LazyLog: A New Shared Log Abstraction and Design for Modern Low-Latency Applications

XUHAO LUO, *University of Illinois Urbana-Champaign*, USA
SHREESHA G. BHAT[*], *University of Illinois Urbana-Champaign*, USA
JIYU HU[*], *University of Illinois Urbana-Champaign*, USA
RAMNATTHAN ALAGAPPAN, *University of Illinois Urbana-Champaign*, USA
AISHWARYA GANESAN, *University of Illinois Urbana-Champaign*, USA

ACM Reference Format:
Xuhao Luo, Shreesha G. Bhat, Jiyu Hu, Ramnatthan Alagappan, and Aishwarya Ganesan. 2025. LazyLog: A New Shared Log Abstraction and Design for Modern Low-Latency Applications. *ACM Trans. Comput. Syst.* 1, 1, Article 1 (August 2025), 32 pages. https://doi.org/10.1145/3750445

Abstract. Shared logs offer linearizable total order across storage shards. However, they enforce this order *eagerly* upon ingestion, leading to high latencies. We observe that in many modern shared-log applications, while linearizable ordering is necessary, it is *not required eagerly* when ingesting data but *only later* when data is consumed. Further, readers are naturally decoupled in time from writers in these applications. Based on this insight, we propose LazyLog, a novel shared log abstraction. LazyLog *lazily binds* records (across shards) to linearizable global positions and enforces this before a log position can be read. Such lazy ordering enables low ingestion latencies. Given the time decoupling, LazyLog can establish the order well before reads arrive, minimizing overhead upon reads. We build two LazyLog systems that provide linearizable total order across shards. Our experiments show that LazyLog systems deliver significantly lower latencies than conventional, eager-ordering shared logs.

1 Introduction

Shared logs [35, 36, 38, 41] have emerged as a crucial building block for datacenter applications. At its core, a shared log is a fault-tolerant, ordered sequence of records that many clients can simultaneously operate on. The shared log exposes a simple interface to applications. Applications ingest records via an append API, upon which they are linearizably [55] ordered and durably stored. Applications retrieve data via a read API, which takes a position and returns the record at that position.

This simple interface and the powerful abstraction make shared logs useful in a variety of modern applications.

This paper, originally titled *LazyLog: A New Shared Log Abstraction for Low-Latency Applications*, was invited by ACM TOCS for extended publication through recommendation by the Chairs of SOSP 2024.

[*]equal contribution

Authors' Contact Information: Xuhao Luo, xuhaol2@illinois.edu, *University of Illinois Urbana-Champaign*, USA; Shreesha G. Bhat, sgbhat3@illinois.edu, *University of Illinois Urbana-Champaign*, USA; Jiyu Hu, jiyuhu2@illinois.edu, *University of Illinois Urbana-Champaign*, USA; Ramnatthan Alagappan, ramn@illinois.edu, *University of Illinois Urbana-Champaign*, USA; Aishwarya Ganesan, aganesn2@illinois.edu, *University of Illinois Urbana-Champaign*, USA.

ACM 1557-7333/2025/8-ART1
https://doi.org/10.1145/3750445

For instance, shared logs are used to record and analyze web accesses [8, 41], build distributed databases [53], log events for postmortem debugging [8, 90], communicate between microservices [10], journal state for fault-tolerance [58], and stream data [15, 52, 89].

Unfortunately, today's shared logs incur high latencies (§2). The key problem is that they *eagerly order* records in the critical path *before* acknowledging appends. That is, by the time an append completes, the record is *eagerly bound to a position* in the shared log. Many shared logs [1, 12, 36, 41, 51, 57] store records on multiple storage shards and provide a linearizable total order across the shards. Thus, to bind records to positions, global coordination across the shards is necessary, which leads to high latencies. For example, in Scalog, upon appends, records are first stored and locally ordered within the shards. Then, after batching many records, the shards coordinate with a global ordering layer to bind records to global positions, after which appends are acknowledged. Thus, ingestion incurs many roundtrips and batching delays. Scalog reports append latencies of 1-2 ms, even in low-throughput regimes [41]. Corfu's [36] append path differs from that of Scalog, but fundamentally, it also eagerly orders records upon ingestion, incurring high latency.

Low-latency ingestion, however, is critical for many real-world shared-log applications. For instance, databases built atop shared logs require quick logging for updates [5, 53]; similarly, high-availability journals [58] built atop shared logs need low-latency ingestion. More broadly, in a recent survey by RedPanda [18, 19], a third of 300 practitioners rated ingestion latency as the most critical latency metric in their shared log deployments. Today's shared logs, due to their high ingestion latencies, cannot satisfy the demands of these applications. Given that such high latency is rooted in eager ordering, this paper asks: Can a shared log avoid eager ordering, yet also provide the linearizable ordering guarantee that applications require from shared logs?

Insight. At first sight, it may seem like one cannot avoid establishing the order eagerly, before acknowledging appends. However, we observe that in many modern shared-log applications, while linearizable order is necessary, it is not required right away upon ingestion but only later during reads; further, readers and writers in these applications are naturally decoupled in time. This allows a shared log *to establish the order in the background after acknowledging appends but before reads arrive.* Consider distributed databases built atop shared logs that separate readers from writers [5, 53, 74]. While readers in these databases must process updates in linearizable order [53], updates need not be eagerly ordered when writers log them. Further, the readers in such databases consume updates at *their own pace* [53], much later than when updates are logged. We identify several real-world applications (§3.1), including activity logging [59], event-sourcing [75], message queues [14], journaling [58], and log-aggregation [90], where linearizable order is required but not eagerly upon ingestion, and reads are naturally decoupled in time from writes.

LazyLog Abstraction. Based on the above insight, we propose LazyLog, a novel shared log abstraction. LazyLog makes a small yet powerful change to the shared log interface: in LazyLog, appending a record does *not* eagerly bind it to a log position; it only provides durability and a guarantee that the record will be eventually bound to its correct position that respects linearizability [55]. While LazyLog binds records to positions lazily, it enforces this binding before the log positions can be read. Lazy binding hides the overhead to establish global order across shards (and the local order within shards), reducing ingestion latency. However, LazyLog preserves the ordering guarantees of the conventional shared log abstraction and enforces the correct order before reads.

Given that reads are decoupled in time from writes in many shared-log applications, LazyLog can comfortably establish the order in the background before reads arrive; thus, reads do not incur overhead. Some applications, however, can read records immediately after appends and thus incur overhead. However, even when many reads take such a slow path, LazyLog would preserve the overall performance of conventional shared logs: while conventional logs incur ordering cost upon appends, LazyLog shifts this cost to reads.

LazyLog is inspired by the general idea of deferring work until needed, which has been explored in different contexts [46, 77, 78]. Skyros [49] applies this idea to defer ordering required for replication within *a single shard*. LazyLog, like Skyros, defers shard-internal ordering, but critically, it also defers *global ordering across shards*, a key cause of high latencies in shared logs. Occult [73] enforces order upon reads across shards. However, it only provides *causal* ordering across shards, a weaker model than linearizability that LazyLog provides. Our work is the first to build a shared log that offers linearizable order across shards with low latency by deferring ordering until needed. This end is enabled by our new observations about modern shared-log applications.

LazyLog Systems. We build two systems that implement the LazyLog abstraction: Erwin-bb (black-box) and Erwin-st (scalable throughput). Both offer linearizable total order across multiple shards. However, unlike conventional shared logs, they lazily bind records to shared log positions. Further, they avoid the cost for local ordering within a shard in the critical path that impacts existing systems like Scalog. Thus, appends in LazyLog systems complete in 1RTT.

The main challenge is to establish the linearizable order after appends have been acknowledged. The key idea to solve this, in both systems, is to write enough information about the records on a *fault-tolerant sequencing layer*, using which the order can be established in the background. Clients write this information to the sequencing-layer replicas without coordination in 1RTT. The high-level intuition is that if an append b follows another append a in real time, then b's information will naturally appear after a's in all sequencing replicas, while only information of concurrent appends will appear in different orders. Thus, despite sequencing-replica failures, records can be bound to their linearizable positions.

Erwin-bb (§4) aims to work with *unmodified* shards (e.g., Kafka shards or standard primary-backup shards). If clients write to such unmodified shards, they will incur the overhead to order within the shard, preventing 1RTT appends. Records must also be globally ordered, further increasing latency. Erwin-bb hides both these overheads by writing the records in the critical path only to the sequencing layer, which then orders and pushes the records to shards in the background. This design allows one to bolt-on Erwin-bb's sequencing layer atop existing per-shard-order systems like Kafka and achieve low-latency total order across shards. Erwin-bb offers high throughput for small records; however, since records pass through the sequencing layer, it quickly becomes the bottleneck for bigger (e.g., 4KB) records.

Erwin-st (§5) alleviates this bottleneck by writing *only metadata* that identifies the records to the sequencing layer and the actual records directly to the shards. For low latency, Erwin-st writes data and metadata in parallel. Internally, the metadata writes happen without coordination. For data writes, if unmodified shards are used, the writes would see the shard-internal ordering overhead. To avoid this, Erwin-st modifies the shards: Erwin-st realizes that since the information (i.e., the metadata) from the sequencing layer provides the correct order in the background, shards need to only provide durability for record-data in the critical path. Thus, clients perform the data writes to the shard replicas in parallel without coordination. Overall, all writes – the metadata to sequencing replicas and the data to the shard replicas – are done without coordination, completing appends in 1RTT.

The two versions show that the LazyLog abstraction can be implemented in disparate shared log architectures. The two versions are architecturally different because Erwin-bb uses Corfu-style position-to-shard mapping, while Erwin-st, like Scalog, allows clients to choose shards. As a result, like Corfu, Erwin-bb can spread data across shards evenly, while Erwin-st, like Scalog, can seamlessly add/remove shards.

Results. We show (§6) that when operating at the same throughput levels, LazyLog systems reduce append latencies by ~4× over our Corfu implementation and two orders of magnitude over open-source Scalog [17]. We run several experiments to show that reads rarely incur overhead in LazyLog systems. Erwin-bb offers ~1M small record appends/s, but its throughput flattens with big records. Erwin-st scales throughput with shards for big records with low latencies. We show Erwin-bb's black-box ability by enabling total order across off-the-shelf Kafka shards with low latency. We also demonstrate that Erwin-st can seamlessly add shards like Scalog and that the sequencing layer can be quickly reconfigured upon failures. We finally build three applications (key-value store, log aggregation, and journaled stream-processing), and demonstrate that LazyLog can deliver significant benefits for these end applications.

LazyLog systems are not without limitation. While Erwin-st can scale like Corfu, it cannot match the scalability level of Scalog. Scalog achieves scalability at the cost of latency [41]; Erwin-st trades off some scalability for low latencies. This trade-off suits many applications that need reasonably high throughput but at low latencies [18, 19].

Contributions. This paper makes four contributions.

- We make new observations about modern shared-log applications that present a new opportunity for a shared log to defer ordering, enabling low-latency ingestion.
- We present LazyLog, a novel shared log abstraction that builds upon this opportunity.
- We design two LazyLog systems, Erwin-bb and Erwin-st, that lazily establish global order, and avoid ordering cost within shards; our work is the first to offer linearizable total order in shared logs with low latency (specifically, 1RTT) by deferring ordering until needed. Our implementations are publicly available at `https://github.com/dassl-uiuc/LazyLog-Artifact`.
- We show the benefits of LazyLog systems via experiments.

2 Motivation

We explain how eager ordering in today's shared logs leads to high latencies. We then discuss the need for low-latency ingestion in modern applications.

2.1 Shared Logs: Background

The shared log offers a powerful abstraction with a simple interface [35–37, 41, 57]. Applications ingest records via an append API, upon which the shared log assigns positions for the records and stores them durably. Shared logs provide linearizable ordering [55]: if a record append B starts in real time after another record append A completes, then B is guaranteed to be ordered after A. Applications read records via read, which takes a position and returns the record at that position. checkTail finds the log tail and trim garbage collects a log prefix. Many applications like distributed databases [53], streaming [89], metadata stores [37], and state machine replication (SMR) [37] can be built using the above interface.

Shared logs have gained significant attention in research and practice alike. Prior research has built many shared logs [36, 37, 41, 57]. On the practical front, all cloud providers offer a shared log service (e.g., Kinesis [31], PubSub [54]); hyper-scalers use them for metadata [35]; open-source systems like Kafka [28] and others [12, 29, 83] offer the shared log functionality [41]. While some systems [28, 35, 83] only offer ordering within a shard, many practical and research implementations [1, 12, 36, 37, 41, 51, 57] offer total ordering where records across shards are linearizably ordered.

2.2 Eager Ordering Considered Harmful

Despite years of research and the ubiquity of shared logs, all existing shared logs today, unfortunately, suffer from high latencies. This high latency is rooted in the *eager-ordering* nature of shared logs. Upon an append, existing eager-ordering shared logs replicate records to storage servers within a shard. More importantly, they also determine the global position for the record and confirm that order, binding the record to that position. Both replicating the records and binding them incur coordination overhead, leading to high latencies.

Figure 1(a) shows Scalog's append path [41]. Clients first write records to the primary of a shard. The primary logs and replicates the records in FIFO order to its backup. Replication finishes when the backup logs the records (A in Figure 1(a)). At this point, the records are durable and locally ordered but their global position is yet to be determined. Periodically, all shard servers batch records and send their log lengths to a Paxos-based global

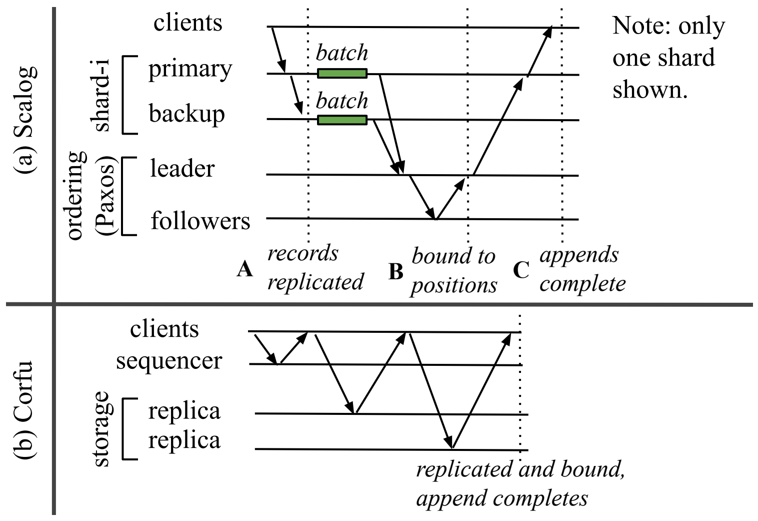


Fig. 1. **Append Path in Eager-Ordering Shared Logs.**

ordering layer. The ordering layer determines the log prefix that is durable, i.e., stored on both shard replicas. It establishes the global "cut", i.e., global order of durable records *across* shards, and makes this cut fault-tolerant (via Paxos). The records are now bound to global positions (B). The ordering layer sends the cut to the shard primaries, which then acknowledges appends of records for which global ordering has been established (C). Thus, an append sees the replication latency within the shard[‡], the batching delay, and the coordination latency in the ordering layer to bind records to positions. Scalog reports a mean append latency of 1-2 ms, even in low-throughput regimes.

Figure 1(b) shows Corfu's append path [36, 37]. A client first obtains a position from a sequencer. The client then writes the record to the storage servers responsible for the position via a client-driven chain protocol, where it updates the replicas serially one after the other. When the record is written at the tail of the chain, it is durable and is also bound to the position obtained. Thus, appends incur multiple RTTs, leading to high latencies. Note that merely getting a position from the sequencer does *not* bind the record to the position. Corfu's sequencer is merely an optimization [37, §2.2]; the record is bound to the obtained position only after the record has been written at the position on the storage servers.

Systems that offer per-shard ordering [28, 83] eagerly order as well. To reduce latency, these systems provide an option to finish appends after writing to one replica [7]. While this reduces latency (by 10× in Kafka), it leads to undesirable guarantees for applications: data could be lost upon failures.

2.3 Need For Low-Latency Ingestion

Low-latency ingestion is critical for many shared-log applications. For example, databases built atop shared logs need quick durability for updates [5, 53]. Similarly, high-availability journal [58] requires low-latency ingestion. Further, in a 2023 survey [18, 19], a third of 300 practitioners rated ingestion latency as the most critical latency metric in their shared log deployments. Today's shared logs, unfortunately, cannot satisfy the demands of these applications. Scalog's authors note this problem [41, §7]: Scalog doesn't serve applications that require low append latencies well.

Summary. Low-latency ingestion is critical for applications. Unfortunately, however, existing shared logs incur

[‡]Although the shard primary does not wait for the backup's response, Scalog still incurs latency to coordinate replication via the primary.

high latencies. Given that eager ordering is the cause for high latencies, we ask: *Can a shared log avoid eager ordering, yet also preserve the ordering guarantees of conventional shared logs?* We answer this question affirmatively in the next section.

3 LazyLog Insight and Abstraction

We explain our key insight and observations about modern shared-log applications, and how the LazyLog abstraction leverages the insight to realize low latencies.

3.1 Insight and Applications

Our key insight to avoid eager ordering is that although linearizable ordering is necessary, in many shared-log applications, it is not required right away upon appends but only later during reads. Specifically, many real-world applications do not require to know the indexes of the appended records immediately. Further, readers are naturally decoupled in time from writers in these applications. This offers a shared log an opportunity to defer ordering upon appends but establish it before reads arrive, reducing ingestion latencies without incurring overhead upon reads. We now present many real applications, where the above observation holds.

Distributed databases with decoupled readers [5, 53]. Modern distributed databases separate readers from writers to scale reads and writes independently, avoid fate sharing, and minimize performance interference [5, 53]. Shared logs ease this separation: writers ingest new updates to the log and readers independently process them from the log. In these databases, writers need to only achieve quick durability, while readers must process updates in a linearizable order [5]. For example, in Firescroll [5], a distributed database, writers expect to durably record updates to the shared log but do *not* require or use the indexes at which the records are appended [48]. Thus, the order need not be established eagerly when logging updates but only when readers consume them. Further, readers in these databases consume at “their own pace” [53], much later after updates are logged. Thus, a shared log can defer ordering upon appends, achieving low latency; it will also have ample time to establish the order before reads arrive.

Event sourcing [9, 13, 75]. With event sourcing, data is solely stored as a sequence of change events on a shared log, instead of storing the objects themselves and performing in-place updates on them. Downstream services enable queries by building views via replaying the events. The shared log itself also serves as audit trails [20]. While downstream services and audits must see the events in correct order, events need not be eagerly ordered when the data changes. Further, event sourcing systems [33, 62] adopt a popular software design pattern called command-query responsibility segregation or CQRS [32, 74, 88] that intends to avoid write-read interference, making readers typically lag behind writers.

Message queues [10, 14]. Components of an application communicate or queue work through a shared log [50, 65]. Messages and work items must be stored safely and delivered in the correct order. However, messages or items need not be eagerly bound to positions within the queue when senders add them. Further, consumers often are *time decoupled* from producers [42, 44]: “[messages are] consumed at a later time or at a much lower rate than it is produced” [42].

High-availability journal [58]. An application is made fault-tolerant by logging its state changes to a shared log; a fail-over instance can reload the changes and continue, should the application fail [58]. State changes must appear in linearizable order in the log so that correct state can be reconstructed. However, ordering needs to be enforced only when the log is read upon fail-over and not necessarily when recording state changes. For example, Samza [16], a stream-processing framework using Kafka for high-availability journaling, does not require the order when it performs checkpointing. Further, given that the journal is accessed only upon fail-over, there is a long gap between writes and reads.

```
// append to log; returns true if record is durable
bool append(record r);

// read 'len' records starting at 'from'
list<record> read(logpos_t from, uint64_t len);

// returns the number of durable records in the log
uint64_t checkTail();

// trim the log upto 'index'
bool trim(logpos_t index);

// append to log; returns the index of the record
logpos_t append_eager(record r);  // optional
```

Fig. 2. **LazyLog API.**

Activity logging [41, 59]. Applications log user activity to a shared log for analytics. For example, marketplaces [41] log product views and purchases for recommendations. The ordering of activities need not be established during logging and can be deferred until analytics engines process them. If the analytics jobs run in an offline fashion (e.g., every hour or so), reads will lag significantly behind writes. Even when the analytics jobs run alongside ingestion, they still lag behind to avoid interference between readers and writers [41]. For instance, in marketplaces [41, §5.1], to avoid interference, the writers append new records to active shards, while the analytics jobs read data from finalized shards.

Log aggregation [2, 86, 90]. Components of a distributed application record events to a shared log for postmortem debugging and analysis. While the log must ensure correct order of events to enable reliable debugging (e.g., after an incident), the position of events need not be determined eagerly during logging. For example, Log4j-Kafka [72], a Kafka-based logging framework, does not require the order during logging. Further, the records are accessed much later: after a failure or a performance anomaly that needs analysis.

Besides the above applications, other use cases such as ETL (extract, transform, load) pipelines [4] and resynchronization logs [60] also do not eagerly require order upon ingestion but only upon reads, and reads are decoupled in time from writes. In ETL pipelines, data from various sources such as sensors, web logs, social media, or IoT devices is ingested into a shared log. Records need to be ordered only when the stream processing engines consume and process the data from the shared log. In the resynchronization logs use case, commit records in a distributed system are written to a shared log so that upon failures, nodes can resync with the log to restore their data. Ordering is therefore unnecessary during appends but only required upon consumption, which happens during failure events.

3.2 The LazyLog Abstraction

Based on our observations, we propose the LazyLog abstraction. LazyLog does *not* eagerly bind a record to a position upon an append; it only makes the record durable immediately and provides a guarantee that the record will be eventually bound to its correct linearizable position. Although LazyLog binds records to positions lazily, it enforces the ordering before the positions can be read. Lazy binding enables LazyLog to avoid coordination in the critical path, reducing latency. However, it binds records to their correct positions before they can be consumed, preserving the linearizable ordering guarantee of the conventional shared log abstraction.

To end applications, LazyLog still provides the same abstraction of a fault-tolerant, linearizably ordered sequence of records, with the only change that the order is not eagerly determined upon appends. Figure 2 shows the LazyLog interface. An append makes the record durable, but it does *not* eagerly bind the record to a position. Thus, unlike the conventional interface, append does not return a position but only a flag denoting whether or

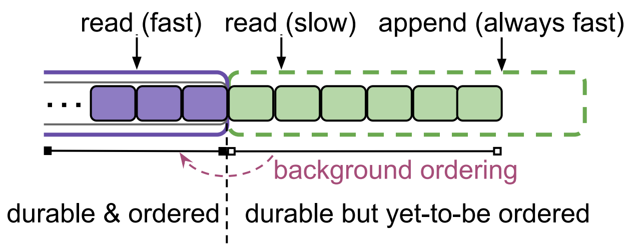


Fig. 3. **Lazily Ordered Log.** *The figure shows the ordered and unordered portions in LazyLog, and append and read latency characteristics.*

not the record was made durable. This modified interface suits real-world applications because, as we discussed above, these applications do not require or utilize the index (typically returned by conventional shared logs). The read, checkTail, and trim calls are identical to those of the conventional interface. Applications can invoke them in the same way as they usually would.

A LazyLog system must bind a record to a position before that position can be read. However, doing this on demand upon every read is inefficient. A practical system would thus keep ordering in the background. Thus, as shown in Figure 3, the lazily ordered log has two parts: one for which the order has been established and another for which the order is yet to be confirmed. If a read accesses the ordered portion, it is served quickly. Conversely, if a read accesses positions in the unordered portion, it takes a slow path: it is served after establishing the order at least up to the requested position.

However, reads predominantly take the fast path in the applications we discussed, given the gap between writes and reads. Thus, the system can comfortably order the records in the background before a position is read, avoiding or minimizing slow reads. Some applications, however, can (and do) immediately read after appending and thus incur overhead. For instance, SMR [36, 37] appends commands to the shared log and reads them back until the tail to apply all committed commands to the state machine; this can result in many slow reads. However, in practice, a LazyLog system can minimize this overhead. Specifically, if the records are ordered in large batches (in the background), then only the first read to the unordered portion would incur overhead; subsequent reads will be fast. Even in cases where batching opportunities do not exist, LazyLog would offer the same *overall* performance as a conventional shared log: while the latter incurs ordering cost upon appends, LazyLog would do so upon reads.

An Optional Eager-Ordering Append Interface. The applications we studied (in §3.1) do not require the positions of the appended records immediately and thus the lazy-ordering append interface suffices. However, some applications may need to know the positions of record upon appends. Consider an application that wants to efficiently implement read-your-writes consistency; this application might want to know the position of the record upon an append and store it in a secondary index. Without an eager-ordering append interface, subsequent readers from the same client must catch up to the tail of the log to provide read-your-writes consistency, which would be wasteful and unnecessary. For such applications, LazyLog systems implement an optional eager-ordering append interface, where the index of the appended record is returned upon the completion of the append.

Summary. In many applications, eager ordering upon ingestion is unnecessary; thus, LazyLog fits these applications and enables them to realize low-latency ingestion. Further, in these applications, reads are time decoupled from writes and thus LazyLog would not incur overhead on reads. For applications that read immediately after appends, a LazyLog system can minimize slow-path reads, and in the worst case, preserve the performance of an eager-ordering shared log.

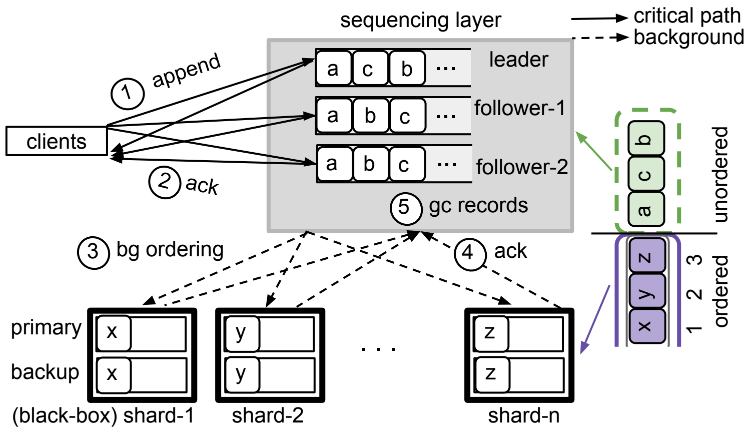


Fig. 4. **Erwin Architecture.** *x,y, and z have been ordered; a, b, and c are durable but unordered yet and thus reside in the sequencing layer.*

4 Erwin-bb Design

Our LazyLog systems offer linearizable total order across multiple shards. They do so with low latencies by only lazily binding records to global positions. Further, they avoid the cost for local ordering within a shard in the critical path, enabling 1RTT appends. This section describes how Erwin-bb achieves this goal and the next Erwin-st. For brevity, we will sometimes refer to Erwin-bb as simply Erwin. We first provide an overview and explain the append path (§4.1). We then explain how Erwin ensures and establishes linearizable order (§4.2, §4.3), and serves reads (§4.4). We then describe how Erwin handles failures correctly (§4.5). We then describe how Erwin can make progress even in the presence of failures and stragglers in the sequencing layer using supermajority quorums (§4.6). Finally, we describe the implementation of the optional eager-ordering append interface (§4.7).

4.1 Design Rationale, Overview, and Append Path

The practical benefit of Erwin-bb (over Erwin-st) is that it treats the shards as black boxes; internally, the shards could use any standard replication scheme like primary-backup [39] or Paxos [66, 70]; a shard could be even a per-shard-ordering shared log like Kafka (as we show in §6). This section describes how Erwin works with primary-backup shards.

Erwin requires shards to support the following operations: append an entry and read the entry at a specified index. Additionally, during view changes (which we explain later (§4.5)), a shard must be able to overwrite entries at the tail of the log portion that it stores. To change the tail, shards are not required to physically overwrite records; they only need to support a logical way to do so. For example, with Kafka shards, this can be achieved by deleting tail records and then appending new entries.

Figure 4 shows Erwin's architecture. Erwin has a set of unmodified primary-backup shards and a *sequencing layer*. If clients directly write records to such black-box shards in the critical path, then they will see the coordination to replicate within the shard, i.e., records will first be written to the shard primary, which would then replicate to the shard backup. In addition, records must be ordered across shards, increasing latencies further. Erwin avoids both these overheads by writing records only to the sequencing layer in the critical path. The sequencing layer is *fault-tolerant*: it contains a few replicas, usually $f+1$ to tolerate f failures. It is *coordination-free*: the replicas do not coordinate among them; clients write to the replicas in parallel without coordination.

Figure 4 shows Erwin's append path. Erwin clients directly write the records to the sequencing-layer replicas in parallel without any coordination (step-1). Erwin clients assign an unique record-id[*] to each record and piggyback it with the record data. Each replica appends incoming records to a local log and directly responds to clients. Once the client gets a response from all sequencing replicas, the append completes in 1RTT (step-2). The records are now durable, but their global positions are yet to be determined. Since there is no coordination, the records on the sequencing replicas can appear in different orders. However, as we explain in next subsection, the sequencing layer can construct the linearizable order despite this (even if f sequencing replicas fail). In the background, the sequencing layer establishes the linearizable order and pushes the records to the shards (step-3). The shards store the records and acknowledge (step-4). Once safe on the shards, the records are garbage collected on the sequencing replicas (step-5).

The Erwin shared log has two distinct parts for acknowledged records (as shown in Figure 4): a portion for which order is established and another for which order is yet to be determined. The ordered portion resides on the shards, while the yet-to-be-ordered portion on the sequencing layer.

The sequencing replicas provide only short-term durability: once records are safe on the shards, they are garbage collected from the sequencing layer. Thus, the amount of storage required in the sequencing layer is far less than the shards and so the records can be maintained in memory (on multiple replicas). In contrast, storage shards must provide long-term durability, requiring them to write eventually to the disk. Thus, the sequencing layer can run at much higher throughput than a single shard (whose performance is limited by the disk) and thus support multiple shards.

Erwin's approach of writing to multiple memories in the critical path for durability with eventually writing to disks is a standard practice in high-performance replicated systems [70, 80, 87, 95, 96]. For records that have been acknowledged to clients but yet to be flushed to disk, Erwin provides the same durability guarantee as those systems. Similar to those systems, Erwin can tolerate up to f simultaneous failures. In the unlikely case where more than f replicas fail simultaneously, Erwin correctly remains unavailable, preserving safety.

The main challenge is to correctly bind records in the yet-to-be-ordered portion to their linearizable positions in the background (even in the face of failures). Another challenge is to handle reads that may access either the ordered portion or the unordered portion of the log. The remainder of this section describes how Erwin solves these challenges.

4.2 Ensuring Linearizable Ordering

Because there is no cross-replica coordination in the sequencing layer, records from clients could appear in different orders across the replicas. For example, in Figure 4, records a, b, and c (which have been acknowledged) appear in different orders across the replicas. How does the sequencing layer then establish the correct linearizable order?

The main intuition is that if the append for a record b starts in real time after an append of another record a has completed, then it is guaranteed that b will appear after a in all the sequencing replica logs. This is true because when *append(a)* completes, a will be present on all logs. If *append(b)* starts after this, then b is guaranteed to appear after a in all logs. As a result, only records that are concurrently appended may (but not necessarily) appear in different orders across the logs. For example, in Figure 4, the actual real-time ordering is: *append(a)* completes and then *append(b)* and *append(c)* happen concurrently with each other. The sequencing logs capture this ordering correctly: a appears before b and c in all the logs, while b and c appear in different orders.

Erwin must order concurrently appended records in some way to produce a total order. For this purpose, Erwin treats one of the sequencing replicas as the leader and others as followers. The leader's log is used to establish the order in the failure-free case. For example, in Figure 4, positions up to 3 are ordered; thus, Erwin would try to bind a to position 4, c to 5, and b to 6 because this is the leader's order (although the followers have these records

[*]record-id is a combination of client-id and request-id.

in different orders).

Note that the leader's order *cannot* be exposed to clients until that order is finalized. This is because if the leader fails, then the records could be ordered differently. Upon a leader failure, Erwin chooses *any* one follower's log to assign order for unordered records. This is safe because all replica logs would respect the real-time dependencies and only concurrently appended records may appear in different orders; §4.5 expands on how Erwin handles failures.

4.3 Establishing the Order in the Background

Erwin establishes the order in the background. At a high level, the sequencing leader assigns records to positions (according to its local log) and pushes them to the appropriate shards. Erwin uses a deterministic function to map positions to shards similar to Corfu [36], where a shared log position p is assigned to shard $p \textit{ mod } n$, where n is the number of shards[†]. Each shard uses standard primary-backup to replicate the records. Once safe on the shards, the records are garbage collected at the sequencing replicas. For performance, Erwin does this background work in batches, ordering many records at once. We explain these steps below. For detailed background ordering process and pseudo code, please see Appendix A.

Since the sequencing leader's log provides the required ordering, the leader initiates the background ordering. Every sequencing replica maintains a counter called *last-ordered-gp*: the last global position in the log up to which the order has been established that the replica knows of. Periodically, the leader takes a batch of unordered records from its local log and assigns them to positions starting from its *last-ordered-gp*+1. It uses the deterministic function to map and push the records to appropriate shards. Each shard replicates its records and acknowledges the sequencing leader. Once all shards acknowledge, the sequencing leader garbage collects the records from its log and updates its *last-ordered-gp*. It then instructs and waits for the followers to garbage collect the records and update their *last-ordered-gp* [‡].

After this, the sequencing leader sets another counter called the *stable-gp* to its *last-ordered-gp*. Erwin maintains the following invariant with respect to the *stable-gp*: records for all positions up to the *stable-gp* are stable and will remain unchanged regardless of future failures. Intuitively, when the *stable-gp* is set, the binding for positions up to the *stable-gp* is complete. After this, the binding for the next batch starts. The sequencing leader then informs the shards of the *stable-gp* by piggybacking it with the next batch of records. The shards can then safely serve reads to positions up to *stable-gp*. Allowing reads only up to the *stable-gp* is critical to ensure correctness (as we soon discuss, §4.5).

If the sequencing leader fails, any follower's log can be used to determine the order for the unordered records. However, Erwin must maintain the *stable-gp* invariant, i.e., the order for the log portion that was previously stabilized (and thus could have been exposed to readers) does not change. We soon discuss how Erwin's recovery ensures this (§4.5).

4.4 Log Reads

Erwin clients submit reads to the shard servers. Clients use the deterministic mapping to find from which shard they must read a particular position p. Upon receiving a read, a shard server first checks if records up to p are stable (by checking if $p \leq$ *stable-gp*). If yes, the shard quickly serves the read; this is a fast-path read. Otherwise, the server waits until the log is stable at least up to p (i.e., *stable-gp* advances up to p) and then serves the record; this is a slow-path read. This check is critical because binding is complete only up to the *stable-gp*. The order for positions greater than *stable-gp* could change if the sequencing leader fails. For the same reason, the reads cannot

[†]Like Corfu, adding shards won't require moving existing records [36].

[‡]A subtle case is when a client request reaches a sequencing follower after the leader has informed to garbage collect that record. Erwin handles this request as a duplicate and filters it. Erwin uses the record-id of the request to filter duplicates.

be served from the sequencing leader.

Usually, applications keep track of the last read position and keep advancing this to read more records. Some applications require reading from the current position until the tail. Such applications invoke checkTail to know how many records are durable in the log and then read up to that point. Erwin serves the checkTail from the sequencing leader.

4.5 Failures, Views, and Reconfiguration

Failures within a shard are masked by standard techniques. Shards in Erwin use primary-backup; a Paxos/VR or Raft-based ensemble could also mask the failures within a shard.

However, Erwin must carefully handle failures in the sequencing layer. First, intermittent failures such as network blips are easily handled by having clients retry the operation until they are able to write to all sequencing replicas. If the retries result in duplicates, Erwin correctly filters them using record-ids. Second, Erwin handles failures such as crashes via *views* and a *reconfiguration* protocol. The sequencing layer operates in a series of monotonically increasing views. Upon a replica failure, the view advances and the system moves to a new configuration. Erwin does this in a sequence of steps: first, a control plane detects the failure, upon which it seals the current view; then, the unordered records in the sealed view are flushed to the shards; next, a new view starts with a new configuration and normal processing can resume.

Detection. Erwin detects sequencing replica failures using a standard technique used by many systems [57, 91]: a control plane that consists of a Zookeeper instance [30] and a controller. The controller itself is stateless and a new one can be started if the current one fails. We ensure that only one active controller exists using ZooKeeper. Each record-sequencing replica maintains a session with Zookeeper. A failure is detected when a replica's session with Zookeeper breaks and the controller is notified (via Zookeeper watches [21]).

Sealing the view. Once notified, the controller *seals* the old view to ensure that new records cannot be appended in that view. This sealing protocol resembles that of Delos [35] and Boki [57]. The controller sends a seal command to all the sequencing replicas. A sealed replica rejects new requests. Once a replica is sealed, new records cannot commit in that view. This is because a client waits for acknowledgments from all sequencing replicas in the same view.

Flushing unordered records. The controller then flushes unordered records in the sealed view. First, Erwin chooses *any* of the available sequencing replicas from the sealed view as the *recovery replica*. Then, the recovery replica flushes its log to the shards, assigning records to positions starting from its *last-ordered-gp*+1. This is the most critical step in the recovery; we now explain why this procedure is correct.

Correctness Sketch: Choosing *any* replica as the recovery replica will maintain durability of records committed in the old view; this is because records are replicated to *all* replicas during normal operation. The recovery replica may not contain some records that were part of the old leader (e.g., due to client failures); however, such records wouldn't have been acknowledged and thus need not be recovered. On the flip side, it is also possible that the recovery replica may contain records that were not part of the old leader; these wouldn't be acknowledged as well but it is harmless to recover them. Even if such unacknowledged records that were recovered are later retried by clients, they can be easily filtered out as duplicates using the record-ids.

Erwin must also ensure linearizable ordering of the recovered records. As we discussed, any two appends that have real-time order between them will appear in the correct order on all the sequencing replicas; only concurrent appends may appear in different orders. Thus, the recovery replica's log will correctly capture the real-time ordering dependencies. However, to guarantee linearizability, Erwin must ensure that any order that has been exposed to readers does not change. That is, it must maintain the *stable-gp* invariant.

Not maintaining the invariant violates linearizability. Consider the state in Figure 4. Suppose the sequencing leader tries to establish the order $[4 : a, 5 : c, 6 : b]$ by writing a, c, and b at log positions 4, 5, and 6, respectively on the shards. Assume an *incorrect* protocol where *stable-gp* is advanced *before* sequencing replicas garbage collect the records and set their *last-ordered-gp*. Now, suppose a client reads the order $[4 : a, 5 : c, 6 : b]$ from the shards. Suppose the leader now fails and follower-1 becomes the recovery replica. If follower-1's local order ($[a, b, c]$) is flushed starting at position 4 (*last-ordered-gp* +1), a subsequent reader will see an order ($[4 : a, 5 : b, 6 : c]$) inconsistent with the previous read.

Erwin prevents the above scenario by carefully orchestrating background ordering and reads (§4.2). Erwin allows reads to a position p only after ensuring that the order up to p will not change in the future. Erwin ensures this by advancing the *stable-gp* only after *all* sequencing replicas have garbage collected their records and advanced their *last-ordered-gp* [§].

Two cases are possible. (i) If the leader fails after *stable-gp* advances, then it is not possible for the recovery replica to change the order, ensuring correctness. In the above example, if the leader fails after *stable-gp* advances to 6, then the followers' logs would be empty and their *last-ordered-gp* would be 6. Thus, the order $[4 : a, 5 : c, 6 : b]$ established by the failed leader will prevail. (ii) If the leader fails before advancing *stable-gp*, the recovery replica may overwrite the order written by the old leader, but that is safe. In the example, suppose the leader fails before garbage collection on the followers and a recovery replica has $[a, b, c]$ with its *last-ordered-gp* as 3. The recovery replica will flush its records starting at position 4 (its *last-ordered-gp*+1), overwriting the old leader's order $[4 : a, 5 : c, 6 : b]$. However, this is safe because no client could have read any position greater than 3; this is because *stable-gp* could have been at most 3.

Starting a new view. Once the recovery replica's log has been flushed, the *last-ordered-gp* on all replicas are set appropriately and all local logs are cleared. Only replicas that have cleared the logs from the old view and set their *last-ordered-gp* will be part of the new configuration. Thus, any failed sequencing replica will be removed from the new configuration. Erwin can also add new replicas to the new configuration; new replicas will start with empty logs and *last-ordered-gp* set correctly. The new configuration is stamped with the new view number and written to Zookeeper; then, *stable-gp* is advanced and sent to the shards to update the local copy of *stable-gp* on the shards. Writing the new configuration before advancing the *stable-gp* prevents any partitioned replica from overwriting records potentially exposed if a failure happens during reconfiguration. The controller then sends a *StartView* message to replicas in the new configuration; the system can now accept new requests.

4.6 Tolerating Failures and Stragglers in the Sequencing Layer

While the detect-and-reconfigure approach described earlier safely and correctly excludes a failed sequencing replica, clients still experience a short period of unavailability before the reconfiguration completes in the presence of a sequencing replica failure. Furthermore, even in the absence of failures, by requiring clients to wait for *all* sequencing replicas to complete an append, Erwin's latency is determined by the slowest sequencing replica. Therefore, a single straggler replica could also degrade Erwin's performance. To address this issue, we designed a variant of Erwin, which uses a quorum-based approach like Skyros[49] to tolerate failures and straggler sequencing replicas. In this approach, with $2f + 1$ sequencing-layer replicas, for an append to be considered successful, clients wait only for a *supermajority* of $f + \lceil f/2 \rceil + 1$ acknowledgements including one from the sequencing leader. The Skyros protocol describes the recovery scheme and the correctness proof for this modified update protocol [49, §4] and we thus omit the details here.

[§]An alternate design is to advance *stable-gp* before garbage collection and have the new leader flush only entries that are not on the shards yet. However, updating *stable-gp* after garbage collection made the protocol simpler and the added time for garbage collection is anyway negligible.

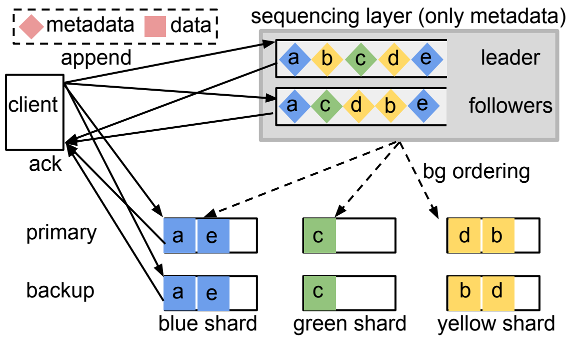


Fig. 5. **Erwin-st Architecture and Append Flow.**

4.7 Eager-Ordering Append

As we discussed in §3.2, a LazyLog system implements an optional eager-ordering append interface which returns the index of the appended record. One approach to implement the eager-ordering append is at the sequencing layer, by blocking the append until the *stable-gp* advances beyond the record's appended index. Instead, we opted to implement this interface at the client. To do so, the *tentative* log index of the durable-but-unordered record on the sequencing leader as well as the current view number are returned to the client. After the append successfully completes on all sequencing replicas, the client polls the sequencing leader through checkTail to retrieve the current *stable-gp* until the *stable-gp* advances beyond the tentative log index in the same view. At this point, the eager append completes and the log index is returned to the application. An error will be returned if the view number of the *stable-gp* is inconsistent with the view number returned by the append.

5 Erwin-st Design

Erwin-bb aims to work with unmodified shards. Thus, to avoid incurring shard-internal coordination, it funnels records through the sequencing layer. However, this has a downside: the sequencing layer can become the bottleneck. For small records (~100 bytes) which are common in practice [47], Erwin-bb can still offer high throughput. However, with bigger records (4KB [36, 41]), the sequencing layer can be quickly saturated, limiting throughput. Erwin-st solves this problem.

5.1 Main Idea and Overview

Erwin-st's main idea is to split a record into data and a piece of metadata that identifies the record. Clients then write record-data to the shards directly and only the metadata to the sequencing layer. With this design, the record-data does not pass through the sequencing layer, improving scalability.

To achieve low latency, clients write the data and metadata in parallel. The individual metadata writes to the sequencing replicas themselves are done in parallel as well (i.e., the sequencing layer is coordination-free). The metadata helps establish the total linearizable order in the background. However, data writes to shards will incur coordination latency within the shards if the shards use standard replication. To avoid this, Erwin-st modifies the shards. Erwin-st realizes that since the metadata from the sequencing layer provides the ordering information, shards need to only provide durability for record-data in the critical path. Thus, clients perform the record-data writes to the shard replicas in parallel without coordination, achieving durability in 1RTT.

In summary, by lazily sequencing only metadata, Erwin-st scales throughput even for bigger records. Further, by writing data to the shard replicas in parallel, Erwin-st achieves durability in 1RTT. By writing metadata in the same RTT, Erwin-st completes appends in 1RTT.

Figure 5 shows the architecture and flow of appends. A client directly writes a record to a shard of its choice (like Scalog). In parallel, it also writes the metadata, which is a tuple of <record-id, shard-id> to the sequencing replicas; record-id is a combination of client-id and request-id. In the example, clients have appended four records a, b, c, and d to the different shards. b and d have arrived in different orders on the yellow shard's primary and backup; the correct order will be determined by the sequencing layer. Now, a client is appending e; it writes the data to the blue-shard replicas, and the metadata to the sequencing replicas. Once all of them acknowledge, the append completes in 1RTT.

5.2 Background Ordering

The background ordering in Erwin-st is identical to Erwin-bb, except that only the metadata identifiers are sent to the shards and the data already exists on the shards. Periodically, the sequencing leader tries to establish the order of records by assigning metadata identifiers to positions; it then pushes the metadata along with the assigned positions to the shard primaries. Each shard primary then processes this information, and orders the records accordingly. For example, in Figure 5, the yellow-shard primary receives the ordering information $[1 : a, 2 : b, 3 : c, 4 : d, 5 : e]$, of which only $[b, d]$ concern the yellow shard. Then, the shard primary assigns and writes b to global position 2 and d to 4 (although it received them in a different order from clients). It then informs the shard backup to do the same. The shard primary then acknowledges the sequencing leader. Once all shards complete this process, the metadata identifiers are garbage collected from the sequencing layer and *stable-gp* is advanced, after which shards can serve reads up to the *stable-gp*.

5.3 Reads

In Erwin-bb, records are assigned to shards in a deterministic manner; thus, locating the shard to read a record from is straightforward. However, in Erwin-st, clients write the record to a target shard of their choice (similar to Scalog); a global position is later assigned for the record by the sequencer during background ordering. So, a client cannot directly determine the shard to read given a log position.

Erwin-st solves this problem by storing the metadata log that the sequencing leader sends during background ordering on the shards. A reading client can find the shard for a position by contacting any shard server and then perform the actual read at the target shard. Erwin-st amortizes this cost by having the clients fetch the position-to-shard mapping for many positions at a time and then cache it; for subsequent reads, clients look up their local cache to find the shard and read the record from there. Again, similar to Erwin-bb, a shard can serve reads only up to the *stable-gp*. If the *stable-gp* is not high enough, then the read takes a slow path waiting for *stable-gp* to advance at least up to the requested position.

5.4 Failure Handling and Correctness

Failures within a shard are handled by replacing the failed replica with a new one after copying both ordered and unordered records from a live node to the new one. Erwin-st handles sequencing replica failures in a way similar to Erwin-bb. In particular, it maintains the same invariant: order established up to *stable-gp* is guaranteed to remain unchanged. All the steps (detection, sealing, etc) are identical to Erwin-bb with the only difference that when flushing unordered records during reconfiguration, only metadata is flushed.

Additionally, Erwin-st must handle client failures that introduce two problems due to data-metadata separation. First, the sequencing leader receives the metadata, while the shard primary does not receive the data. Second, the shard primary receives the data but there is no corresponding metadata at the sequencing leader. The latter is not a serious issue: it just creates orphaned (uncommitted) records on the shards, which can be garbage collected via periodic scrubbing. In contrast, the former case needs more care. When the metadata reaches the shard during background ordering and the record is not present, the shard primary first waits for a timeout to receive the record from the client (in case this is due to a network delay). If it is a client failure, then the timeout will happen on the shard primary, upon which it sets the record to a special no-op record. The shard primary also instructs the backup to replace its record with a no-op. Clients ignore no-ops during reads. Setting to no-op is correct because the record would not have been acknowledged. The request may arrive after the no-op has been set. Erwin-st handles this correctly by rejecting the delayed request at the shard.

5.5 Limitations of LazyLog Systems

Our LazyLog implementations have one potential limitation. Erwin-st scales like Corfu: both systems scale with shards and are limited by the sequencing-layer. However, Erwin-st cannot provide the scalability level of Scalog. Scalog improves scalability by having shards contact the ordering layer in a batched manner. Such improved scalability is fundamentally at odds with low latencies. This is because, in Scalog, the shards must batch and contact the ordering layer in the critical path; deferring these steps to the background will violate append linearizability. While Scalog trades off latency, Erwin-st forgoes some scalability for low latencies (but is still as scalable as Corfu). This trade-off suits many applications that need reasonably high throughput but at lower latencies [18, 19]. Achieving Scalog's scalability with the low latencies of LazyLog remains an open challenge.

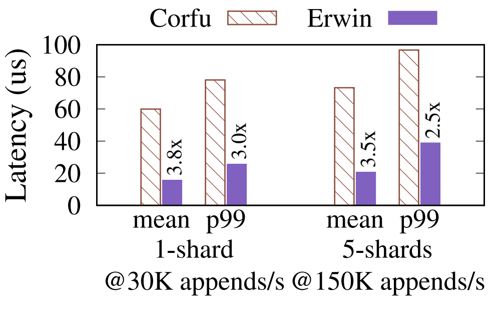
5.6 Implementation

Erwin code-base is mainly composed of a client library, sequencing layer, and storage shard, all of which are implemented in C++ (~8K LOC). Our code is publicly available [11]. The client uses eRPC [61] to issue requests to the sequencing layer and shards. On the sequencing layer, the log is implemented as a ring buffer with a head and tail pointer. New entries or metadata identifiers are added at the tail. For background ordering, we run a separate process that reads unordered log portion and pushes the entries or the metadata identifiers to the shards. For efficiency, our implementation runs this process separately and uses RDMA reads to access the ring buffer without interrupting the sequencing leader's CPU. To garbage collect, this process uses RDMA write to modify the head pointers on the sequencing replicas, freeing space in the ring buffer. Our shards use primary-backup replication. A shard stores its log portion across multiple files, each with a fixed number of entries. Thus, it can easily locate the target file to satisfy a read. Files are cached when read and thus subsequent reads are served from memory.

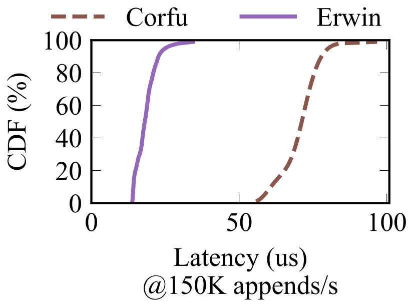
6 Evaluation

In our evaluation, we ask the following questions:

- What are the latency benefits of a lazy-ordering shared log compared to eagerly ordering shared logs? (§6.1)
- How do reads perform in a lazy-ordering shared log compared to an eagerly ordering one? (§6.2)
- How does periodically reading affect latencies? (§6.3)
- How does the append rate impact read latency? (§6.4)
- How does the eager-ordering append interface perform? (§6.5)
- Does the supermajority approach in the sequencing layer tolerate stragglers well? (§6.6)
- How does record size impact Erwin-bb's throughput? (§6.7)
- How well does Erwin-st scale compared to Erwin-bb? (§6.8)
- Does Erwin-st maintain the latency benefits of lazy ordering like Erwin-bb? (§6.9)
- How do reads perform in Erwin-st? (§6.10)
- Can Erwin-bb enable total order across existing per-shard-ordering shared logs with low latencies? (§6.11)
- Can Erwin-st seamlessly add and remove shards like Scalog? (§6.12)
- What is the impact of sequencing replica failures? (§6.13)
- Do end applications benefit from LazyLog? (§6.14)

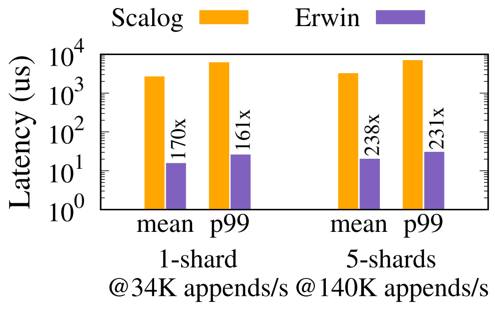


(a) Average and p99

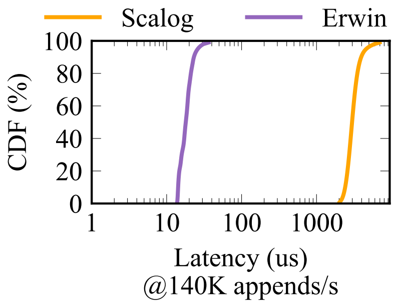


(b) CDF

Fig. 6. **Append Latency: Erwin vs. Corfu**



(a) Average and p99



(b) CDF

Fig. 7. **Append Latency: Erwin vs. Scalog**

Setup. We run our experiments on a xl170 [3] CloudLab [85] cluster. Each machine has an Intel 10-Core E5-2640v4 CPU, 64GB DRAM, a 25Gb Mellanox ConnectX-4 NIC, and a 480GB SATA SSD. We do not have access to many machines in this cluster and can run only five shards at a time. However, for the scaling experiments (§6.8), we use a different cluster, where we have more machines. Our sequencing layer has three replicas (one leader and two followers). Each storage shard has one primary and one or two backups. At places, we refer to Erwin-bb as Erwin for brevity.

6.1 Benefit of Lazy Ordering

We first demonstrate the benefit of lazy ordering by comparing Erwin against Corfu and Scalog. We implement Corfu from scratch. For Scalog, we use the publicly available artifact [17]. We run an append-only workload with 4KB records. We run Erwin and its competitors at the same throughput with each shard at about 30K appends/s, and compare the average and p99 latencies with one and five shards.

Comparison to Corfu. We run both Corfu and Erwin with three replicas within each shard. Figure 6 shows the mean and p99 latencies and the latency distribution. At the same throughput (shown in the bottom), Erwin reduces latencies significantly (by up to 3.8× for mean latency with one shard). This is because Corfu eagerly orders by first obtaining positions from the sequencer and then binding records to positions via the client-driven chain protocol, incurring 4RTTs with three replicas. Erwin, in contrast, completes appends in 1RTT by writing to the sequencing replicas without coordination and establishes the order in the background. This is the fundamental benefit of lazy ordering.

Comparison to Scalog. We next compare against Scalog. Since Scalog shards use one primary and one backup, we run our shards also with two replicas. Scalog's performance depends upon *interleaving interval* [41], which

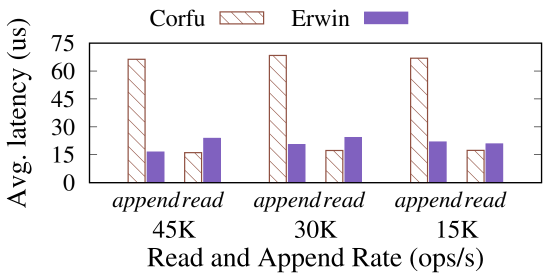


Fig. 8. **Reads Lagging Behind Appends.**

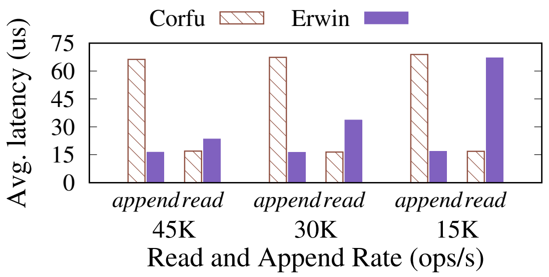


Fig. 9. **No Lag b/w Appends and Reads.**

determines how often shards contact the ordering layer. We set this to 0.1 ms as in the Scalog paper. For correct comparison, we run the Scalog and Erwin shards in a comparable performance regime. When writing to the shards in isolation (without the rest of the system involved), the performance of a shard are almost identical in the two systems: Scalog shard's latency is 693us at 34.3KOps/s throughput, while Erwin shard's latency is 772us at 32.3KOps/s throughput.

Figure 7 shows end-to-end append latencies. Erwin reduces mean and p99 latencies by two orders of magnitude. Scalog incurs high latency due to eager ordering: overhead for locally ordering within a shard, batching records before contacting the ordering layer, and global ordering. In contrast, although Erwin's shards have almost the same latency as Scalog shards, Erwin hides shard-internal coordination latency and also defers global ordering, offering low latency.

We note that there are implementation differences between Erwin and the Scalog artifact (e.g., Scalog uses gRPC [6], while Erwin uses eRPC [61]). Thus, the absolute latencies in a better Scalog implementation will be lower. However, even such an implementation will incur Scalog's fundamental overheads mentioned above. Thus, Erwin will offer significant latency benefits over such an implementation as well.

6.2 Read Latencies: Lazy vs. Eager Ordering

We now compare read latencies of Erwin and Corfu under two cases: (i) where reads *lag behind* appends, reflecting many applications in §3.1 (ii) where there is *no lag* between appends and reads, which is a bad case for Erwin. In both cases, appends and reads run at the same rate. We examine three different rates (15K, 30K, and 45K) and measure append and read latencies. We use one shard for 15K and 30K rates and two shards for 45K rate.

With Time Lag. Figure 8 shows the results when reads lag behind appends by a small window (3 ms) at different rates. At all rates, Erwin offers lower append latencies than Corfu as expected. Since reads lag behind appends, Erwin completes the ordering in the background by the time reads arrive. Thus, reads do not incur overhead: the read latency of Erwin approximates Corfu's. The small increase compared to Corfu is because reads contend with background writes (which happen in batches in Erwin) at the storage shards. Overall, Erwin offers significantly lower append latencies while providing almost the same read latencies as Corfu.

Without Lag. Figure 9 shows the result when there is no time lag between appends and reads; readers aggressively read as records are appended. Again, Erwin reduces append latencies. However, since there is no lag, reads in Erwin incur overhead: they see the cost of ordering. However, when the append rate is moderately high (45K), Erwin has batching opportunities: it can order many records in a big batch. Thus, only the first read that accesses the unordered log portion incurs overhead; subsequent reads are faster. Reads are thus only slightly slower than in Corfu. However, with fewer batching opportunities, more reads take the slow path. However, even in such scenarios, Erwin preserves the overall performance of Corfu: while Corfu eagerly orders and pays the overhead on appends, Erwin pays this cost upon reads.

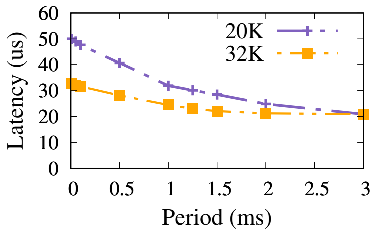


Fig. 10. **Periodicity vs Latency.**

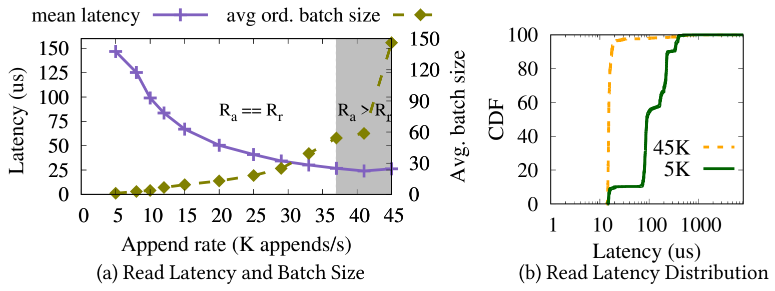


Fig. 11. **Append Rate vs. Read Latency.**

6.3 Performance With Periodic Reads

We now analyze how reads perform in applications that periodically read records up to the current tail. Here, the application periodically does a checkTail and then reads up to the obtained tail. We vary the period and measure read latencies. As shown in Figure 10, with 20K rate, for longer periods (e.g., 3 ms), latencies are very low. This is because many appends accumulate with longer periods (with only records near the tail being unordered). However, by the time the application reads the tail, the background ordering orders those unordered records. With short periods, not many appends accumulate between two consecutive checkTail-s, and therefore many reads take the slow path. A similar pattern holds for the 32K rate as well, but the latencies are lower because of more batching opportunities at this higher rate.

6.4 Impact of Append Rate

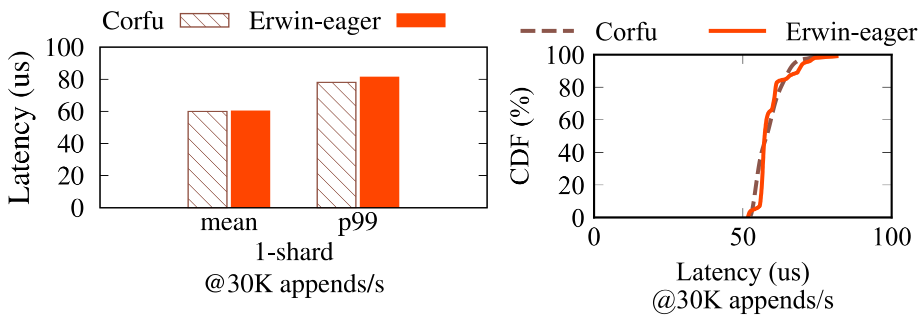
The previous two experiments showed that append rates (and background batch sizes) affect read latencies, which we now analyze further. To do so, we run reads alongside appends, and readers aggressively read whatever records are available. A single reader can run at 37K reads/s, beyond which the reader is unable to keep up with appends. If the append rate (R_a) is lower than this, the read rate (R_r) also matches the append rate. Thus, as shown in Figure 11(a), there are two regions: $R_a == R_r$ and $R_a > R_r$ (the gray region).

$\mathbf{R_a > R_r}$. In this region, reads run at a lower rate behind appends, mimicking message queues where consumers run at a lower rate than producers [42]. Here, by the time a log position is read, the records are already ordered. Thus, almost all reads take the fast path, resulting in low latencies.

$\mathbf{R_a == R_r}$. Here, reads catch up with appends. So, more reads can be slow. But, even in this unfavorable region, Erwin's latencies are low at high append rates. This is because with high append rates, the background-ordering batch sizes are larger (see right y-axis of 11(a)). When the rate is low (5K), the batch size is small, many reads take the slow path, resulting in high latencies. Figure 11(b) shows this: at 5K, almost all (91%) reads take the slow path (compared to all reads taking the fast path at 45K). The read latency at such low rates must ideally match the append latencies of an eager-ordering system (e.g., 70 μs that Corfu incurs in Figure 9). We see higher latencies than this because our background-ordering is optimized for bigger batches to improve throughput. However, this is not fundamental: our implementation could be modified to optimize for latency with small batches; thus, the absolute read latencies (in these worst cases) would be lower.

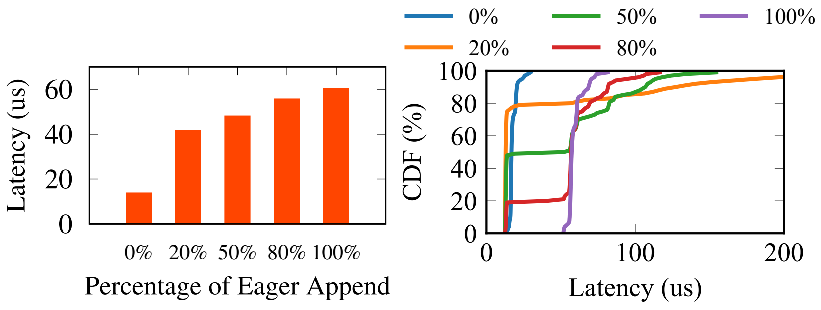
6.5 Eager-Ordering Append Performance

We compare the performance of the eager-ordering append interface in Erwin against Corfu. Figure 12 shows the mean latency, p99 latency, and the latency CDF when appending 4KB records. As shown, Erwin's eager-ordering append are nearly the same as Corfu's. This result indicates that even if some applications require eager ordering,



(a) Average and p99 (b) CDF

Fig. 12. **Append Latency: Erwin eager append vs. Corfu.**



(a) Average (b) CDF

Fig. 13. **Average Append Latency: Mixed Lazy and Eager Appends.**

they can use Erwin and obtain latencies similar as an eager-ordering shared log like Corfu.

Figure 13 shows the case where an application requires eager ordering only on certain appends. We vary the percentage of eager-ordering appends (from 0% to 100%) and plot the average append latency and latency CDF in Erwin. With lazy appends mixed with eager appends, the tail latency increases. This is because, an eager append that follows a bunch of lazy appends has to wait for all the lazyily-appended log records prior to it to get ordered. The higher the ratio of lazy appends, the more log records an eager append needs to wait behind on average, thus the longer the tail latency. However, the average latencies are still no worse than Corfu under different mix ratios from 0% to 100%.

6.6 Tolerating Straggler Sequencing Replicas with Supermajority Quorums

We now show that the supermajority-quorum-based append scheme tolerates straggler sequencing replicas. To do so, we run five sequencing replicas, where a supermajority is four replicas and therefore the supermajority-quorum scheme can tolerate one straggler replica. We simulate a straggler replica by injecting a network delay such that the append RPC to that replica experiences high latency. We measure the average and p99 append latencies of the supermajority-quorum version and the original append-to-all scheme with and without stragglers. As shown in Figure 14, in the normal case without any stragglers, the average and p99 latency of the quorum approach is nearly the same as the original append-to-all scheme. However, when there is a straggler, append-to-all suffers from high latencies because its latency is determined by the straggler, whereas the supermajority-quorum approach keeps the latency low as it does not have to wait for all replicas to respond.

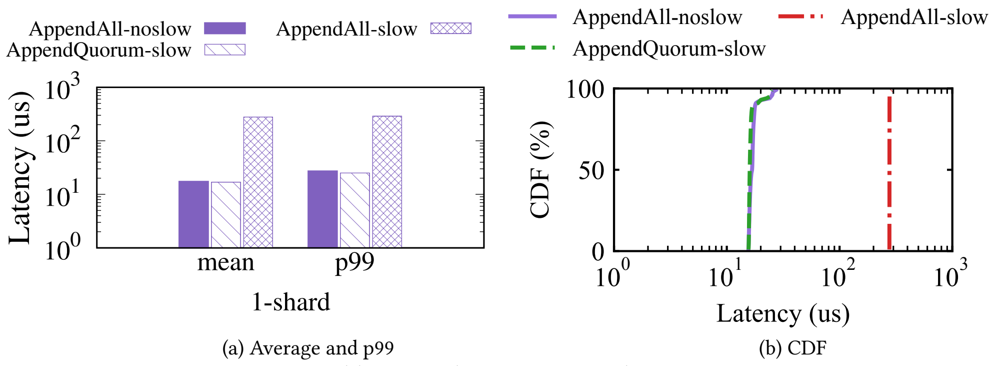


Fig. 14. **Erwin-bb Append Latency w/ and w/o Quorum**

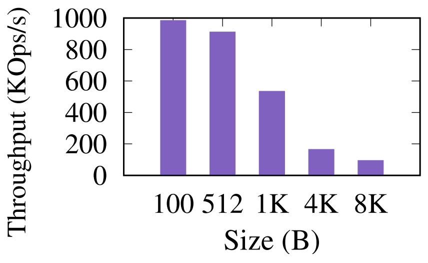


Fig. 15. **Record Size vs. Tput in Erwin-bb.**

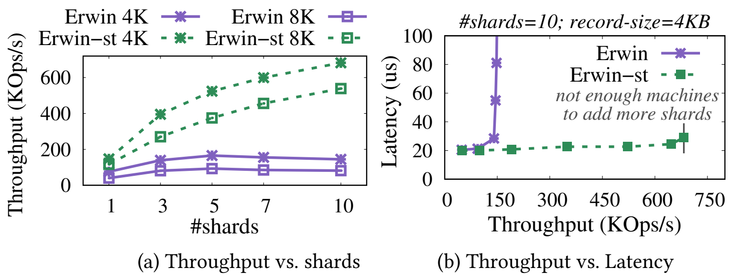


Fig. 16. **Scalable Throughput with Erwin-st.**

6.7 Erwin-bb: Record Size vs. Throughput

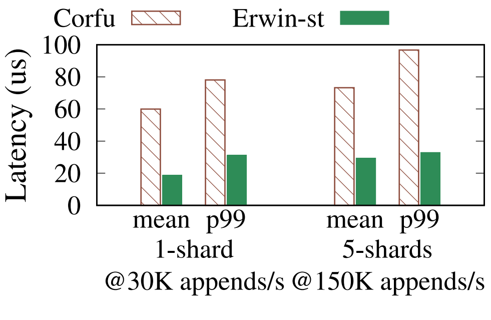
So far, we have measured latencies. We now measure the append throughput of Erwin-bb. As shown in Figure 15, with small records, Erwin-bb offers high throughput (~1M appends/s with 100-bytes); Erwin-bb can be useful in deployments that use small records [47]. However, because data itself passes through the sequencing layer, it quickly becomes the throughput bottleneck, limiting Erwin-bb's throughput with larger records. We next show how Erwin-st solves this.

6.8 Scalability of Erwin-st over Erwin-bb

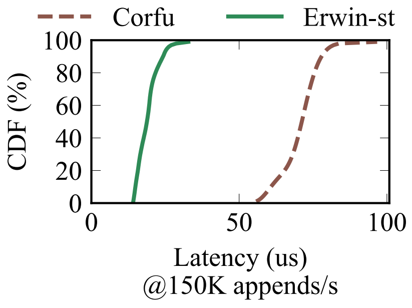
To measure scaling with more than five shards, we use a different (c6525-25g [3]) CloudLab cluster. As shown in Figure 16(a), Erwin-bb's throughput flattens quickly with large records. In contrast, by writing only metadata to the sequencing layer, even with large records, Erwin-st scales well. With 4KB records and 10 shards, it offers ~700K appends/s. Erwin-st can scale beyond this point; however, we do not have enough machines to run more than 10 shards. Erwin-st's throughput is limited by the sequencing layer (like Corfu). Our sequencing layer can run at 1.34M metadata-appends/s. More shards will enable Erwin-st to scale up to that point. Erwin-st achieves high throughput with low latencies as shown in Figure 16(b) because Erwin-st writes metadata and data without any coordination in 1RTT. For instance, at 700K appends/s, Erwin-st's latency is 29μs.

6.9 Erwin-st: Append Latency

We also compare the append latency of Erwin-st with Corfu and Scalog to demonstrate that the scalability improvement of Erwin-st does not hurt the latency benefit. We run the same 4KB append-only workload as §6.1 in 1-shard and 5-shards setting. Figure 17 and Figure 18 show that Erwin-st reduces average append latency by

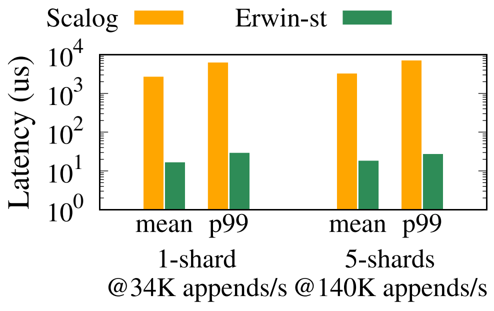


(a) Average and p99

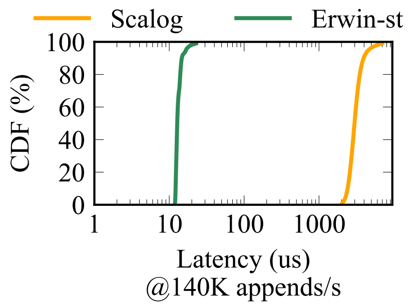


(b) CDF

Fig. 17. **Append Latency: Erwin-st vs. Corfu**



(a) Average and p99



(b) CDF

Fig. 18. **Append Latency: Erwin-st vs. Scalog**

up to 3.18× compared to Corfu and by two orders of magnitude compared to Scalog. The results indicate that Erwin-st maintains the append latency benefit as Erwin-bb.

6.10 Erwin-st Reads

We now analyze reads in Erwin-st. We compare read latencies with and without lag between appends and reads. Similar to §6.2, the appends and reads run at the same rate but the absolute rate is higher (200K). Figure 19 shows the result. First, when reads lag by 1s (lag-1s), no reads take the slow path, resulting in low latencies. Even in the no-lag case, very few reads are slow, making it only slightly worse than lag-1s. The absolute latencies are higher than in §6.2 because, here, we read 25 records at a time. When reading one record at a time, we notice that the latency of Erwin-st closely matches that of Erwin-bb (recall that Erwin-st clients cache the position-to-shard map to avoid a roundtrip (§5.3)).

6.11 Total Order across Kafka and Redpanda Shards

Erwin-bb enables total order at low latencies across per-shard-order off-the-shelf shared logs like Kafka and Redpanda. To demonstrate this, we run a 1KB append-only benchmark on stand-alone Kafka and Redpanda and Erwin-bb with Kafka and Redpanda as its shards. As shown in Figure 20 and Figure 21, with one shard, Erwin-bb reduces latency by three orders of magnitude. With three shards, Erwin-bb offers similar latency benefits while enabling linearizable total order.

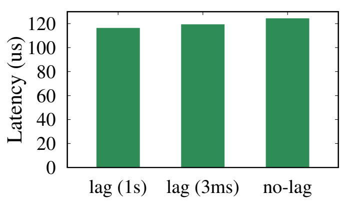


Fig. 19. **Reads in Erwin-st.**

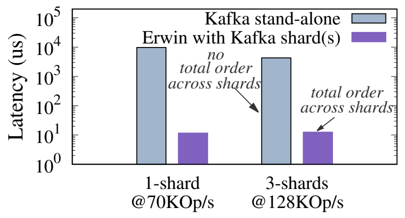


Fig. 20. **Erwin-bb: Total Order with Kafka Shards.**

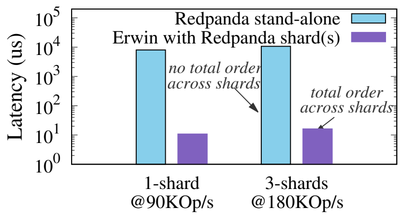


Fig. 21. **Erwin-bb: Total Order with Redpanda Shards.**

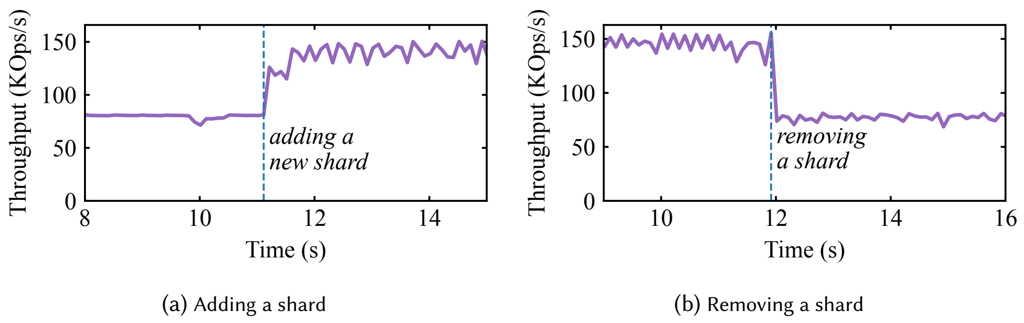


Fig. 22. **Erwin-st: Seamlessly Adding/Removing a Shard.**

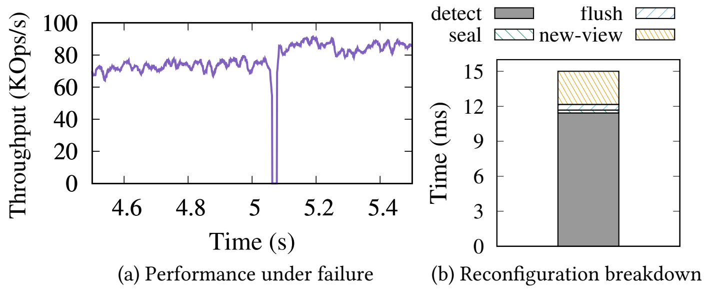


Fig. 23. **Reconfiguration in Erwin.**

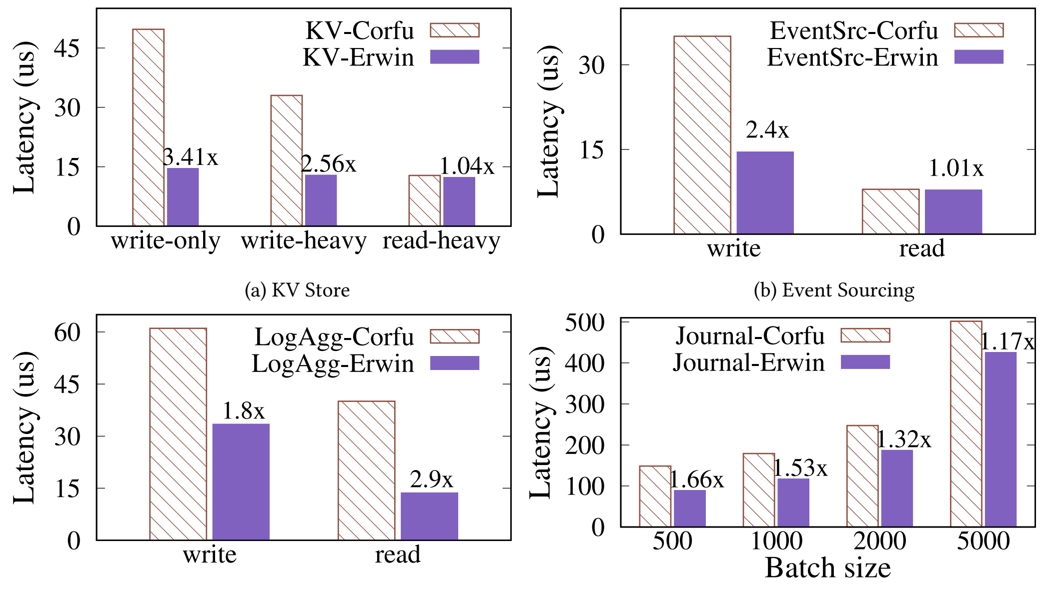
6.12 Seamlessly Adding and Removing Shards in Erwin-st

Scalog can seamlessly add/remove shards (unlike Corfu) [41]; this is enabled by allowing clients to choose shards instead of using a fixed position-to-shard mapping. Since Erwin-st also allows clients to choose shards, it can seamlessly add/remove shards as well. Figure 22(a) illustrates this: in the middle of a workload, we add a shard without downtime; clients start writing to the new shard, increasing the throughput.

Similarly, as shown in Figure 22(b), we can also remove a shard without downtime. Clients that originally write to the shard that will be removed will now write to other shards and the throughput will decrease accordingly.

6.13 Sequencing-Layer Reconfiguration

We next examine how quickly Erwin reconfigures after sequencing-replica failures. To do so, we crash a sequencing replica during a workload. As shown in Figure 23(a), the workload is impacted for a small period



(a) KV Store (b) Event Sourcing (c) Log Aggregation (d) Journaling

Fig. 24. **Applications.** *The figure shows Erwin's benefits for applications. (a) shows the average request latency in a shared-log-based KV store. (b) shows the average request latency in event sourcing. (c) shows the average transaction latency in log aggregation. (d) shows the average record-processing latency for a journaled, streaming word-count application.*

(~15 ms) after which it resumes. However, as shown in 23(b), a big portion of the reconfiguration time comes from failure detection and writing the new view's configuration, both of which involve ZooKeeper and suffer from its inefficiencies. The core recovery takes only 600 μs. Using a faster alternative to ZooKeeper could cut reconfiguration time to ~1 ms, which aligns with fail-over times for microsecond-scale applications [26].

6.14 Applications

We now demonstrate that end applications can benefit from LazyLog. To do so, we have built a writer-reader decoupled key-value (KV) store, a log-aggregation application, and a journaled stream-processing application.

These applications represent different points in the spectrum of the ratio between shared-log interaction and other computation the application performs to satisfy an end request. In the KV store, shared-log interaction is the most significant part in processing a user-level write request (like in real databases such as Firescroll [48]). In log-aggregation, the application performs other significant computation (such as processing a transaction) in addition to interacting with the shared log. Finally, in journaling, the computation can be much more significant than shared-log interaction.

KV Store. Modeled after Firescroll [5], we build a shared-log-based key-value store, where readers and writers are decoupled. The store supports put-s and get-s. Put-s are handled by write-processing servers, which receive and validate requests from end clients, serialize the KV pairs and append them to the shared log, and finally acknowledge clients. A set of read servers consume the log at their own pace, construct local state, and serve reads. In KV stores that decouple readers from writers, readers do not synchronize with the log before every read and thus reads are typically eventually consistent [34, 75]. Our store also follows the same design.

We run the store atop two shared logs: Corfu and Erwin. We use three YCSB [40] workloads: Load (write-only), YCSB-A (write-heavy: 50% updates, 50% reads), and YCSB-B (read-heavy: 5% updates, 95% reads). We configure the store with one writer and one reader server, and the underlying shared log with one shard (with three replicas). Keys are 24 bytes and values 1KB in size. Figure 24(a) plots the average KV request latency. For the write-only workload, all operations benefit from low latency; thus, Erwin offers the most benefit here: 3.4× lower latency than Corfu. With write-heavy workloads, the benefits are still considerable (~2.5×). With read-heavy workloads, Erwin does not offer much benefit because reads incur the same cost in Corfu and Erwin. Since the most significant part of a Put is appending to the shared log, Erwin offers the maximum benefit for this application.

Event Sourcing. We have also built an event sourcing application that processes banking transactions. End clients can perform operations like account creation, withdrawals, deposits, and transfers. Like other event-sourcing applications, this application stores all the above operations as events on the shared log, instead of directly storing the state. Similar to the KV store, this application also follows the CQRS pattern, where writers and readers are separated. Like the KV store, a set of write-processing servers handle end-client requests and record the events to the shared log. However, unlike the KV store, different query servers construct different views of the shared log over which they serve queries. For example, an anomaly detection server would look for anomalies by processing the transfer events, whereas a different query server could serve balance enquiries.

We run with one write server and two query servers (one serving balance enquiries and other transaction status). We run a workload with 50% writes and 50% reads and measure the average latency.

Figure 24(b) shows the average request latency for writes and reads. LazyLog achieves 2.4× lower latency for write requests as we do not need the shared log index in the command-execution path and the lower latency directly impacts client-perceived command execution latencies. Furthermore, the client queries are not tied to the shared log consumption and therefore are not impacted by the background ordering in LazyLog as the shared log consumption at the query servers happen in an asynchronous fashion.

Log Aggregation. We next build audit-logging for a transaction-processing application. The application allows clients to perform write operations like account creations, withdrawals, deposits, and transfers, and read operations like balance and transaction-status queries. The application shards accounts across a set of servers; each server processes transactions for accounts in its shard against a local database. Additionally, the servers also log information about transactions for audits to a shared log. Since audits are critical, logging happens synchronously [71]. Each transaction server uses a local RocksDB [45] instance to store data and run transactions.

We run a workload with 50% read transactions and 50% write transactions, and measure the average transaction latency. Note that irrespective of transaction type, operations on the underlying shared log is write-only. The shared log is read in this application only in an offline fashion, which our workload does not exercise. Figure 24(c) shows the result. As shown, Erwin offers latency benefits over Corfu for both application-level writes and reads. However, compared to the KV store, the benefits are smaller because this application performs transaction processing in addition to logging to shared log. The benefits vary depending on the type of operation. Specifically, the execution latency for writes is more significant than that of reads: writes incur ~23μs, while reads only take ~4μs; thus, the logging overhead for reads is much more significant. As a result, Erwin offers more benefit for logging read transactions than write transactions.

Journaling for Stream Processing. Finally, we have built a stream-processing word-count application, where the task workers use a shared log for checkpointing their state. In stream processing, checkpointing is a commonly used approach to provide fault-tolerance and exactly-once semantics [27, 63, 92]. In particular, before a task worker produces a record (e.g., for the next stage), it durably stores the produced state in a log. For example, Samza uses Kafka for this purpose [16, 63]. Should a task worker fail, it can use the log to recover its state without violating exactly-once semantics.

We run a word-count task with five workers. The workers process inputs and emit word counts. Before emitting, the workers durably store their state to the shared log. Stream-processing frameworks (like MillWheel [27]) do this for a batch of inputs. Our implementation also does the same. Similar to prior systems [27], we measure the latency for records to be processed and emitted. This latency internally consists of reading the record from an input source, processing it, checkpointing it to the shared log, and finally emitting.

Figure 24(d) shows the result. As shown, with big batches (5K), the fraction of time spent in logging compared to computation is smaller. Since Erwin optimizes only the checkpointing portion, the improvement with big batches is small (only 1.17× lower latency than with Corfu). However, with smaller batches, logging becomes a more significant portion and thus Erwin offers more benefits; for example, with a batch size of 500, Erwin offers 1.66× lower latency.

Applications Summary. LazyLog offers benefits to end applications by reducing the logging latency. The benefit varies based on portion of time spent in interacting with the shared log compared to the the overall execution required to satisfy an end-application request.

7 Related Work

Shared Logs. Corfu scales throughput with shards. Scalog improves over Corfu by providing more scalability and the ability to seamlessly add/remove shards. Erwin-st can scale like Corfu but cannot achieve the scalability level that Scalog achieves via batching, which is fundamentally at odds with low latency. Erwin-st forgoes some scalability for low latencies, a trade-off that suits many applications (§5.5). However, unlike Corfu, Erwin-st offers Scalog's ability to seamlessly add/remove shards (§6.12). Mason [56] is a recent system that has many similarities to Scalog, but it additionally supports the notion of multi-sequencing and service execution. However, it still eagerly orders records. Boki [57] and FlexLog [51] build shared logs for serverless computing. Boki's architecture resembles Scalog's and it introduces the idea of a metalog that simplifies reads compared to Scalog. However, Boki has the same ordering overheads, incurring high latencies. FlexLog avoids Paxos overhead in the ordering layer, but, it still eagerly orders and further assumes reliable broadcast, which requires coordination or programmable switches.

Kafka [28] and other systems [29, 83] offer linearizable order only within a shard and they incur high latency due to eager ordering. LogDevice [12] and DistributedLog [1] provide total order. LogDevice is similar to Corfu, but it uses a different data-placement policy [41]. DistributedLog forwards data via a single-writer. This has similarities to Erwin-bb; however, Erwin-bb makes records durable on the sequencing layer in 1RTT, and lazily establishes the order.

Defer Until Needed. LazyLog's idea to defer work until needed has similarities to deferring IO in local [78] and distributed [77] file systems, and execution in databases [46]. Skyros [49] and Occult [73] defer ordering until reads in distributed data stores. However, LazyLog differs from them in important ways. Skyros hides the coordination for replicating within a *single shard* by deferring ordering and execution of so-called nil-externalizing operations until reads. LazyLog systems avoid shard-internal overheads like Skyros, but importantly, they also hide the cost of global ordering *across* shards. Occult does not enforce ordering on writes but does so upon reads, and works with multiple shards. However, it only provides causal ordering across shards, a weaker model than linearizability that LazyLog provides.

Ordering. Consensus protocols like Paxos [66] and others [70, 79] can be used to order requests in 2RTTs. However, they offer a different interface than shared logs. Further, while they can order log entries in a shard, when the shared log *itself* is partitioned, these protocols cannot be used to establish a total order for log entries across shards. Speculation [64, 82, 94] and network ordering [69] provide 1RTT ordering for consensus but only within a shard. Eris [68] uses network ordering for multi-shard transactions but requires special hardware. Prior approaches that exploit commutativity [76, 81] need not wait for ordering (similar to LazyLog) when writes commute. However, log appends do not commute, so this approach does not work for shared logs. Kronos [43] is an event-ordering service that provides efficient ordering; however, it only provides a partial order.

Metadata Separation. Gnothi [93], a block store, separates the replication of data blocks and metadata; this separation has similarities to Erwin-st. However, unlike Gnothi, Erwin-st writes data and metadata in parallel without coordination and *lazily* sequences the metadata. Such data-metadata separation has been useful in storage systems as well [67, 84].

8 Conclusion

Today's shared logs eagerly order, leading to high latencies. We identify that in many shared-log applications, such eager ordering is unnecessary and order can be enforced later upon reads; further, reads are time decoupled from writes. LazyLog exploits this insight by deferring ordering and establishing it upon reads. Our work shows that linearizable total order across shards can be achieved in a shared log system with low ingestion latencies and little to no overhead upon reads.

Acknowledgments

We thank our SOSP '24 shepherd, SOSP '24 reviewers, and TOCS reviewers for their insightful comments. We greatly appreciate Mahesh Balakrishnan for his early feedback on the idea and the paper. We thank Vijay Chidambaram for his comments on the paper draft. We also thank the other DASSL members for their discussions. This work was funded by NSF CNS-2340218 and CNS-2339784, and an IIDAI grant.

References

[1] Apache DistributedLog. https://github.com/apache/distributedlog.
[2] Apache Log4j Kafka Appender. https://logging.apache.org/log4j/log4j-2.4/manual/appenders.html#KafkaAppender.
[3] Cloudlab Hardware. https://docs.cloudlab.us/hardware.html.
[4] ETL pipeline. https://redpanda.com/guides/kafka-tutorial/etl-pipeline.
[5] FireScroll. https://github.com/FireScroll/FireScroll.
[6] gRPC. https://grpc.io/.
[7] Kafka Configuration. https://kafka.apache.org/22/generated/producer_config.html.
[8] Kafka Use Cases. https://kafka.apache.org/uses.
[9] Kafka Use Cases - Event Sourcing. https://kafka.apache.org/uses#uses_eventsourcing.
[10] Kafka Use Cases - Messaging. https://kafka.apache.org/uses#uses_messaging.
[11] LazyLog Code Repository. https://github.com/dassl-uiuc/LazyLog-Artifact.
[12] LogDevice: distributed storage for sequential data. https://logdevice.io/.
[13] Pattern: Event sourcing. https://microservices.io/patterns/data/event-sourcing.html.
[14] RabbitMQ. https://www.rabbitmq.com/.
[15] Real-Time Analytics Explained. https://rockset.com/real-time-analytics-explained/.
[16] Samza - Kafka Checkpoint. https://github.com/apache/samza/blob/2eb556a5bcfb4aff83f3ba00fc221108d6cba0b2/samza-kafka/src/main/scala/org/apache/samza/checkpoint/kafka/KafkaCheckpointManager.scala/#L176-L177.
[17] Scalog Github Repository. https://github.com/scalog.
[18] The State of Streaming Data Report 2023-24. https://go.redpanda.com/state-of-streaming-data-report-2023-24.
[19] The State of Streaming Data Report 2023-24. https://anonymous.4open.science/r/2023datastreaming-survey-report-DCF1/.
[20] Why are CQRS and Event Sourcing good options for instant payments? https://iconsolutions.com/blog/cqrs-event-sourcing/.
[21] ZooKeeper Programmer's Guide. https://zookeeper.apache.org/doc/current/zookeeperProgrammers.html#sc_zkDataMode_watches.
[22] Farmington, Pennsylvania, October 2013.
[23] Boston, MA, February 2019.
[24] Banff, Canada, November 2020.
[25] Virtual, October 2021.

[26] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J. Marathe, Athanasios Xygkis, and Igor Zablotchi. Microsecond Consensus for Microsecond Applications. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI '20)* [24].

[27] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel: Fault-tolerant Stream Processing at iInternet Scale. *Proceedings of the VLDB Endowment*, 6(11), 2013.

[28] Apache. Kakfa. http://kafka.apache.org/.

[29] Apache. Pulsar. https://pulsar.apache.org/.

[30] Apache. ZooKeeper. https://zookeeper.apache.org/.

[31] AWS. Amazon Kinesis. https://aws.amazon.com/kinesis/.

[32] AWS. CQRS pattern. https://docs.aws.amazon.com/prescriptive-guidance/latest/modernization-data-persistence/cqrs-pattern.html.

[33] AWS. Event Sourcing Pattern. https://docs.aws.amazon.com/prescriptive-guidance/latest/modernization-data-persistence/service-per-team.html.

[34] AWS. Event Sourcing Pattern. https://docs.aws.amazon.com/prescriptive-guidance/latest/cloud-design-patterns/event-sourcing.html#:~:text=Eventual%20consistency%3A%20Data,the%20current%20state.

[35] Mahesh Balakrishnan, Jason Flinn, Chen Shen, Mihir Dharamshi, Ahmed Jafri, Xiao Shi, Santosh Ghosh, Hazem Hassan, Aaryaman Sagar, Rhed Shi, et al. Virtual Consensus in Delos. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI '20)* [24].

[36] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei, and John D. Davis. CORFU: A Shared Log Design for Flash Clusters. In *Proceedings of the 9th Symposium on Networked Systems Design and Implementation (NSDI '12)*, San Jose, CA, April 2012.

[37] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D Davis, Sriram Rao, Tao Zou, and Aviad Zuck. Tango: Distributed Data Structures over a Shared Log. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)* [22].

[38] Philip A Bernstein, Colin W Reid, and Sudipto Das. Hyder – A Transactional Record Manager for Shared Flash. In *CIDR*, volume 11, pages 9–20, 2011.

[39] Navin Budhiraja, Keith Marzullo, Fred B Schneider, and Sam Toueg. The Primary-backup Approach. *Distributed systems*, 2, 1993.

[40] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '10)*, Indianapolis, IN, June 2010.

[41] Cong Ding, David Chu, Evan Zhao, Xiang Li, Lorenzo Alvisi, and Robbert Van Renesse. Scalog: Seamless Reconfiguration and Total Order in a Scalable Shared Log. In *Proceedings of the 17th Symposium on Networked Systems Design and Implementation (NSDI '20)*, Santa Clara, CA, February 2020.

[42] Philippe Dobbelaere and Kyumars Sheykh Esmaili. Kafka versus rabbitmq: A comparative study of two industry reference publish/subscribe implementations: Industry paper. In *Proceedings of the 11th ACM international conference on distributed and event-based systems*, pages 227–238, 2017.

[43] Robert Escriva, Ayush Dubey, Bernard Wong, and Emin Gün Sirer. Kronos: The Design and Implementation of an Event Ordering Service. In *Proceedings of the EuroSys Conference (EuroSys '14)*, Amsterdam, The Netherlands, April 2014.

[44] Patrick Th Eugster, Pascal A Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM computing surveys (CSUR)*, 35(2):114–131, 2003.

[45] Facebook. RocksDB. http://rocksdb.org/.

[46] Jose M Faleiro, Alexander Thomson, and Daniel J Abadi. Lazy Evaluation of Transactions in Database Systems. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*, Snowbird, UT, June 2014.

[47] Eric Falk, Vijay K Gurbani, and Radu State. Query-able kafka: An agile data analytics pipeline for mobile wireless networks. *Proceedings of the VLDB Endowment*, 10(12), 2017.

[48] FireScroll. FireScroll - Source Code. https://github.com/FireScroll/FireScroll/blob/main/api/handler.go#L96.

[49] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Exploiting Nil-Externality for Fast Replicated Storage. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP '21)* [25].

[50] Jimena Garbarino. Communicate Between Microservices with Apache Kafka. https://developer.okta.com/blog/2022/09/15/kafka-microservices.

[51] Dimitra Giantsidi, Emmanouil Giortamis, Nathaniel Tornow, Florin Dinu, and Pramod Bhatotia. FlexLog: A Shared Log for Stateful Serverless Computing. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*, pages 195–209, 2023.

[52] Ken Goodhope, Joel Koshy, Jay Kreps, Neha Narkhede, Richard Park, Jun Rao, and Victor Yang Ye. Building linkedin's real-time activity data pipeline. *IEEE Data Eng. Bull.*, 35(2):33–45, 2012.

[53] Dan Goodman. Serving low-latency data in 32 regions with FireScroll and Redpanda. https://redpanda.com/blog/multi-region-deployment-redpanda-firescroll.
[54] Google. Pub/Sub. https://cloud.google.com/pubsub.
[55] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3), July 1990.
[56] Christopher Hodsdon, Theano Stavrinos, Ethan Katz-Bassett, and Wyatt Lloyd. MASON: Scalable, Contiguous Sequencing for Building Consistent Services. *Journal of Systems Research (JSys)*, 3(1), 2023.
[57] Zhipeng Jia and Emmett Witchel. Boki: Stateful Serverless Computing with Shared Logs. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP '21)* [25].
[58] Kafka. Kafka Documentation - Compaction. https://kafka.apache.org/documentation.html#compaction.
[59] Kafka. Kafka Documentation - Website Activity Tracking. https://kafka.apache.org/uses#uses_website.
[60] Kafka. Kafka Use Cases - Commit Logs. https://kafka.apache.org/uses#uses_commitlog/.
[61] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be general and fast. In *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI '19)* [23].
[62] Joe Karlsson. Event sourcing with Kafka. https://www.tinybird.co/blog-posts/event-sourcing-with-kafka.
[63] Martin Kleppmann and Jay Kreps. Kafka, Samza and the Unix Philosophy of Distributed Data. 2015.
[64] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine Fault Tolerance. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 45–58. ACM, 2007.
[65] ASC LAB. LAB Insurance Sales Portal - A Simplified Insurance Sales System. https://github.com/asc-lab/micronaut-microservices-poc/.
[66] Leslie Lamport. Paxos Made Simple. *ACM Sigact News*, 32(4):18–25, 2001.
[67] Lanyue Lu and Thanumalayan Sankaranarayana Pillai and Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, Santa Clara, CA, February 2016.
[68] Jialin Li, Ellis Michael, and Dan RK Ports. Eris: Coordination-free Consistent Transactions using In-Network Concurrency Control. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '17)*, Shanghai, China, October 2017.
[69] Jialin Li, Ellis Michael, Naveen Kr Sharma, Adriana Szekeres, and Dan RK Ports. Just Say No to Paxos Overhead: Replacing Consensus with Network Ordering. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI '16)*, Savannah, GA, November 2016.
[70] Barbara Liskov and James Cowling. Viewstamped Replication Revisited. 2012.
[71] Log4j. Log4j as An Audit Logging Framework. https://logging.apache.org/log4j/2.x/manual/async.html.
[72] Kafka Log4j. https://github.com/apache/kafka/blob/9db5c2481f8cefb5ec9a97d6e715a350dbc929c7/log4j-appender/src/main/java/org/apache/kafka/log4jappender/KafkaLog4jAppender.java#L359.
[73] Syed Akbar Mehdi, Cody Littley, Natacha Crooks, Lorenzo Alvisi, Nathan Bronson, and Wyatt Lloyd. I Can't Believe It's Not Causal! Scalable Causal Consistency with No Slowdown Cascades. In *Proceedings of the 14th Symposium on Networked Systems Design and Implementation (NSDI '17)*, Boston, MA, March 2017.
[74] Microsoft. CQRS pattern. https://learn.microsoft.com/en-us/azure/architecture/patterns/cqrs.
[75] Microsoft. Event Sourcing Pattern. https://learn.microsoft.com/en-us/azure/architecture/patterns/event-sourcing.
[76] Iulian Moraru, David G Andersen, and Michael Kaminsky. There is More Consensus in Egalitarian Parliaments. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)* [22].
[77] Edmund B Nightingale, Peter M Chen, and Jason Flinn. Speculative Execution in a Distributed File System. October 2005.
[78] Edmund B Nightingale, Kaushik Veeraraghavan, Peter M Chen, and Jason Flinn. Rethink the sync. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, WA, November 2006.
[79] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, Philadelphia, PA, June 2014.
[80] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, et al. The ramcloud storage system. *ACM Transactions on Computer Systems (TOCS)*, 33(3):7, 2015.
[81] Seo Jin Park and John Ousterhout. Exploiting Commutativity For Practical Fast Replication. In *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI '19)* [23].
[82] Dan RK Ports, Jialin Li, Vincent Liu, Naveen Kr Sharma, and Arvind Krishnamurthy. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *Proceedings of the 12th Symposium on Networked Systems Design and Implementation (NSDI '15)*, Oakland, CA, May 2015.
[83] RedPanda. RedPanda. https://redpanda.com/.

[84] Kai Ren, Qing Zheng, Swapnil Patil, and Garth Gibson. IndexFS: Scaling File System Metadata Performance with Stateless Caching and Bulk Insertion. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 237–248. IEEE, 2014.

[85] Robert Ricci, Eric Eide, and CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login:*, 39(6), 2014.

[86] Splunk. Log Aggregation: Everything You Need to Know for Aggregating Log Data. https://www.splunk.com/en_us/blog/learn/log-aggregation.html.

[87] Michael Stonebraker, Samuel Madden, Daniel J Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: It's time for a complete rewrite. In *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*, pages 463–489. 2018.

[88] Event Store. CQRS pattern. https://www.eventstore.com/cqrs-pattern.

[89] Gang Tao. Realizing low latency streaming analytics with Timeplus and Redpanda. https://redpanda.com/blog/low-latency-streaming-analytics-timeplus-redpanda.

[90] Nitish Tiwari. JVM-free centralized logging with Redpanda and Parseable. https://redpanda.com/blog/unify-log-data-parseable-redpanda.

[91] Robbert Van Renesse and Fred B Schneider. Chain Replication for Supporting High Throughput and Availability. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, San Francisco, CA, December 2004.

[92] Stephanie Wang, John Liagouris, Robert Nishihara, Philipp Moritz, Ujval Misra, Alexey Tumanov, and Ion Stoica. Lineage Stash: Fault Tolerance off the Critical Path. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*, Huntsville, ON, Canada, October 2019.

[93] Yang Wang, Lorenzo Alvisi, and Mike Dahlin. Gnothi: Separating Data and Metadata for Efficient and Available Storage Replication. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, Boston, MA, June 2012.

[94] Benjamin Wester, James A Cowling, Edmund B Nightingale, Peter M Chen, Jason Flinn, and Barbara Liskov. Tolerating Latency in Replicated State Machines Through Client Speculation. In *Proceedings of the 6th Symposium on Networked Systems Design and Implementation (NSDI '09)*, Boston, MA, April 2009.

[95] Irene Zhang, Naveen Kr Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan RK Ports. Building consistent transactions with inconsistent replication. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*, Monterey, California, October 2015.

[96] Irene Zhang, Naveen Kr Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan RK Ports. When is Operation Ordering Required in Replicated Transactional Storage? *IEEE Data Eng. Bull.*, 39(1):27–38, 2016.

A Detailed Background Ordering Process

Here we provide a detailed description of the background ordering process of Erwin. We describe the steps for sequencing leader, sequencing follower, and shard server separately, and provide pseudo code for each of them.

A.1 Sequencing Leader Process

Each time the sequencing leader takes a batch of records from its log from *last-ordered-gp* to the current log tail. It then pushes those records to the corresponding shard primaries based on a deterministic mapping function. It also piggybacks the current *stable-gp* with that batch of records.

After getting acknowledgements from all shard primaries it has sent the records to, the sequencing leader initiates the garbage collection. Garbage collecting records on the sequencing leader itself is straightforward by just setting the *last-ordered-gp* to the current log tail. Garbage collecting records on sequencing followers is a little more complicated because in sequencing followers' logs, records may not appear in the same order as the sequencing leader's log. For example, in Figure 4, if the sequencing leader garbage collects the first two records *a* and *c*, it can just set the *last-ordered-gp* to 2, but follower-1 cannot do the same because the first two records in its log are *a* and *b*. To correctly garbage collect records on sequencing followers, the sequencing leader sends the record-ids of the batch of records to the sequencing followers. We will describe the followers' process in the next section.

After all sequencing followers have completed the garbage collection, the sequencing leader can set its *stable-gp* to *last-ordered-gp*. Algorithm 1 shows the pseudo code for the sequencing leader process.

Algorithm 1 Sequencing Leader Process

```
while new records available do
    batch ⟵ log[last-ordered-gp+1, log-tail]
    record-ids ⟵ empty list
    for each record $r$ in batch do
        record-ids.append(r.id)
    end for
    send(batch, stable-gp) to shard primaries
    wait for acknowledgements from all shard primaries
    last-ordered-gp ⟵ log-tail
    for each follower $f$ in sequencing followesrs do
        send_record_ids(record-ids) to $f$
    end for
    wait for GC completion from all sequencing followers
    stable-gp ⟵ last-ordered-gp
end while
```

A.2 Sequencing Follower Process

Each sequencing follower maintains a map from the record-id to its position in the log. Upon receiving the record-ids from the sequencing leader, the follower first determines the maximum log index *max-index* of those record-ids. Then, it goes through its local log from *last-ordered-gp* to *max-index*. For each record, it checks if the record-id of that record is in the record-ids list. If it is, then that record is garbage collected and *last-ordered-gp* is advanced. For the records whose record-ids are not in the list, one way is to leave them in the log. However, that would create non-contiguous log entries, which is not desirable. For example, in Figure 4, garbage collecting *a* and *c* would leave *b* in follower-1's log. Instead, the follower appends them to a list called *pending-records*, and still advances the *last-ordered-gp*. In the end, *last-ordered-gp* will be advanced to *max-index*. In the next garbage collection, the follower will try to garbage collect records in *pending-records* before proceeding to the log.

The records in the *pending-records* are still considered part of the log as if they reside right before the record at *max-index*. In the case of a leader failure and the follower becomes the new leader as described in §4.5, the records in *pending-records* will also be flushed to the shard along with the unordered records in the log. Algorithm 2 shows the pseudo code for the sequencing follower process.

Algorithm 2 Sequencing Follower Process

```
Require: record_ids: list of record IDs received from leader
 1: for rid in record_ids do
 2:     max_index ← max(max_index, record_id_to_log_index[rid])      ▷ find max index of the record IDs
 3: end for
 4: for record r in pending-records do
 5:     if r.id in record_ids then
 6:         pending-records.remove(r)              ▷ garbage collect from pending-records first
 7:     end if
 8: end for
 9: for record r in log[last-ordered-gp+1, max-index] do
10:     if r.id not in record_ids then
11:         pending-records.append(r)              ▷ add to pending-records if not in record-ids
12:     end if
13: end for
14: last-ordered-gp ← max-index
15: send acknowledgement to sequencing leader
```

A.3 Shard Server Process

Each shard maintains a local variable *shard-stable-gp*, which is the shard's view of the *stable-gp* on the sequencing leader. When the shard server receives the records from the sequencing leader, it updates its *shard-stable-gp* with the *stable-gp* from the sequencing leader, which means the shard server can now serve reads up to the end of the previous batch. Then it persists the records using the mechanism provided by the underlying shard implementation. After that, it sends an acknowledgement back to the sequencing leader. Algorithm 3 shows the pseudo code for the shard server process.

Algorithm 3 Shard Server Process

```
Require: records: list of records received from sequencing leader, stable-gp
 1: shard-stable-gp ← stable-gp                        ▷ The shard's view of stable-gp
 2: persist(records)
 3: send acknowledgement to sequencing leader
```