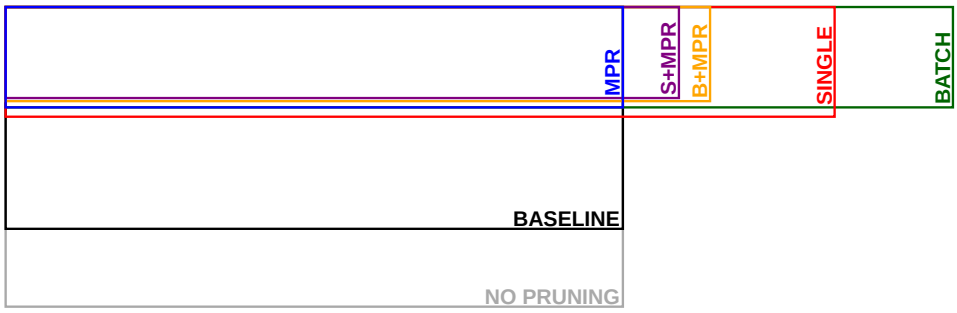

Figure 4.4: The bounding boxes (rectangles drawn around the plots) of each valley plot shown in Figures 4.1, 4.2 and 4.3, overlaid over each other to better demonstrate the differences in size. Rendered like this, we can clearly see that all rewriting strategies produce much shorter sliding windows. We can also see that merging preemptive rewriting (shortened as *MPR* in this figure) helps produce less log entries when rewriting the log.

efficiency would be 1, and the more time (i.e. number of log entries) is spent with rewriting, the further the performance approaches 0.

As a test case run progresses, the rewrite efficiency metric oscillates (increase when modifying the data structure within the tolerated threshold, decrease when rewriting or exceeding k_{max}) around a value specific to a test case, to which it will eventually converge:

$$E_c = 1 - \lim_{N \to \infty} \frac{R}{N} \tag{4.1}$$

Where E_c is the rewrite efficiency under test case c, R is the number of rewrite or k_{max}-exceeding entries and N is the number of total log entries. Since this metric is specific to the test case, it does not serve well as an absolute, all-encompassing metric. However, it enables us to compare the behavior of different rewrite strategies to each other.

In the remainder of this chapter, we analyze the maximal sliding window size and rewrite efficiency of the data structures and rewrite strategies implemented in *PREDSL* in more detail when subjected to different test cases.

4.2 Test Cases & Results

In this Section we describe the results obtained for each test case. We evaluated all applicable rewriting strategies against each test case. For strategies where applicable (all except baseline and standalone *merging preemptive rewriting*) we varied the maximum tolerated overhead k_{max} and analyzed how it impacts sliding window length and rewrite efficiency.

4.2.1 Linked Lists

This test case consisted of 50 sequential insertions into a linked list, followed by 1000 operations at random positions, uniformly distributed between insertion at, deletion of and replacement of the chosen position. The evaluation for this test case was run for values of k_{max} between 1.0 and 5.0 in increments of 0.1. Each configuration was evaluated 20 times.

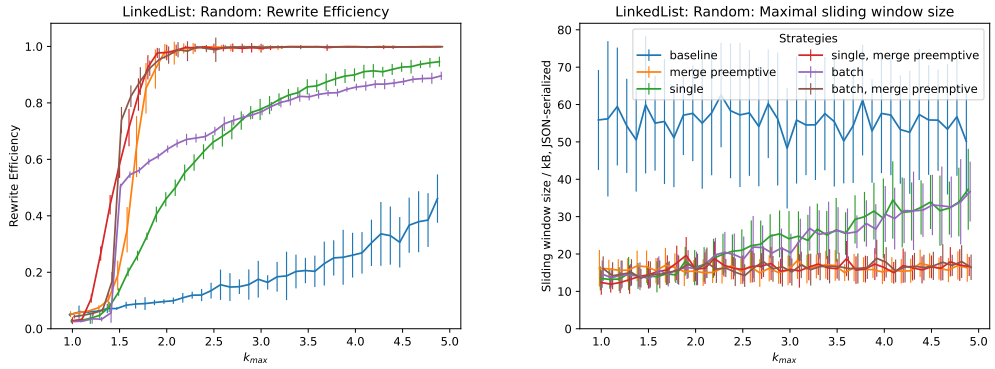

Figure 4.5: Rewrite efficiency and maximal sliding widows size of the randomized linked list test case as a function of k_{max} for each rewrite strategy. The legend on the left-hand plot has been omitted because there was no space for it, however the colors are assigned the same as on the right-hand side.

The results of this test case are shown in Figure 4.5. The plots show the mean values of these 20 runs, with vertical error bars indicating standard deviation.

We can show that all rewriting strategies can achieve a significantly higher rewrite efficiency than the baseline strategy, even for very small values of k_{max}. The higher the rewrite efficiency, the more time is the log's overhead k kept smaller than k_{max}, making rewriting necessary less often, and keeping the sliding window smaller than if no rewriting were applied.

With the strategies applying merging preemptive rewriting, we achieve rewrite efficiency values close to 1 for tolerated overheads k_{max} as low as 2.0. The strategies not employing merging preemptive rewriting do not reach high efficiency quite as fast, exceeding 0.8 only at $k_{max} > 3.5$. Nevertheless, they still achieve a much higher efficiency than the baseline, which never reaches a rewrite efficiency of greater than 0.5 at any value of k_{max} in the evaluated range.

At the same time, we can show that while the baseline log consumes a fluctuating amount of memory averaging around 55kB, while sometimes reaching sizes as high as close to 80 kB, the merging preemptive strategies manage to maintain a sliding window size of only 10 – 20 kB. The non-merging strategies achieve the same size for low values of k_{max}, but begin to approach the sliding window size of the baseline as more log overhead is tolerated.

4.2.2 Linked List Sorting

This test case consisted of 1000 sequential insertions of randomized, uniformly distributed numbers into a linked list, followed by sorting said list in-place using the algorithm discussed in Section 3.3.1.1. The evaluation for this test case was run for values of k_{max} between 1.0 and 5.0 in increments of 0.1. Each configuration was evaluated 20 times.

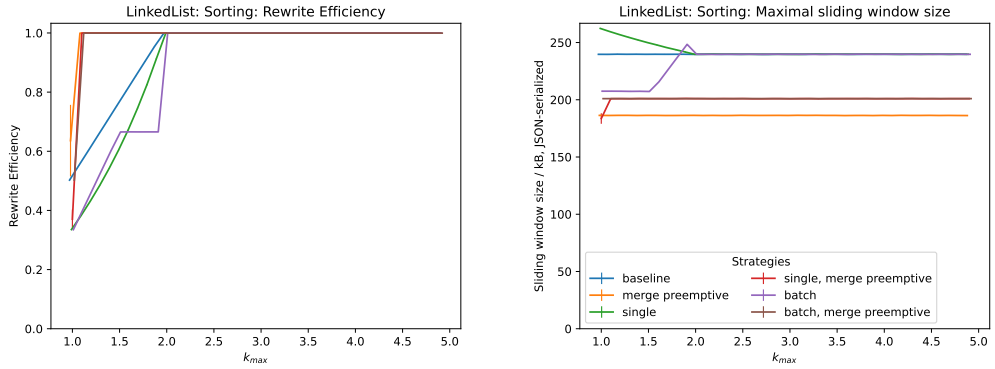

Figure 4.6: Rewrite efficiency and maximal sliding widows size of the linked list sorting test case as a function of k_{max} for each rewrite strategy.

The results of this test case, as shown in Figure 4.6, demonstrate the advantage of the merging preemptive rewrite strategy when the usage pattern consists mostly of modifications of existing items in a data structure, such as here the swap operations performed by the sorting algorithm: The strategies that employ rewrite merging reach a rewrite efficiency of 1 for all values of $k_{max} > 1$, while keeping the sliding window size consistently below the baseline, albeit only by a small amount. This can be explained by the fact that the insertions before sorting are already as compact as they can be, so when the oldest entry is rewritten, only this single, oldest log entry can be pruned from the log, reducing the amount of space that can be recovered.

4.2.3 Dictionaries

This test case consisted of 1000 randomized operations on a dictionary, uniformly distributed between insertion (and if already present, replacement) of and deletion of a random key, itself uniformly distributed, in a dictionary. The evaluation for this test case was run for values of k_{max} between 1.0 and 5.0 in increments of 0.1. Each configuration was evaluated 20 times.

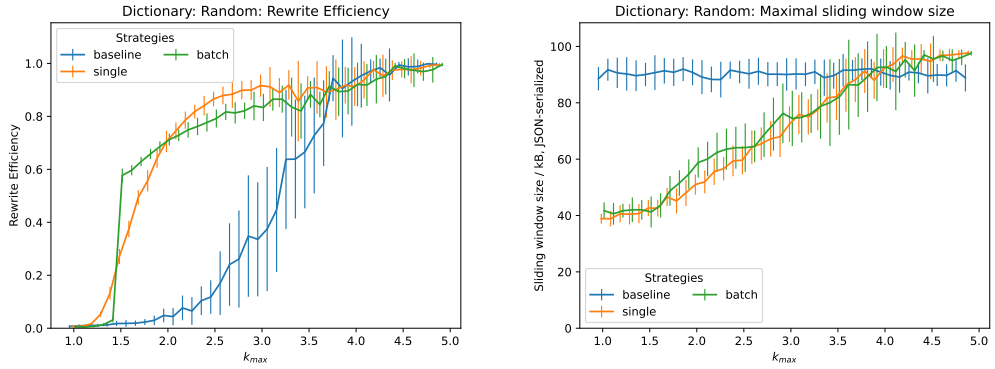

Figure 4.7: Rewrite efficiency and maximal sliding widows size of the randomized dictionary test case as a function of k_{max} for each rewrite strategy.

The results of this test case are shown in Figure 4.7. As discussed in Chapter 3, we did not apply merging preemptive rewriting to dictionaries and sets due to the design choices we made when implementing these two data structures. Both remaining rewrite strategies manage to achieve moderate to high rewrite efficiency at small tolerated overheads, reaching an efficiency of 0.8 as early as $k_{max} = 2.0$, whereas the baseline does not exceed this threshold before $k_{max} > 3.5$.

With a tolerated overhead of $k_{max} \leq 2.0$ both strategies manage to achieve a reduction in memory consumption of the sliding window of 40 – 50%, and converging to the same memory consumption as the baseline, as k_{max} is increased.

4.2.4 Sets

This test case consisted of 1000 randomized operations on a dictionary, uniformly distributed between insertion (and if already present, replacement) of and deletion of a random key, itself uniformly distributed, in a set. The evaluation for this test case was run for values of k_{max} between 1.0 and 5.0 in increments of 0.1. Each configuration was evaluated 20 times.

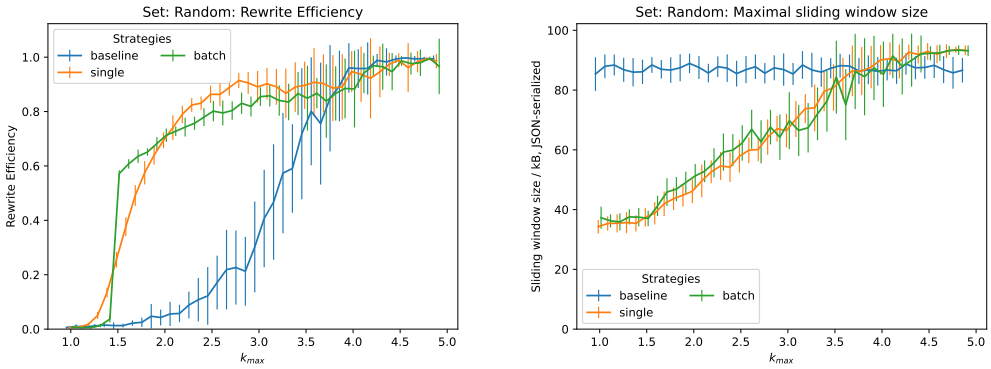

Figure 4.8: Rewrite efficiency and maximal sliding widows size of the randomized set test case as a function of k_{max} for each rewrite strategy.

The results, as shown in Figure 4.8, are nearly identical to those in Section 4.2.3. This comes as no surprise, as sets and dictionary are fairly similar data structures, and in our implementation only differ in whether a key is mapped to a value or simply marked as present.

4.2.5 Graphs

This test case consisted of 1000 randomized operations on a graph. With a chance of 0.25, a new vertex was created and connected to another random vertex in the graph. With a chance of 0.25, an existing vertex was deleted. With a chance of 0.5, two vertices in the graph were picked at random. If there already was an edge between them, the edge was deleted. If no edge existed between the chosen vertices, an edge was created. The evaluation for this test case was run for values of k_{max} between 1.0 and 5.0 in increments of 0.1. Each configuration was evaluated 20 times.

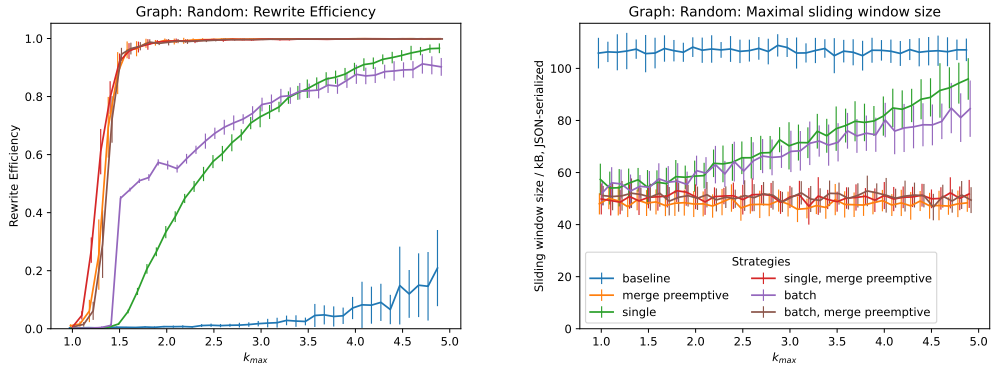

Figure 4.9: Rewrite efficiency and maximal sliding widows size of the randomized graph test case as a function of k_{max} for each rewrite strategy.

The results of this test case are shown in 4.9. Again, we can see the merging strategies to achieve a high rewrite efficiency with all but very small values of k_{max}, while maintaining a sliding window size roughly 50% less than the baseline. In contrast, the non-merging strategies only manage to maintain a comparable sliding window size with a greatly reduced rewrite efficiency. Only with a tolerated overhead of $k_{max} > 3.5$ does the rewrite efficiency exceed 0.8. In general, the results of this test case are very similar to that of the randomized linked list test case.

4.2.6 AVL Trees

This test case consisted of 1000 randomized operations on an AVL tree. With a chance of 0.25, an existing node was deleted from the tree. With a chance of 0.75, a new node with a randomly chosen value was inserted into the tree at the correct position. After each insertion or deletion, the tree rotations required to rebalance the tree are applied implicitly. The evaluation for this test case was run for values of k_{max} between 1.0 and 5.0 in increments of 0.1. Each configuration was evaluated 20 times.

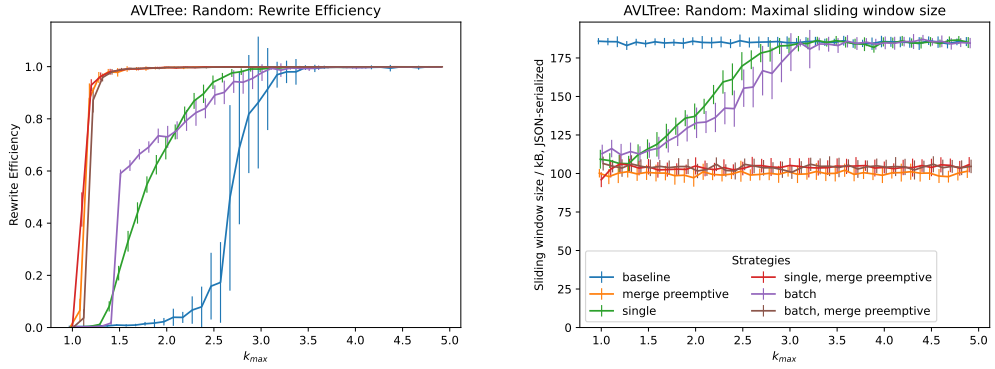

Figure 4.10: Rewrite efficiency and maximal sliding widows size of the randomized AVL Tree test case as a function of k_{max} for each rewrite strategy.

The results of this test case are shown in 4.10. The performance of all strategies is fairly similar to that of the graph and linked list test case in regard to both rewrite efficiency and size of the sliding window, even though the plots appears clinched in comparison; all strategies converge to a rewrite efficiency of 1 with much smaller k_max than they do in the other test cases. However, the non-merging strategies' sliding window size approaches the baseline size at an equally increased slope.

This improved rewrite efficiency can be explained by how our AVL tree implementation treats rebalancing of the tree: After a node is inserted into or removed from the tree, each rebalancing tree rotation is appended as an additional log entry. With every insertion or deletion entry that is rewritten and pruned from the log, the following rebalancing entries can be pruned as well. This means a smaller log size can be maintained with fewer rewrite operations, thus the increased rewrite efficiency.

4.3 Discussion

In the previous section we have analyzed how different rewriting strategies perform against different test cases using different k_{max} parameters.

One of the most significant observations we can make from our results is that each test case exhibits a specific *inherent* overhead $k_{lim}(l) = \lim_{t\to\infty} k(l,t)$ to which the log will converge as entries are appended. This observation appears to be true for all usage patterns with a constant ratio between and reasonably consistent distribution of inserting and non-inserting operations.

The exact value of the inherent overhead depends on the chosen rewrite strategy. In the rewrite efficiency plots shown in Section 4.2, we can see where k_{max} exceeds k_{lim}; at this position, the rewrite efficiency approaches 1. The inherent overhead even appears when no active rewriting is performed, i.e. when the "baseline" strategy is chosen.

From this follows that even without rewriting, a log does not exceed an inherent overhead, assuming that the previous assumptions about the usage pattern hold true. If enough storage space is available to be able to tolerate the overhead inherent to the baseline strategy, *PREDSL*-style logs can be operated even without rewriting.

However, our results show that the overhead inherent to the baseline strategy are fairly large; values of k_{lim} close to or exceeding 5.0 were not uncommon in our results, meaning that space would be required for at least five times as many log entries as would be necessary. On the other hand, when rewriting strategies are applied, the k_{lim} factor was greatly reduced, and some strategies even managed to bring k_{lim} to values around 1.5.

So far, we have been discussing memory requirements in terms of numbers of log entries, and multiples thereof. However, different types of log entries consume different amounts of memory, so let us now consider memory requirements in terms of absolute numbers. For this purpose, we also recorded the maximum sliding window size that occurred during each test run. This size was obtained by, after each entry was appended to the log, dumping the entire sliding window as a JSON document and recording it's size, measured in bytes.

Even though we are now dealing with absolute sizes in bytes, the exact numbers should still be treated with caution, as the actual memory requirement will again vary with usage

patterns, as well as in what format the log is stored. Nevertheless, these results serve as a good basis to compare different rewriting strategies to each other.

The results achieved by comparing the actual memory consumption of different rewriting strategies show that the amount of memory required can be reduced by $50 - -60\%$ in our test cases when choosing a rewrite strategy that employs merging preemptive rewriting, as compared to when no rewriting is applied.

This result differs from our observations regarding the overhead, where we observed much larger differences between our baseline and merging preemptive strategies. This is due to the fact that the overhead metric as we defined it appears to be biased towards favoring merged rewrites: A log entry that contains both a modification operation and a rewrite operation requires less memory than if the operations were split into two separate entries, due to the memory overhead incurred by each log entry. Nevertheless, it still requires more space than any of the non-merged entries would, while still counting to the overhead metric as only a single log entry. Thus, in a log with merging preemptive rewriting, the average size of a single log entry is larger than in a log where merging is not applied.

Nevertheless, this increased average log entry size is outweighed significantly by the reduced number of entries in the sliding window that is achieved by this rewriting strategy, leading to the observed results.

5
Conclusion

In the course of this thesis, we designed, implemented and presented *PREDSL*, a framework for replicating mutable data structures and their modifications as entries in immutable append only logs. We presented the data structures we implemented in *PREDSL* and how these data structures are represented on the log. In order to keep the size of these logs manageable, the oldest entries are pruned after they no longer contain data required to reconstruct the state of the data structure.

Analyzing the distribution of active and stale entries in the logs, we were able to show that with a randomized usage pattern, active log entries are more concentrated and more tightly compacted at the head of the log, and they become more spread apart in older log segments, with larger sequences of stale entries in between. These individual older, but still active entries surrounded by mostly stale entries restricted how much space could be recovered through pruning alone, since large potions of stale entries between still active entries could not be pruned.

To facilitate log pruning, we designed multiple strategies to rewrite old log entries at the head of the log in order to keep the actively maintained sliding window as small as possible at all times, while increasing the amount of log entries that can be pruned and whose space can be recovered. We evaluated these rewriting strategies against the data structures we implemented and presented the results of this evaluation.

Our results have shown that the rewriting strategies we designed indeed manage to reduce the memory footprint of such a log. Compared to the baseline scenario without log rewriting was applied, our strategies managed to reduce memory consumption by as much as 50 – 60 % in our randomized test cases. However, these numbers must be taken with a grain of salt; we have also shown that the capacity for reducing memory consumption of the log is highly dependent on the usage patterns the data structures on the log are subjected to. In test scenarios where the usage pattern already produces rather compact logs, such as the linked list sorting test case, there is not much room for improvement.

On the other hand, with usage patterns where there is a more far-spread range of how long log entries remain active – with some long lived entries and other, much more short-lived entries turning stale quickly, it should be possible to achieve even greater reductions in memory footprint.

6
Future Work

In this chapter we describe areas that we have not covered during the course of this thesis, or which we consider worth revisiting, and into which future research could be conducted.

6.1 Data Structure Design Choices

The evaluation of *PREDSL* revealed that the design choice of how to write dictionary and set modifications to a log was not optimal, as it prevented the use of *merging preemptive rewriting* strategies with these data structures. These strategies turned out to perform the best with regards to reducing a log's memory footprint, while reducing the amount of additional data written to the log. Thus, not having these strategies available for dictionaries and sets is a significant limitation. In future work, the design of how dictionaries and sets are represented in *PREDSL* should be revisited and redesigned in a way that makes them eligible for merging rewrites.

6.2 Robustness

Another limitation on the current implementation of *PREDSL* is that it currently does not concern itself with the handling of malformed log entries. Some malformed entries may create data structures in unintended states, e.g. it would currently be possible to join the last entry of a linked list back to its head, creating a ring, which, if iterated, would loop indefinitely. Other malformed entries may cause *PREDSL* to simply crash. In order to increase robustness, data structures in *PREDSL* would need to clearly define what constitutes a valid modification, and how to handle non-compliant entries. It may also be in the interest of a log implementation to either entirely prevent appending malformed log entries, or at least handle such entries in a well-defined manner.

6.3 Network Replication & Cryptographic Integrity

As we were mainly concerned with the performance of different rewriting strategies, we only implemented a single type of log. This log keeps its entries entirely in ephemeral

memory, and replication happens through access to other log instances within the same process' memory space. Because of this local nature, we designed the current in-memory log implementation to produce naïve logs (see Section 2.3) not authenticated in any way: Neither are there any internal integrity checks by forming a hash chain between subsequent log entries, nor does the log producer create cryptographic signatures for its entries. In future work, log implementations that provide both cryptographic integrity and replication over networks could be explored. One such implementation could be based on e.g. Secure Scuttlebutt (see 2.2.2), where log entries are both signed and arranged in a hash chain, and replication from the log producer to consumers happens through a peer-to-peer mesh network.

Bibliography

[1] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yan. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2:77–89, 2012. ISSN 2190-8516. doi: 10.1007/s13389-012-0027-1. URL https://doi.org/10.1007/s13389-012-0027-1.

[2] Ahto Buldas, Peeter Laud, Helger Lipmaa, and Jan Villemson. Time-stamping with binary linking schemes. In *Advances in Cryptology — CRYPTO '98*, pages 486–501. Springer, Heidelberg, 1998. ISBN 978-3-540-68462-6. doi: 10.1007/BFb0055749. URL https://doi.org/10.1007/BFb0055749.

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. ISBN 978-0-262-03384-8. URL http://mitpress.mit.edu/books/introduction-algorithms.

[4] Ross S. Finlayson and David R. Cheriton. Log files: An extended file service exploiting write-once storage. Technical Report STAN-CS-87-1177, Stanford University, 1987.

[5] Van Jacobson, Diana K. Smetters, James D. Thornton, Michael F. Plass, Nicholas H. Briggs, and Rebecca L. Braynard. Networking named content. In *CoNEXT '09*, CoNEXT '09, page 1–12, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605586366. doi: 10.1145/1658939.1658941. URL https://doi.org/10.1145/1658939.1658941.

[6] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., USA, 1998. ISBN 0201896850.

[7] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, may 1998. ISSN 0734-2071. doi: 10.1145/279227.279229. URL https://doi.org/10.1145/279227.279229.

[8] Alojscha Meyer. Replicated single-writer append-only logs. Unpublished manuscript, 2020.

[9] Alojscha Meyer, Christian Tschudin, and Erick Lavoie. Efficient validation of partial append-only logs for storage-less log production and consumption. Unpublished manuscript, 2020.

[10] Petru Nicolaescu, Kevin Jahns, Michael Derntl, and Ralf Klamma. Near real-time peer-to-peer shared editing on extensible data types. In *Proceedings of the 19th International*

Conference on Supporting Group Work, GROUP '16, page 39–49, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342766. doi: 10.1145/2957276.2957310. URL https://doi.org/10.1145/2957276.2957310.

[11] Maxwell Ogden, Karissa McKelvey, and Mathias Buss Madsen. Dat - Distributed Dataset Synchronization And Versioning. *OSF Preprints*, 2017. doi: 10.31219/osf.io/nsv2c. URL https://doi.org/10.31219/osf.io/nsv2c.

[12] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, page 305–320, USA, 2014. USENIX Association. ISBN 9781931971102. doi: 10.5555/2643634.2643666. URL https://dl.acm.org/doi/10.5555/2643634.2643666.

[13] Arik Rinberg, Tomer Solomon, Roee Shlomo, Guy Khazma, Gal Lushi, Idit Keidar, and Paula Ta-Shma. Dson: Json crdt using delta-mutations for document stores. *Proc. VLDB Endow.*, 15(5):1053–1065, may 2022. ISSN 2150-8097. doi: 10.14778/3510397.3510403. URL https://doi.org/10.14778/3510397.3510403.

[14] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, feb 1992. ISSN 0734-2071. doi: 10.1145/146941.146943. URL https://doi.org/10.1145/146941.146943.

[15] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In Xavier Défago, Franck Petit, and Vincent Villain, editors, *Stabilization, Safety, and Security of Distributed Systems*, pages 386–400, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-24550-3. doi: 10.1007/978-3-642-24550-3_29. URL https://doi.org/10.1007/978-3-642-24550-3_29.

[16] Dominic Tarr, Erick Lavoie, Aljoscha Meyer, and Christian Tschudin. Secure scuttlebutt: An identity-centric protocol for subjective and decentralized applications. In *Proceedings of the 6th ACM Conference on Information-Centric Networking*, ICN '19, page 1–11, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450369701. doi: 10.1145/3357150.3357396. URL https://doi.org/10.1145/3357150.3357396.