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

Xuhao Luo, Shreesha G. Bhat*, Jiyu Hu*,
Ram Alagappan, Aishwarya Ganesan
University of Illinois Urbana-Champaign

*equal contribution

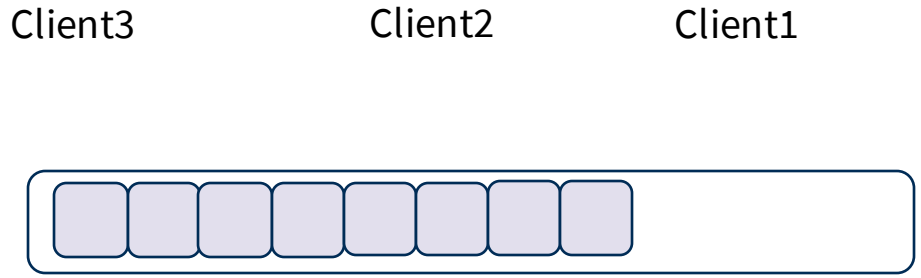
Shared Log: Abstraction and Interface

Shared Log: Abstraction and Interface

Shared Log is pervasive and used by many applications…

Shared Log: Abstraction and Interface

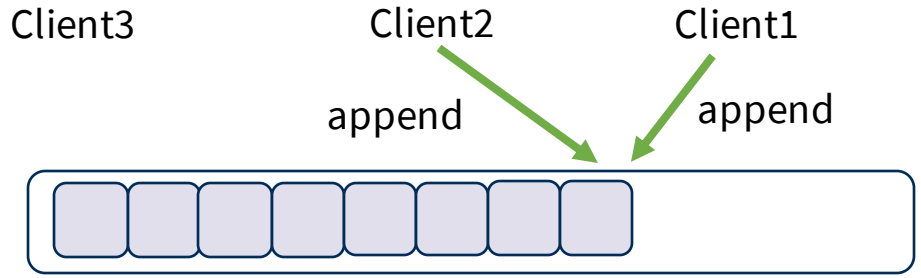
Shared Log is pervasive and used by many applications…



Shared Log: Abstraction and Interface

Shared Log is pervasive and used by many applications…

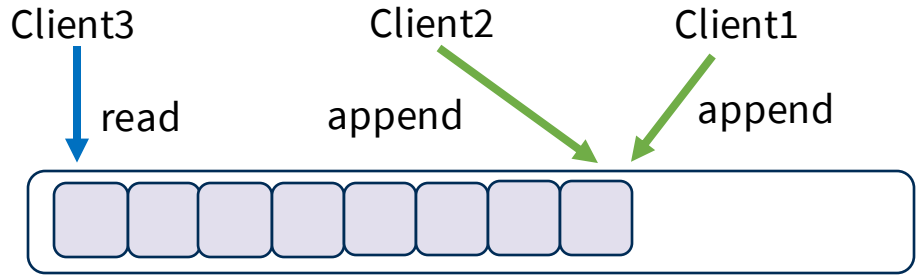
```
// append to log; return log position
uint64_t append(record r);
```



Shared Log: Abstraction and Interface

Shared Log is pervasive and used by many applications…

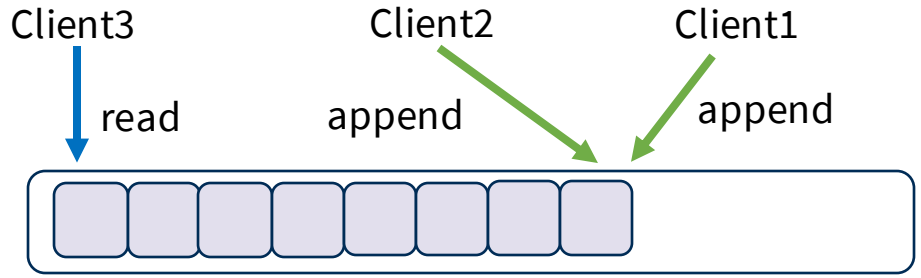
```
// append to log; return log position
uint64_t append(record r);
// read ‘len’ records starting at ‘from’
list read(logpos_t from, uint64_t len);
```



Shared Log: Abstraction and Interface

Shared Log is pervasive and used by many applications…

```
// append to log; return log position
uint64_t append(record r);
// read ‘len’ records starting at ‘from’
list read(logpos_t from, uint64_t len);
```

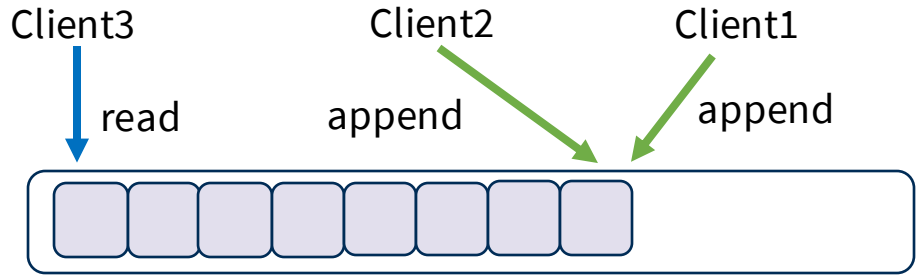


Fault-tolerant, linearizably ordered sequence of records

Shared Log: Abstraction and Interface

Shared Log is pervasive and used by many applications…

```
// append to log; return log position
uint64_t append(record r);
// read ‘len’ records starting at ‘from’
list read(logpos_t from, uint64_t len);
```



Fault-tolerant, linearizably ordered sequence of records

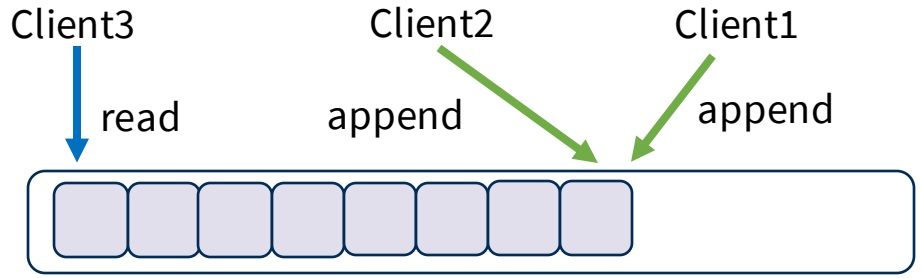
Implementations:

kafka

Shared Log: Abstraction and Interface

Shared Log is pervasive and used by many applications…

```
// append to log; return log position
uint64_t append(record r);
// read 'len' records starting at 'from'
list read(logpos_t from, uint64_t len);
```



Fault-tolerant, linearizably ordered sequence of records

Implementations:

Corfu [NSDI 12]

Scalog [NSDI 20]

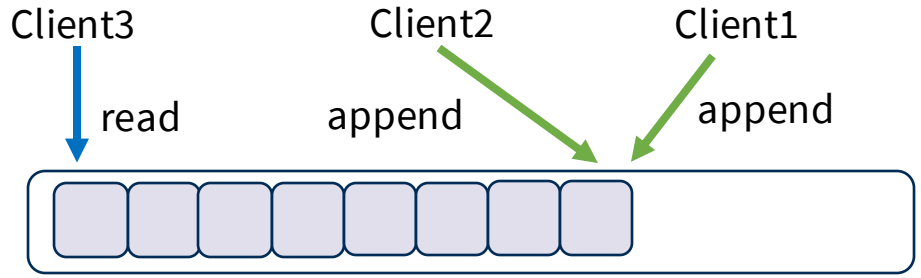
Boki [SOSP 21]

FlexLog [HPDC 23]

Shared Log: Abstraction and Interface

Shared Log is pervasive and used by many applications…

```
// append to log; return log position
uint64_t append(record r);
// read 'len' records starting at 'from'
list read(logpos_t from, uint64_t len);
```



Fault-tolerant, linearizably ordered sequence of records

Implementations:

kafka

amazon Kinesis

Cloud Pub/Sub

Azure Service Bus

Corfu [NSDI 12]

Scalog [NSDI 20]

Boki [SOSP 21]

FlexLog [HPDC 23]

The Problem with Current Shared Logs

The Problem with Current Shared Logs

- State-of-the-art implementations suffer from high ingestion latencies
  - Append takes multiple RTTs in Scalog, Corfu, etc.

The Problem with Current Shared Logs

- State-of-the-art implementations suffer from high ingestion latencies
  - Append takes multiple RTTs in Scalog, Corfu, etc.

- Low ingestion latency is critical to applications

The Problem with Current Shared Logs

- State-of-the-art implementations suffer from high ingestion latencies
  - Append takes multiple RTTs in Scalog, Corfu, etc.
- Low ingestion latency is critical to applications
- Rooted in eager ordering nature of shared logs:

The Problem with Current Shared Logs

- State-of-the-art implementations suffer from high ingestion latencies
  - Append takes multiple RTTs in Scalog, Corfu, etc.

- Low ingestion latency is critical to applications

- Rooted in eager ordering nature of shared logs:
  - Order is established eagerly upon appends

The Problem with Current Shared Logs

- State-of-the-art implementations suffer from high ingestion latencies
  - Append takes multiple RTTs in Scalog, Corfu, etc.

- Low ingestion latency is critical to applications

- Rooted in eager ordering nature of shared logs:
  - Order is established eagerly upon appends
  - Position of record is decided by the time append completes

Can a shared log avoid eager ordering, yet also preserve the ordering guarantees of conventional shared logs?

LazyLog: Idea and Abstraction

Insight

Idea

LazyLog: Idea and Abstraction

Insight

① In many applications, linearizable ordering is not required right away upon ingestion.

Idea

LazyLog: Idea and Abstraction

Insight

① In many applications, linearizable ordering is not required right away upon ingestion.

Idea

- Shared log need not eagerly bind a record to a position upon an append
  - But only make it durable

LazyLog: Idea and Abstraction

Insight

① In many applications, linearizable ordering is not required right away upon ingestion.

② Linearizable order is needed when records are consumed

Idea

- Shared log need not eagerly bind a record to a position upon an append
  - But only make it durable

LazyLog: Idea and Abstraction

Insight

① In many applications, linearizable ordering is not required right away upon ingestion.

② Linearizable order is needed when records are consumed

Idea

- Shared log need not eagerly bind a record to a position upon an append
  - But only make it durable
- Although shared log can bind records to positions lazily, it must enforce ordering before positions can be read

LazyLog: Idea and Abstraction

Insight

① In many applications, linearizable ordering is not required right away upon ingestion.

② Linearizable order is needed when records are consumed

③ In many apps, readers are naturally decoupled temporally from writers

Idea

- Shared log need not eagerly bind a record to a position upon an append
  - But only make it durable
- Although shared log can bind records to positions lazily, it must enforce ordering before positions can be read

LazyLog: Idea and Abstraction

Insight

① In many applications, linearizable ordering is not required right away upon ingestion.

② Linearizable order is needed when records are consumed

③ In many apps, readers are naturally decoupled temporally from writers

Idea

- Shared log need not eagerly bind a record to a position upon an append
  - But only make it durable
- Although shared log can bind records to positions lazily, it must enforce ordering before positions can be read
- Shared log can do the ordering comfortably in the background

LazyLog: Idea and Abstraction

Insight

① In many applications, linearizable ordering is not required right away upon ingestion.

② Linearizable order is needed when records are consumed

③ In many apps, readers are naturally decoupled temporally from writers

Idea

- Shared log need not eagerly bind a record to a position upon an append
  - But only make it durable
- Although shared log can bind records to positions lazily, it must enforce ordering before positions can be read
- Shared log can do the ordering comfortably in the background

LazyLog: A new shared log abstraction built upon these ideas

Results Overview

Results Overview

- Implemented *LazyLog* abstraction

Results Overview

- Implemented *LazyLog* abstraction

- Offers 1-RTT appends, greatly reduce ingestion latency while providing linearizability
    - As opposed to multiple RTTs in Scalog and Corfu

Results Overview

- Implemented *LazyLog* abstraction

- Offers 1-RTT appends, greatly reduce ingestion latency while providing linearizability
    - As opposed to multiple RTTs in Scalog and Corfu
    - Nearly no overhead on reads

Results Overview

- Implemented *LazyLog* abstraction
- Offers 1-RTT appends, greatly reduce ingestion latency while providing linearizability
  - As opposed to multiple RTTs in Scalog and Corfu
  - Nearly no overhead on reads
- Benefits end apps like KV store, audit-logging, checkpointing with LazyLog's low latency

Outline

Introduction

Motivation

LazyLog Insight and Interface

LazyLog System Design

Performance Evaluation

Total Ordering in Shared Logs

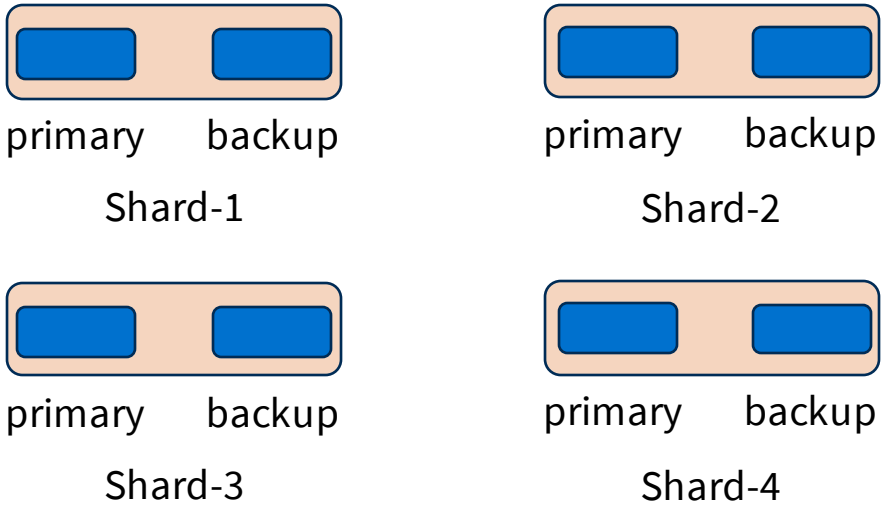
Total Ordering in Shared Logs

- Linearizable ordering: if *append(B)* starts after *append(A)* completes, then *B* appears after *A* in the shared log

Total Ordering in Shared Logs

- Linearizable ordering: if *append(B)* starts after *append(A)* completes, then *B* appears after *A* in the shared log

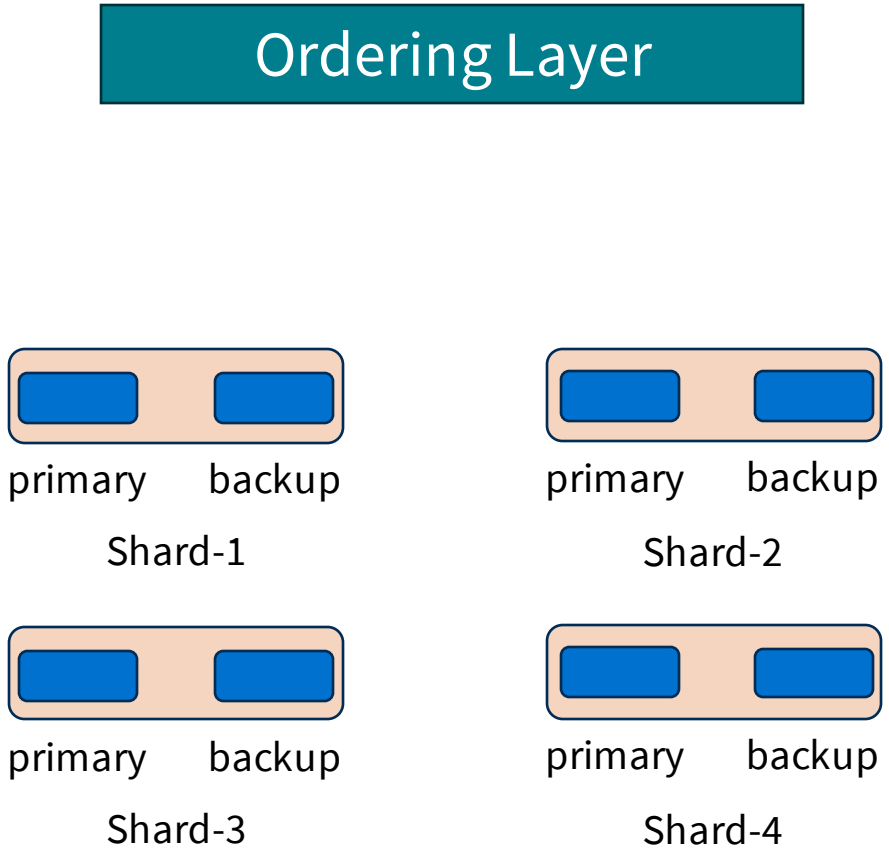
- **Shards**



Total Ordering in Shared Logs

- Linearizable ordering: if *append(B)* starts after *append(A)* completes, then *B* appears after *A* in the shared log

- **Shards**
- **Ordering Layer**

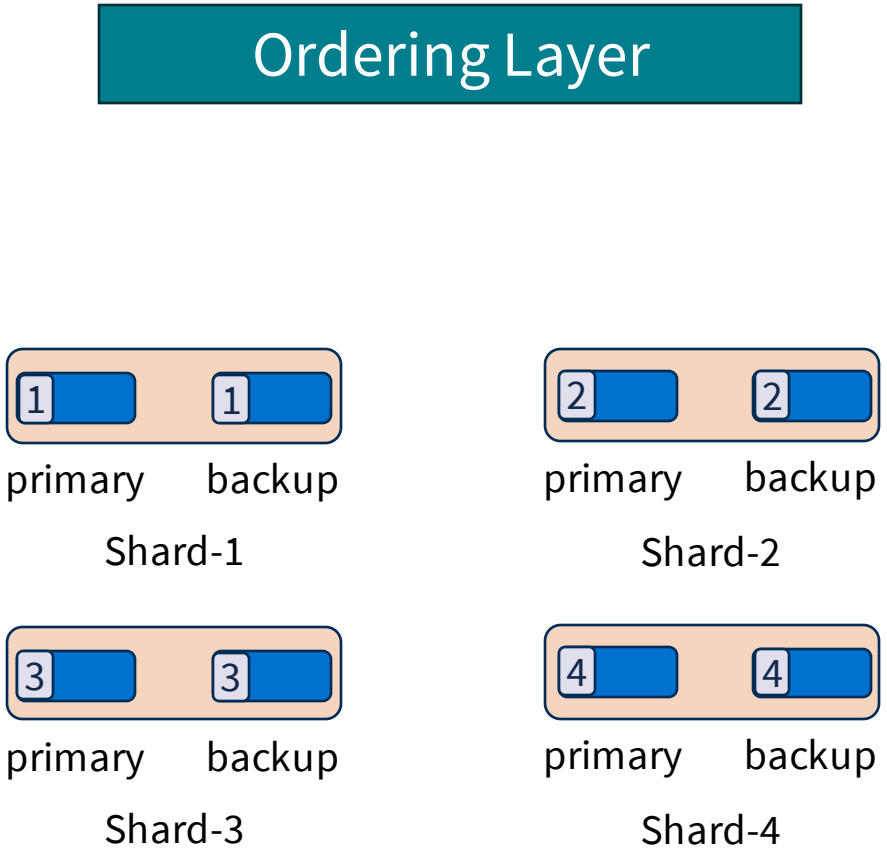


Total Ordering in Shared Logs

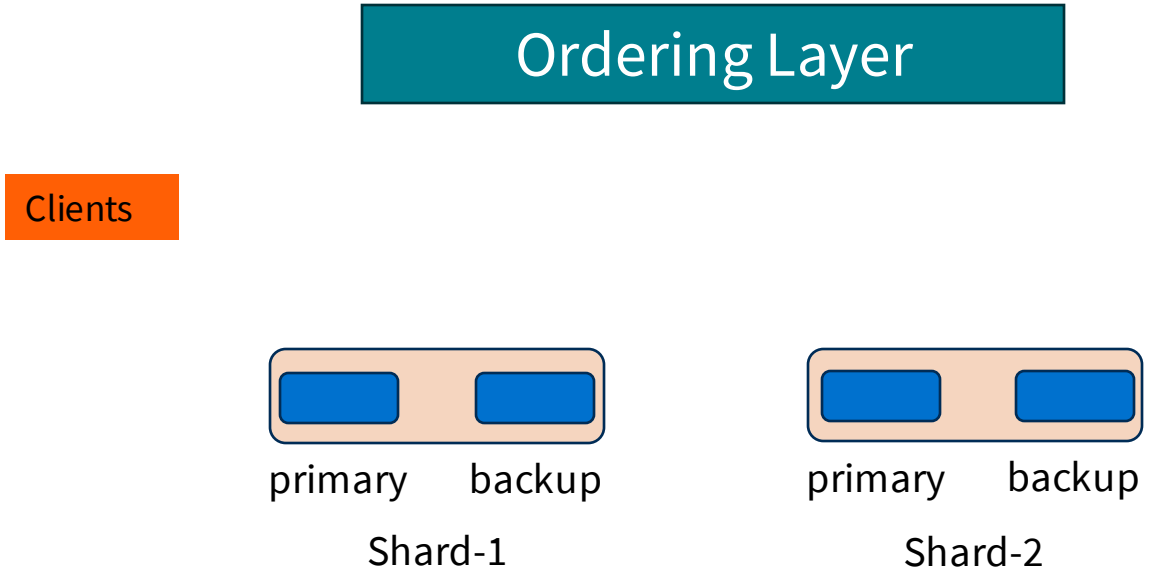
- Linearizable ordering: if *append(B)* starts after *append(A)* completes, then *B* appears after *A* in the shared log

- **Shards**
- **Ordering Layer**

Shared log provides total order across shards

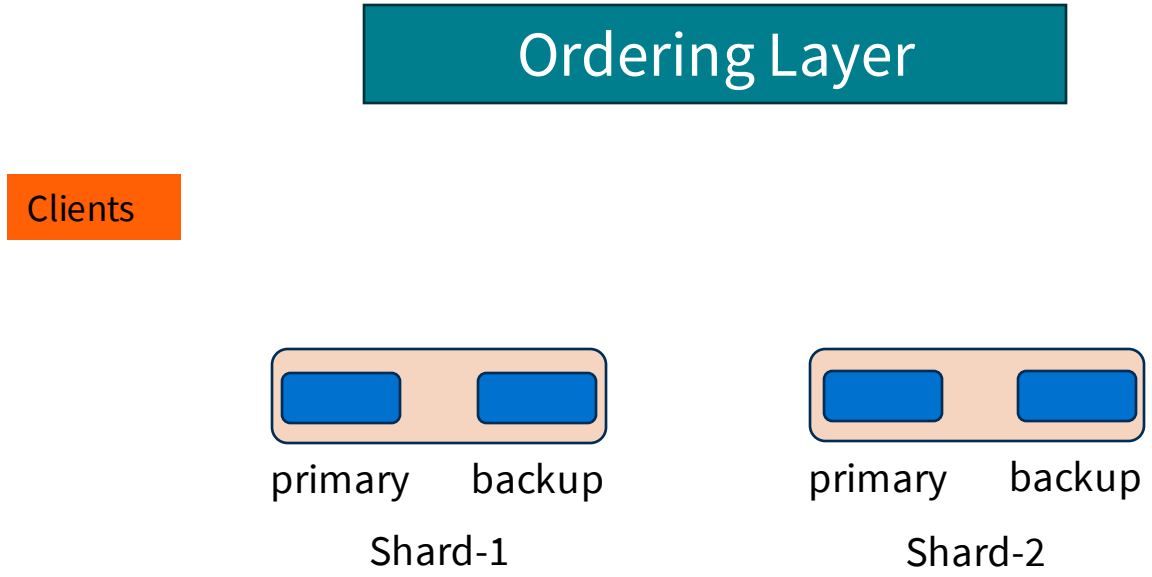


Eager Ordering → High Latency



Eager Ordering → High Latency

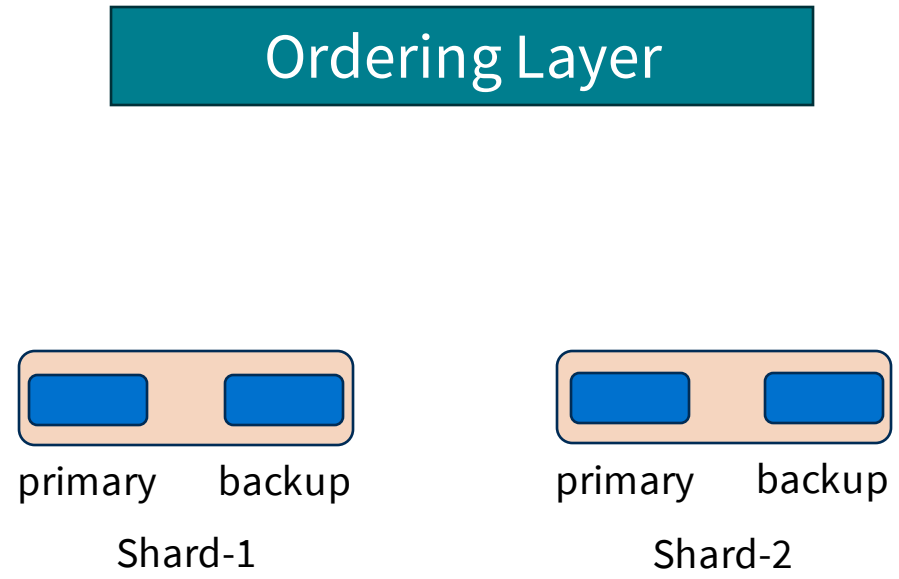
Shared logs today incur high ingestion latency



Eager Ordering → High Latency

Shared logs today incur high ingestion latency

Rooted in eager ordering

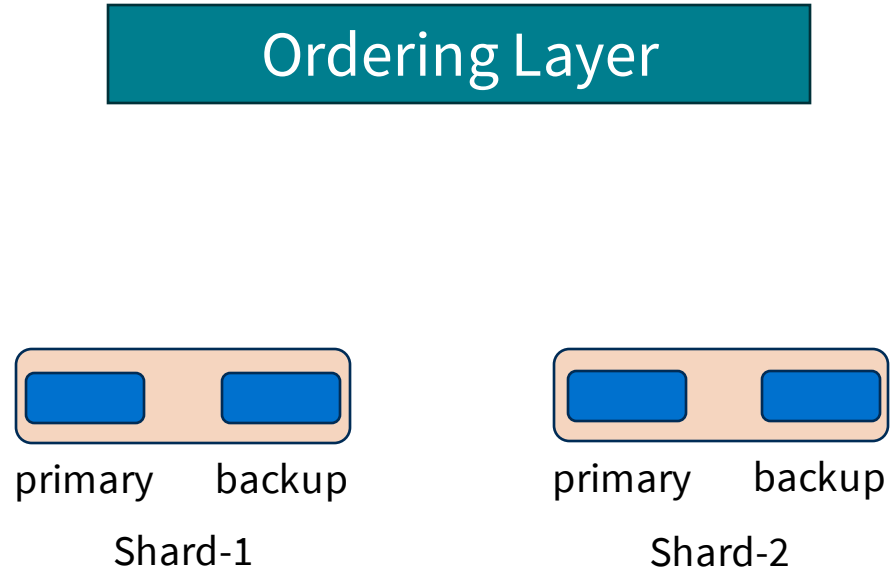


Eager Ordering → High Latency

Shared logs today incur high ingestion latency

Rooted in eager ordering

Both durability and global ordering are completed before getting back to clients



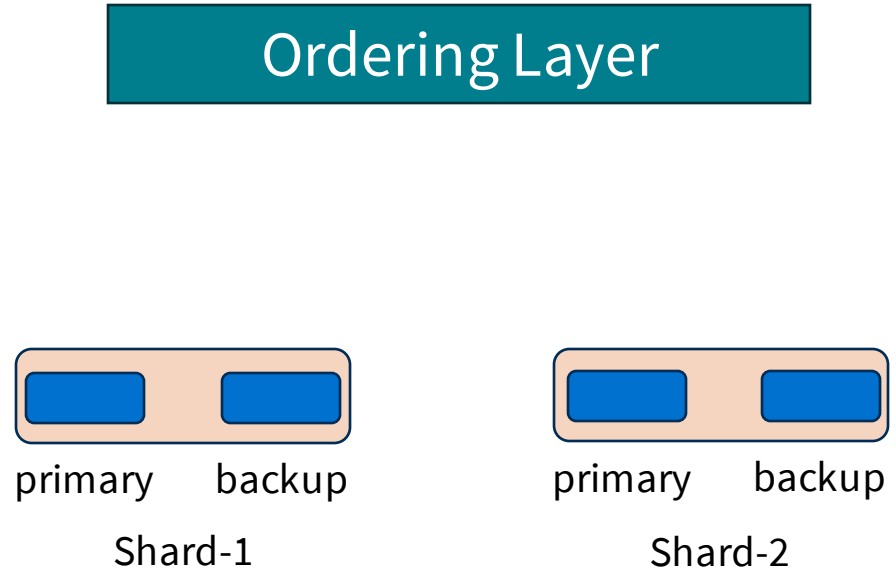
Eager Ordering → High Latency

Shared logs today incur high ingestion latency

Rooted in eager ordering

Both durability and global ordering are completed before getting back to clients

- Scalog



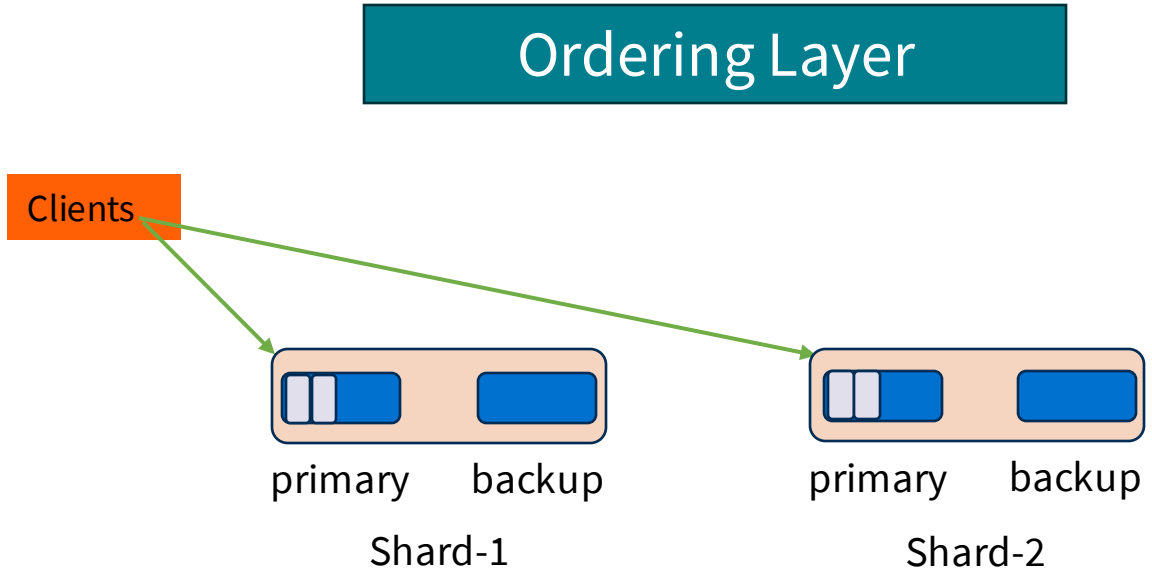
Eager Ordering → High Latency

Shared logs today incur high ingestion latency

Rooted in eager ordering

Both durability and global ordering are completed before getting back to clients

- Scalog
  - durability first



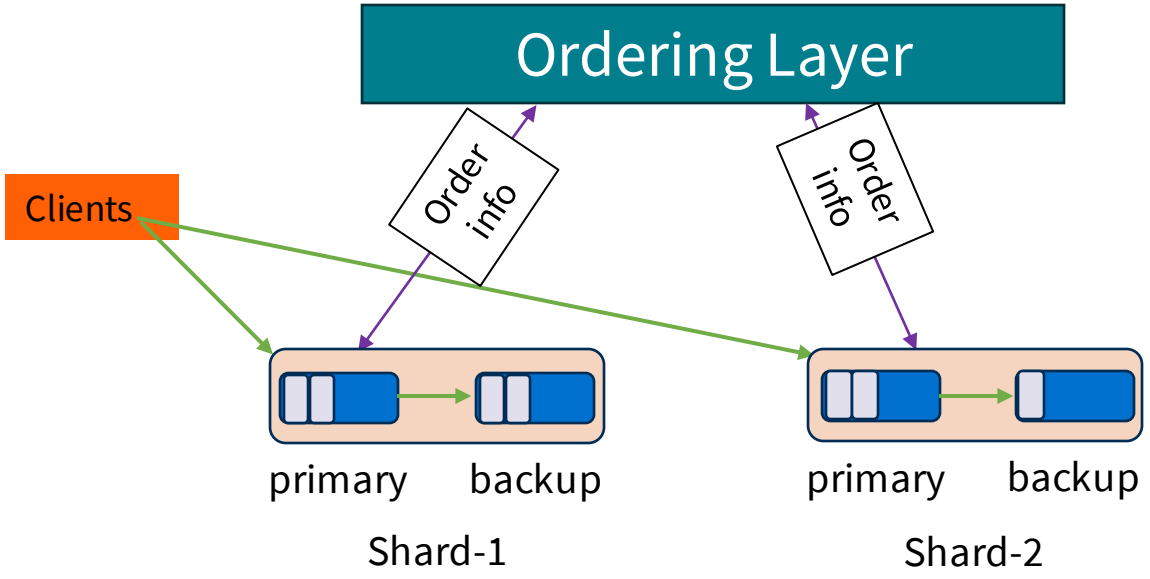
Eager Ordering → High Latency

Shared logs today incur high ingestion latency

Rooted in eager ordering

Both durability and global ordering are completed before getting back to clients

- Scalog
  - durability first
  - then global ordering



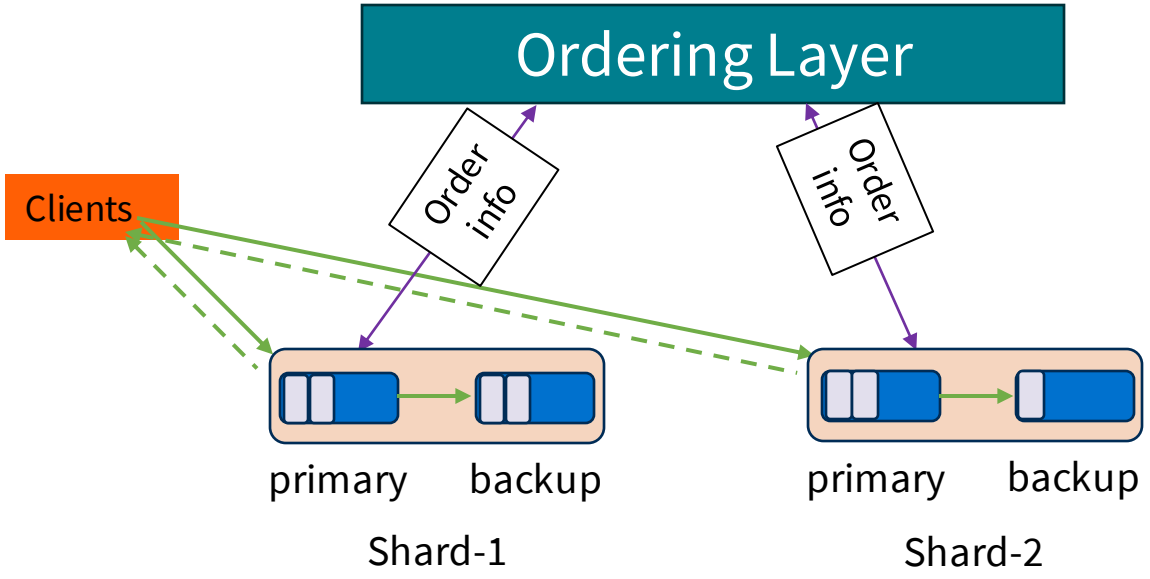
Eager Ordering → High Latency

Shared logs today incur high ingestion latency

Rooted in eager ordering

Both durability and global ordering are completed before getting back to clients

- Scalog
  - durability first
  - then global ordering



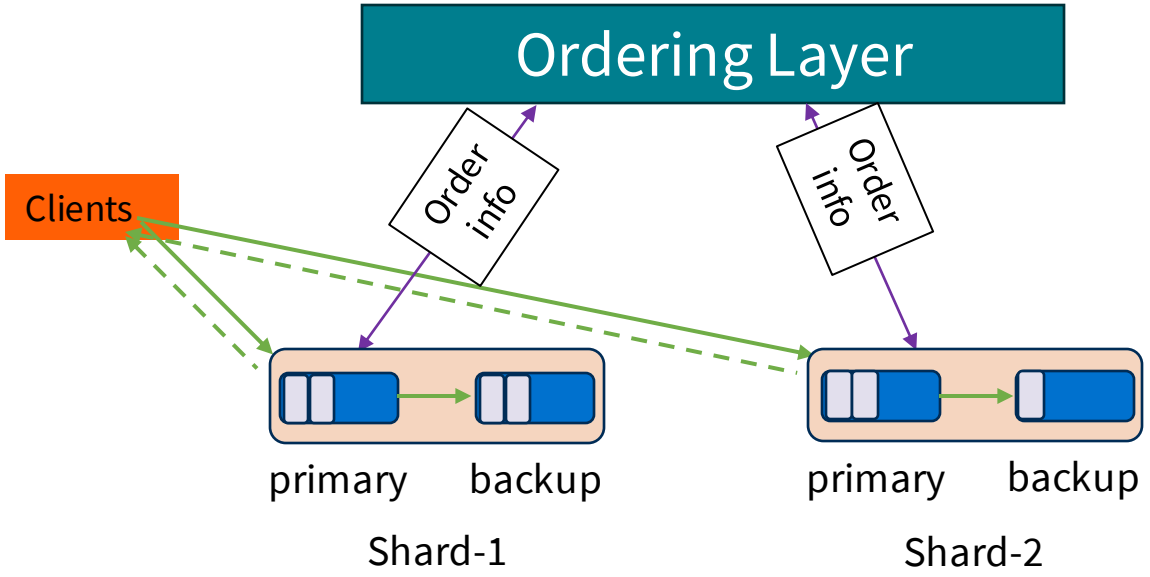
Eager Ordering → High Latency

Shared logs today incur high ingestion latency

Rooted in eager ordering

Both durability and global ordering are completed before getting back to clients

- Scalog
  - durability first
  - then global ordering
  - 3.5RTT + batch interval



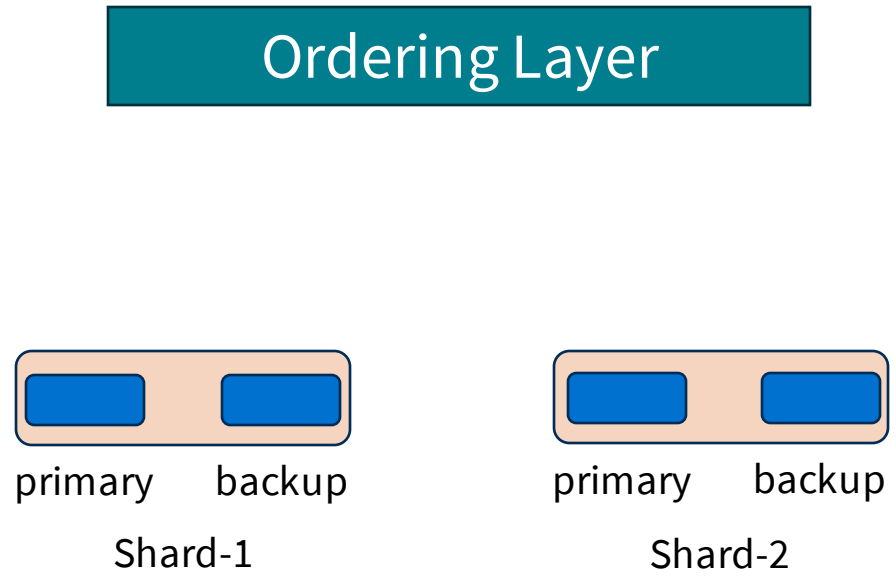
Eager Ordering → High Latency

Shared logs today incur high ingestion latency

Rooted in eager ordering

Both durability and global ordering are completed before getting back to clients

- Scalog
  - durability first
  - then global ordering
  - 3.5RTT + batch interval
- Corfu



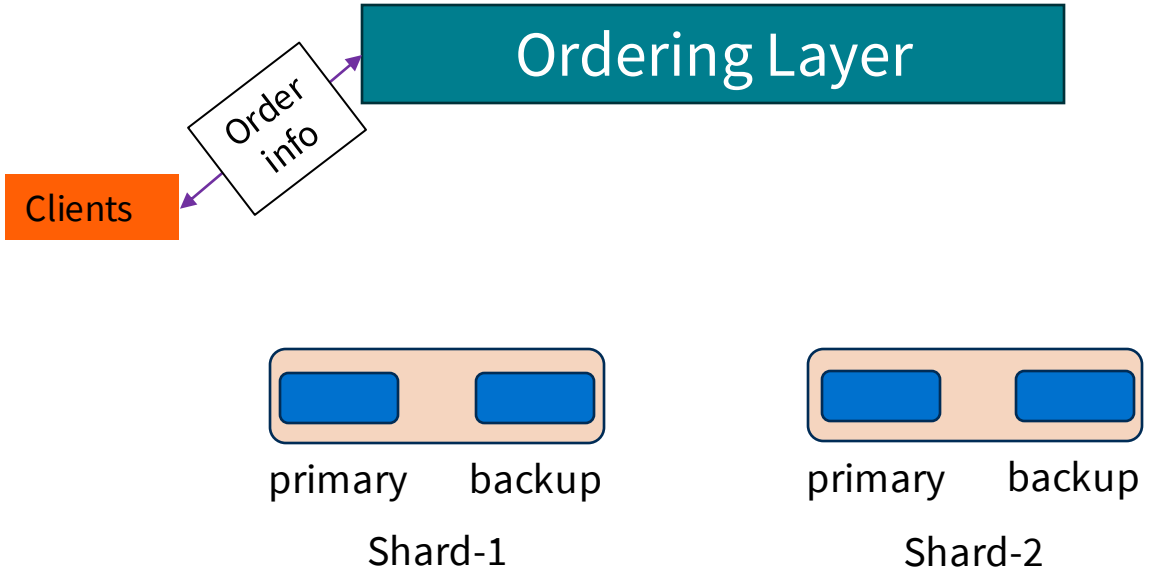
Eager Ordering → High Latency

Shared logs today incur high ingestion latency

Rooted in eager ordering

Both durability and global ordering are completed before getting back to clients

- Scalog
  - durability first
  - then global ordering
  - 3.5RTT + batch interval
- Corfu
  - global ordering first



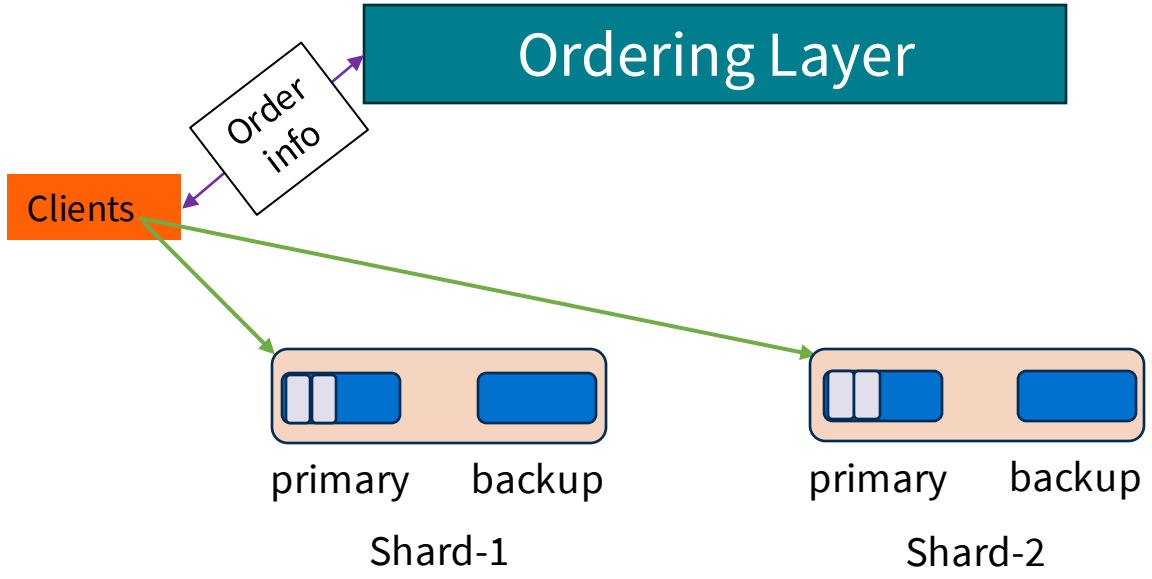
Eager Ordering → High Latency

Shared logs today incur high ingestion latency

Rooted in eager ordering

Both durability and global ordering are completed before getting back to clients

- Scalog
  - durability first
  - then global ordering
  - 3.5RTT + batch interval
- Corfu
  - global ordering first
  - then durability



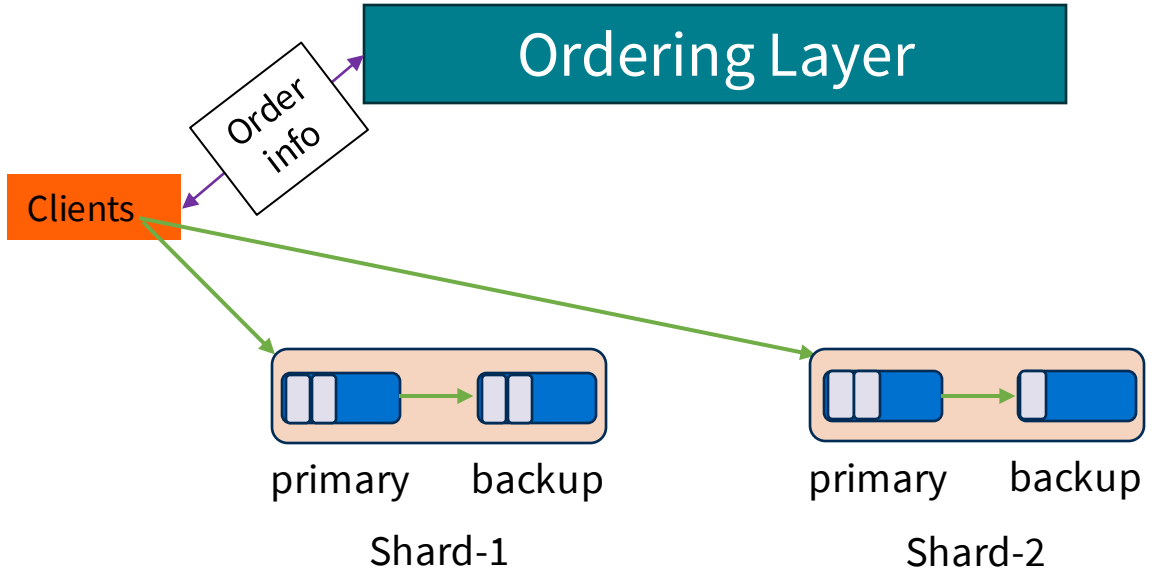
Eager Ordering → High Latency

Shared logs today incur high ingestion latency

Rooted in eager ordering

Both durability and global ordering are completed before getting back to clients

- Scalog
  - durability first
  - then global ordering
  - 3.5RTT + batch interval
- Corfu
  - global ordering first
  - then durability



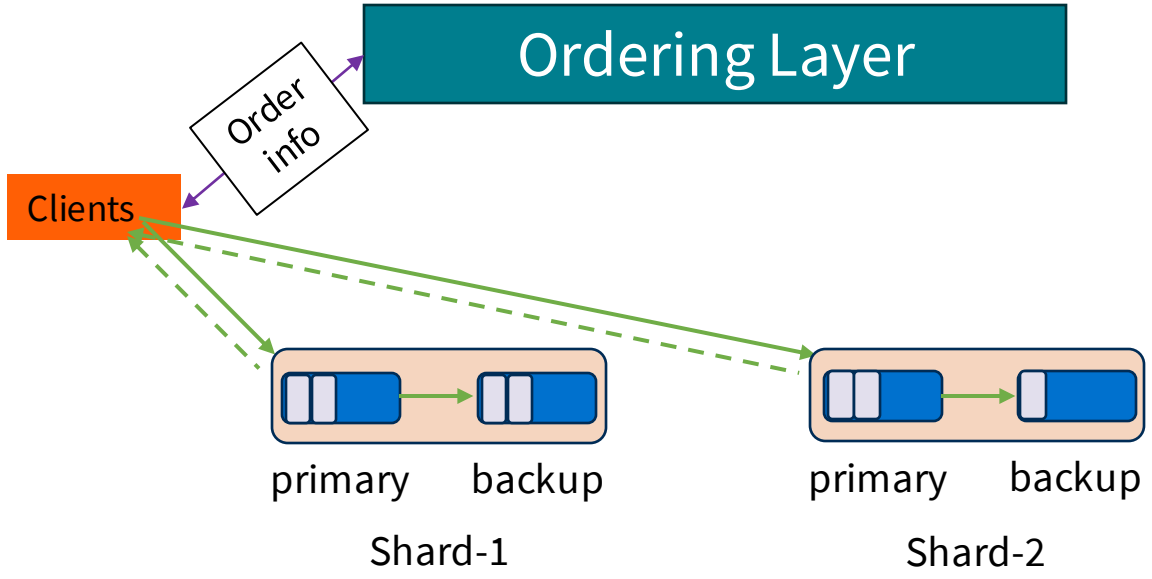
Eager Ordering → High Latency

Shared logs today incur high ingestion latency

Rooted in eager ordering

Both durability and global ordering are completed before getting back to clients

- Scalog
  - durability first
  - then global ordering
  - 3.5RTT + batch interval
- Corfu
  - global ordering first
  - then durability



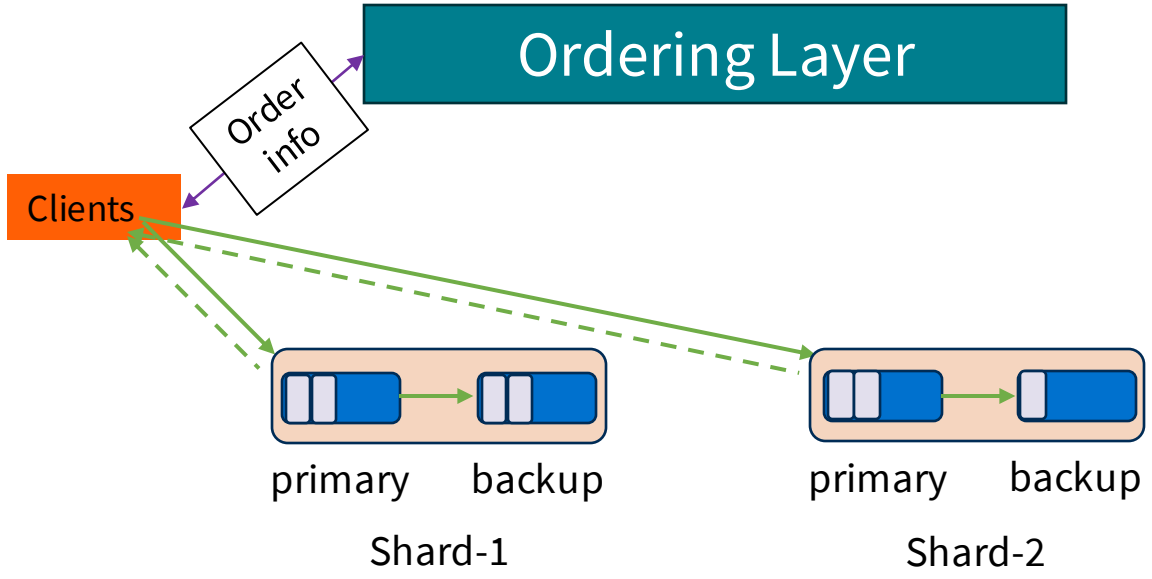
Eager Ordering → High Latency

Shared logs today incur high ingestion latency

Rooted in eager ordering

Both durability and global ordering are completed before getting back to clients

- Scalog
  - durability first
  - then global ordering
  - 3.5RTT + batch interval
- Corfu
  - global ordering first
  - then durability
  - 3RTT



Eager Ordering → High Latency

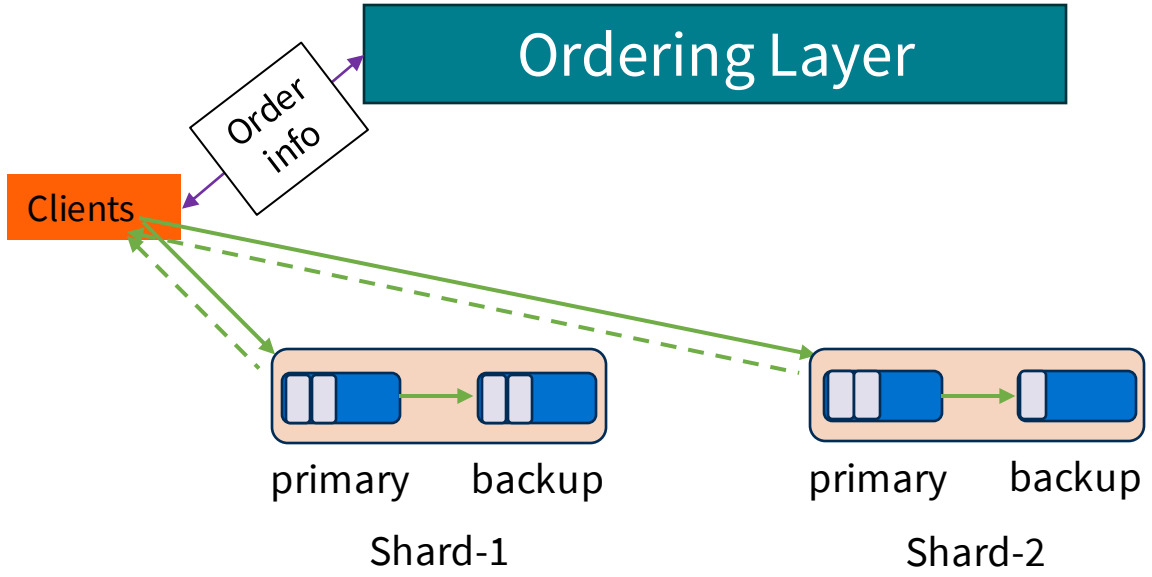
Shared logs today incur high ingestion latency

Rooted in eager ordering

Both durability and global ordering are completed before getting back to clients

- Scalog
  - durability first
  - then global ordering
  - 3.5RTT + batch interval
- Corfu
  - global ordering first
  - then durability
  - 3RTT

Results in high ingestion latency for applications



Low-Latency Ingestion is Critical for Apps

Low-Latency Ingestion is Critical for Apps

Distributed databases

e.g., FireScroll built atop RedPanda requires quick durability

Low-Latency Ingestion is Critical for Apps

Distributed databases

- e.g., FireScroll built atop RedPanda requires quick durability

Similarly, event sourcing, journaling for FT, and log aggregation require low-latency logging

Low-Latency Ingestion is Critical for Apps

Distributed databases

e.g., FireScroll built atop RedPanda requires quick durability

Similarly, event sourcing, journaling for FT, and log aggregation require low-latency logging

A 2023 survey by RedPanda:

1/3 of 300 practitioners rated

ingestion latency as the primary

latency metric they care about

Outline

Introduction

Motivation

LazyLog Insight and Interface

LazyLog System Design

Performance Evaluation

Insight

Insight

Although linearizable order is necessary, in many applications,
it is not needed eagerly upon ingestion
But only later upon reads

Insight

Although linearizable order is necessary, in many applications,

it is not needed eagerly upon ingestion

But only later upon reads

And readers are naturally decoupled temporally from writers

Insight

Although linearizable order is necessary, in many applications,
it is not needed eagerly upon ingestion
But only later upon reads
And readers are naturally decoupled temporally from writers

A shared log can thus defer ordering upon appends
But establish it before reads arrive

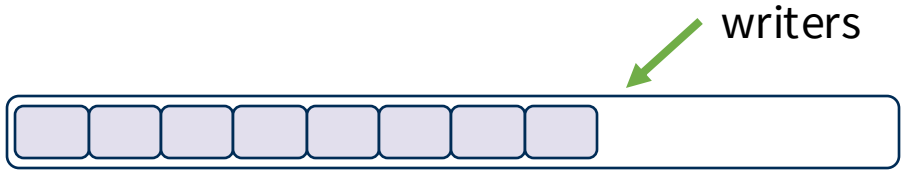
Holds for Many Apps

Holds for Many Apps

Example: reader-writer decoupled databases like FireScroll

Holds for Many Apps

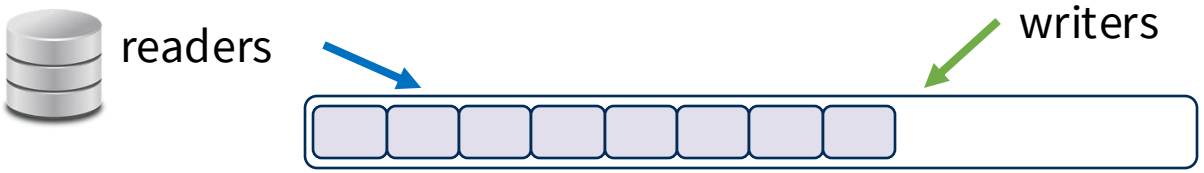
Example: reader-writer decoupled databases like FireScroll



① Writers do not require or use the appended index immediately

Holds for Many Apps

Example: reader-writer decoupled databases like FireScroll

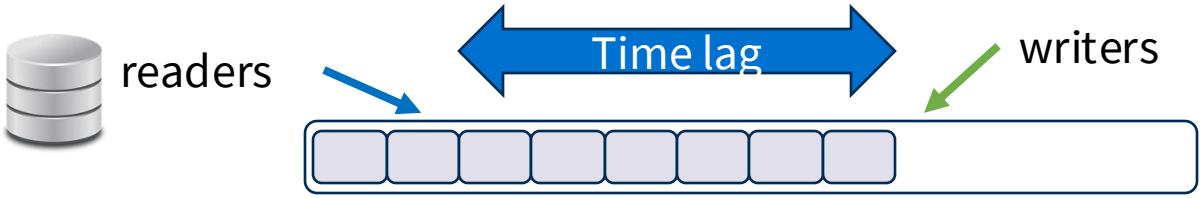


① Writers do not require or use the appended index immediately

② Readers must apply updates in linearizable order to construct the correct state

Holds for Many Apps

Example: reader-writer decoupled databases like FireScroll



① Writers do not require or use the appended index immediately

② Readers must apply updates in linearizable order to construct the correct state

③ Readers and writers are time-decoupled: readers typically lag behind writers

Application Study

Application Study

- Event Sourcing
- Readers lag behind writers to avoid interference

Application Study

- Event Sourcing
- Activity logging
- Readers lag behind writers to avoid interference
- Analytic jobs lag behind writers

Application Study

- Event Sourcing
- Activity logging
- Log aggregation

- Readers lag behind writers to avoid interference
- Analytic jobs lag behind writers
- Logs are only read much later during debugging

Application Study

- Event Sourcing
- Activity logging
- Log aggregation
- High-availability journaling

- Readers lag behind writers to avoid interference
- Analytic jobs lag behind writers
- Logs are only read much later during debugging
- Journal is accessed only upon failures

Application Study

- Event Sourcing
- Activity logging
- Log aggregation
- High-availability journaling

- Readers lag behind writers to avoid interference
- Analytic jobs lag behind writers
- Logs are only read much later during debugging
- Journal is accessed only upon failures

LazyLog's insights also hold for them

LazyLog: Abstraction and Interface

Abstraction

Interface

LazyLog: Abstraction and Interface

Abstraction

- LazyLog doesn’t bind a record to a position upon append

Interface

LazyLog: Abstraction and Interface

Abstraction

- LazyLog doesn’t bind a record to a position upon append
  - Only makes the record durable

Interface

LazyLog: Abstraction and Interface

Abstraction

- LazyLog doesn’t bind a record to a position upon append
  - Only makes the record durable
  - Guarantee the record will be eventually bound to correct position

Interface

LazyLog: Abstraction and Interface

Abstraction

- LazyLog doesn't bind a record to a position upon append
  - Only makes the record durable
  - Guarantee the record will be eventually bound to correct position
- LazyLog lazily binds records to positions and enforces ordering before the positions can be read

Interface

LazyLog: Abstraction and Interface

Abstraction

- LazyLog doesn't bind a record to a position upon append
  - Only makes the record durable
  - Guarantee the record will be eventually bound to correct position
- LazyLog lazily binds records to positions and enforces ordering before the positions can be read

Interface

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

LazyLog: Abstraction and Interface

Abstraction

- LazyLog doesn't bind a record to a position upon append
  - Only makes the record durable
  - Guarantee the record will be eventually bound to correct position
- LazyLog lazily binds records to positions and enforces ordering before the positions can be read

Interface

Index is not returned!

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

LazyLog: Abstraction and Interface

Abstraction

- LazyLog doesn't bind a record to a position upon append
  - Only makes the record durable
  - Guarantee the record will be eventually bound to correct position
- LazyLog lazily binds records to positions and enforces ordering before the positions can be read

Interface

Index is not returned!

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

// read 'len' records starting at 'from'
list read(logpos_t from, uint64_t len);
```

Performance Property

Performance Property

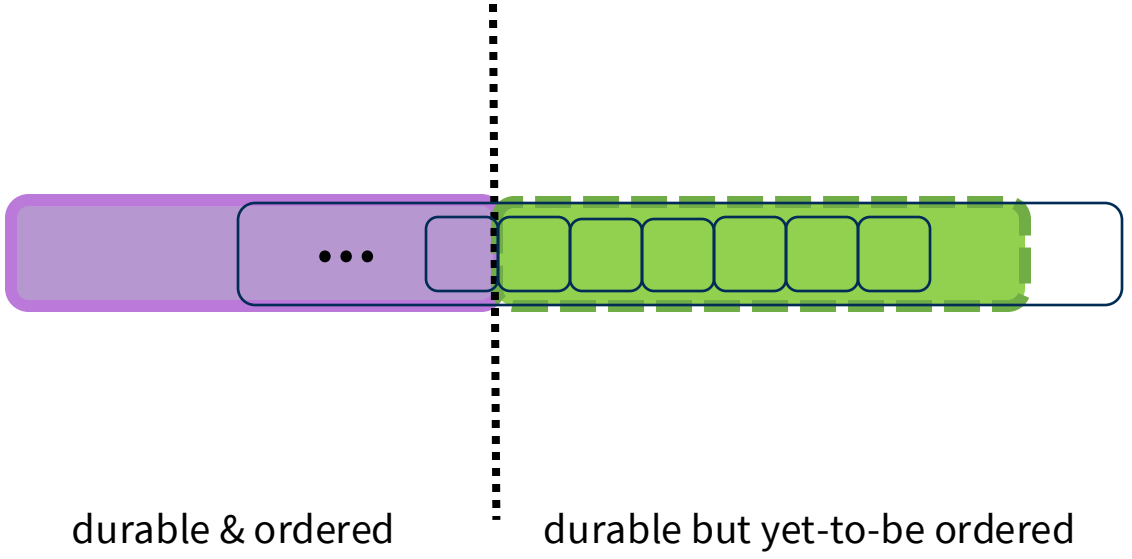
- Cannot be too lazy – keep ordering in the background

Performance Property

- Cannot be too lazy – keep ordering in the background

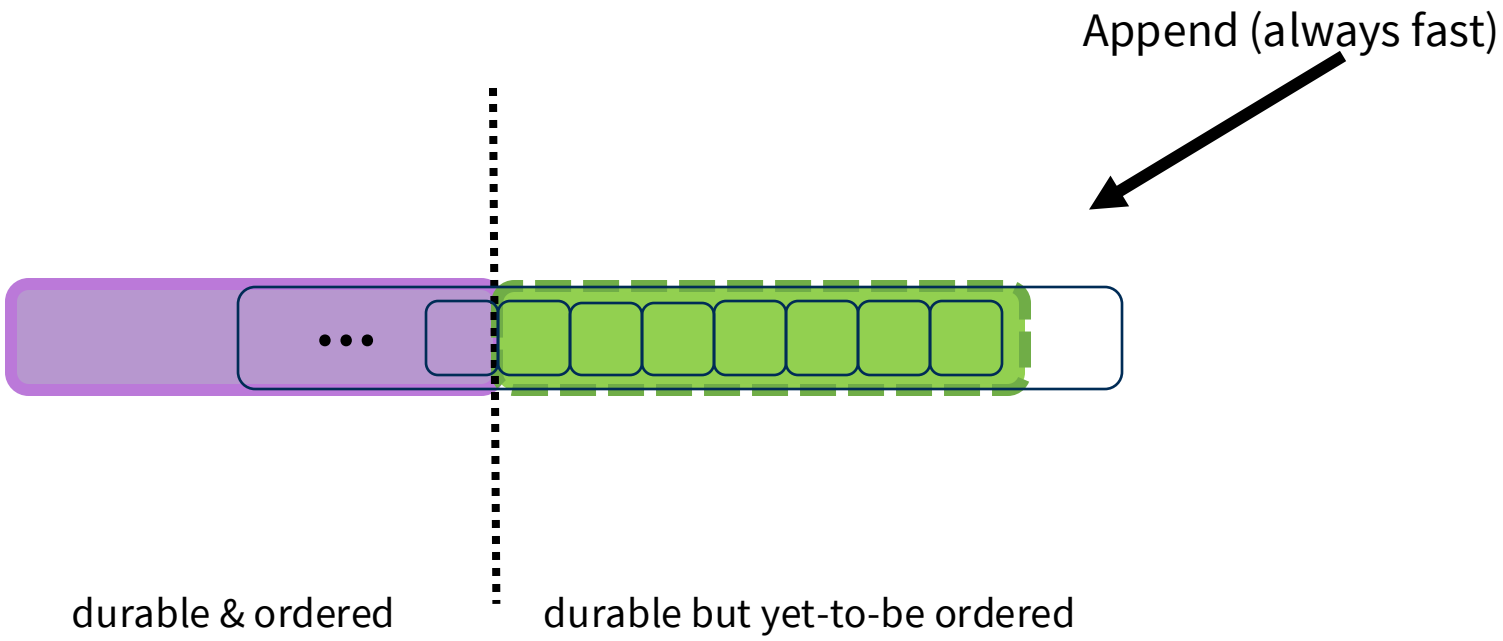
Performance Property

- Cannot be too lazy – keep ordering in the background



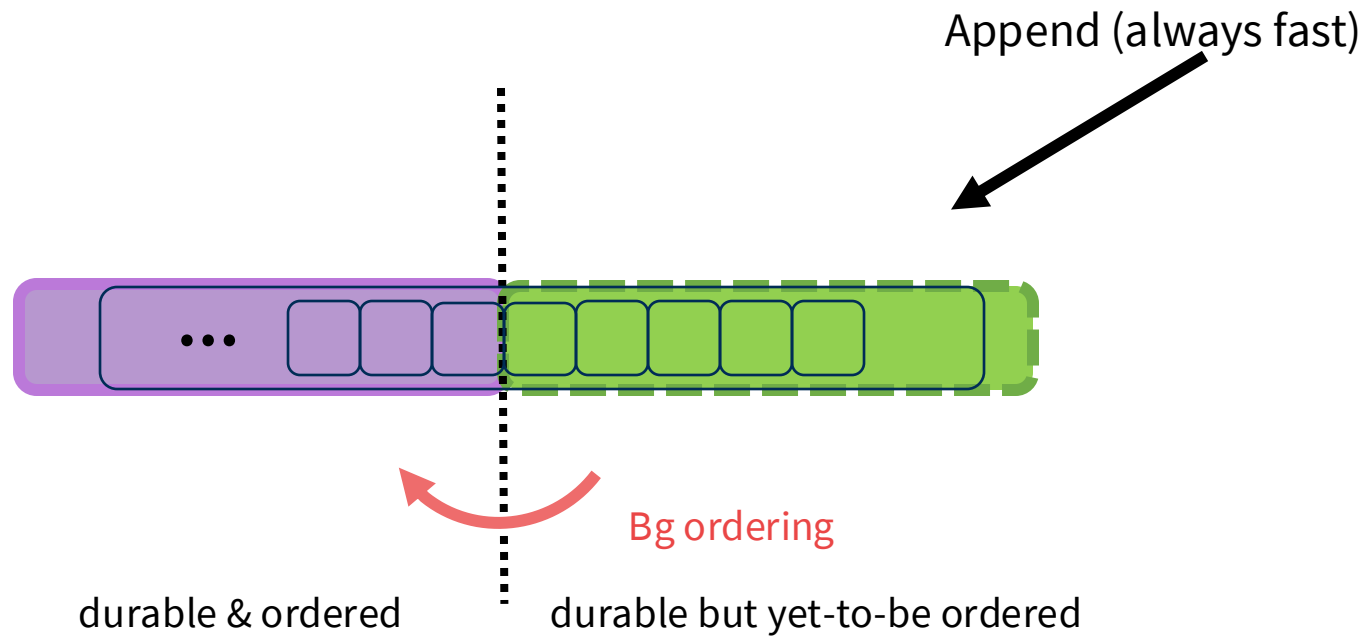
Performance Property

- Cannot be too lazy – keep ordering in the background



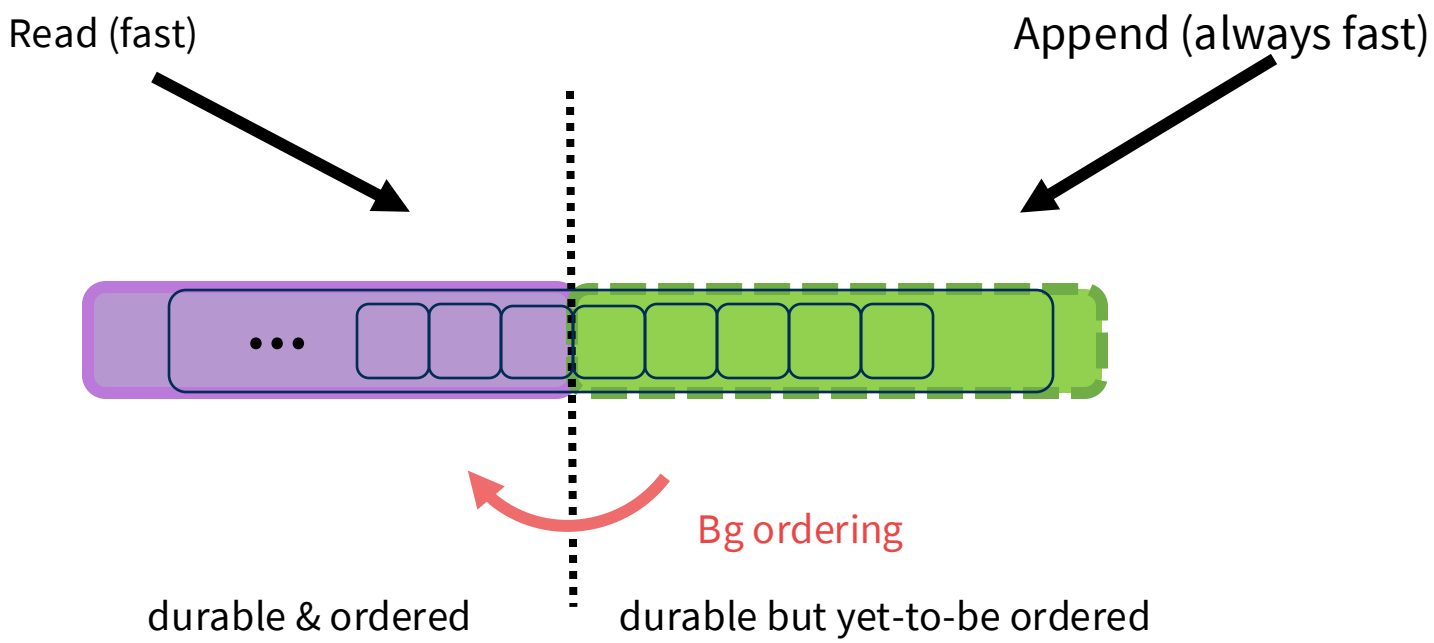
Performance Property

- Cannot be too lazy – keep ordering in the background



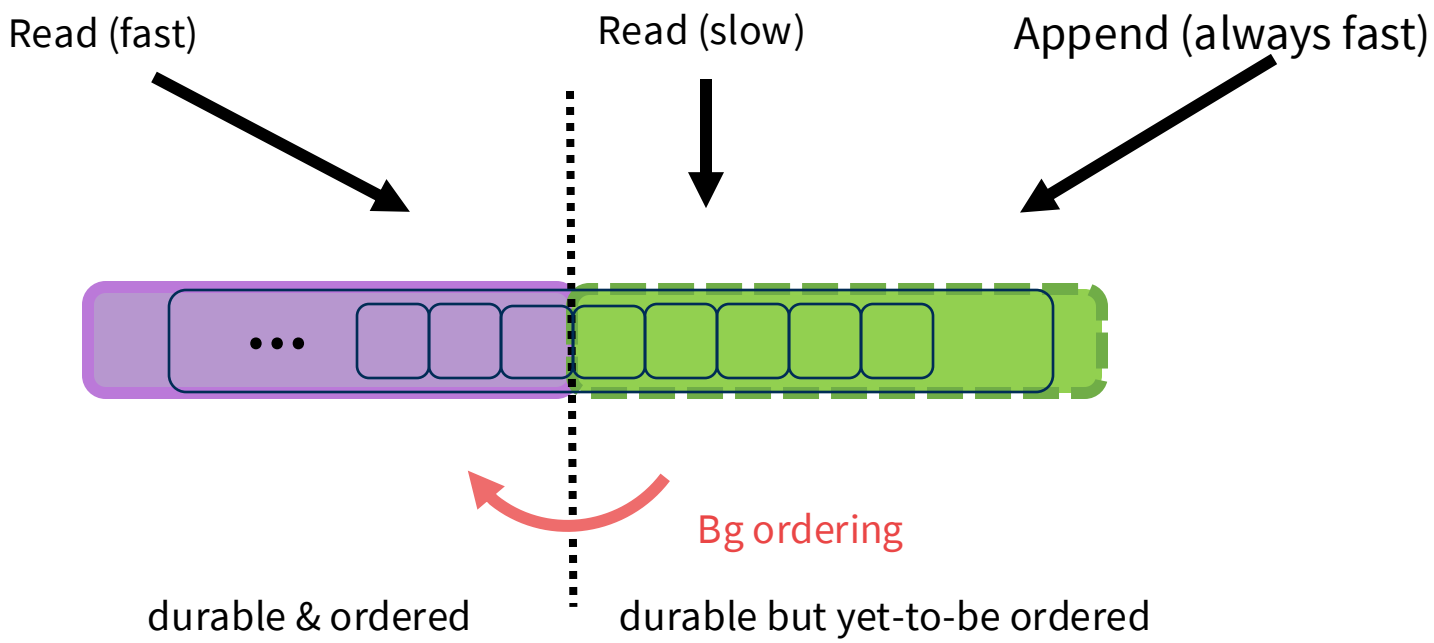
Performance Property

- Cannot be too lazy – keep ordering in the background



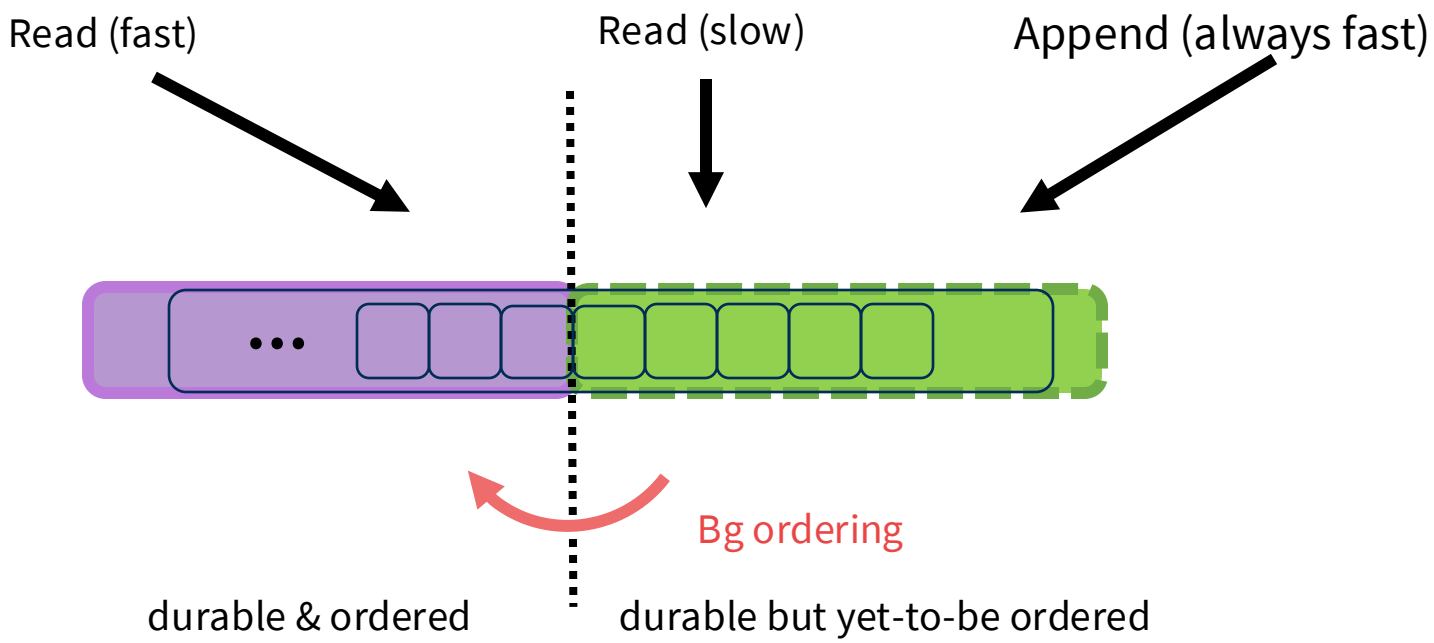
Performance Property

- Cannot be too lazy – keep ordering in the background



Performance Property

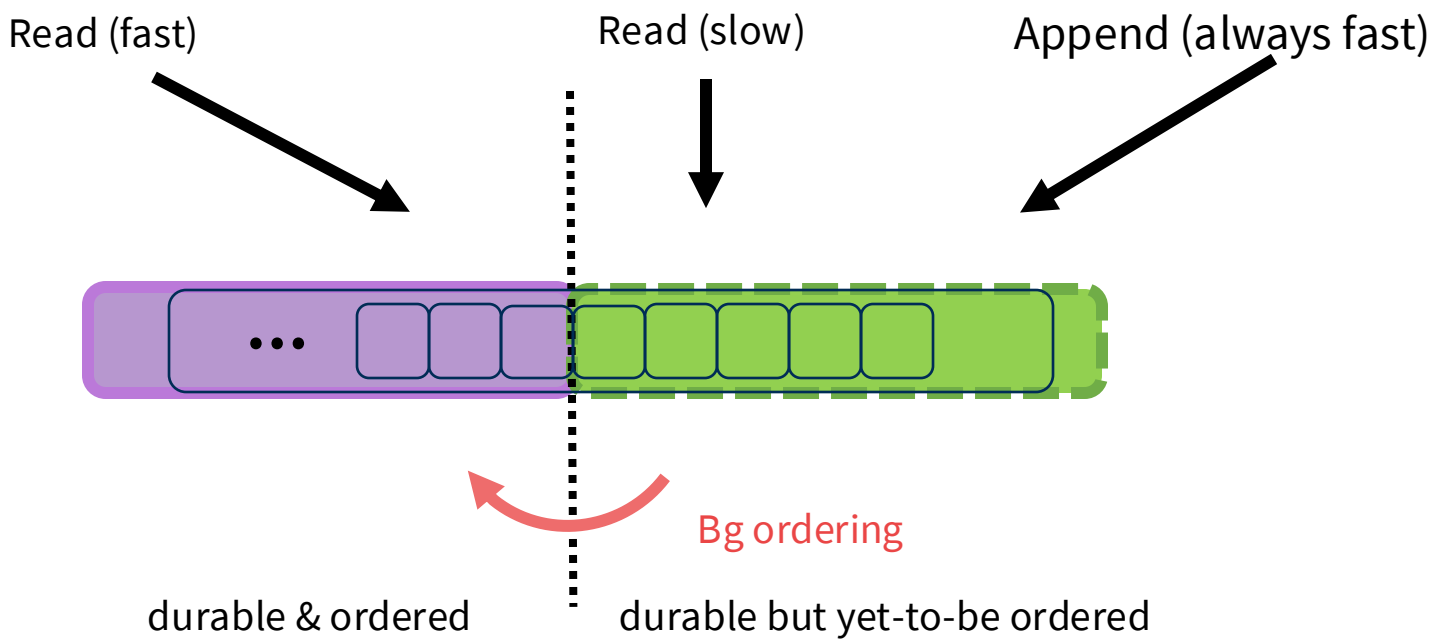
- Cannot be too lazy – keep ordering in the background



- For many apps – reads are always fast

Performance Property

- Cannot be too lazy – keep ordering in the background



- For many apps – reads are always fast
- Even if immediately read, LazyLog preserves the performance of eager shared logs
  - never worse than an eager-ordering shared log!

Related Work

Related Work

LazyLog is inspired by the general idea of deferring work until needed

Related Work

LazyLog is inspired by the general idea of deferring work until needed

Exploited in other contexts like filesystems (Speculator[SOSP'05]) and databases (Lazy Eval Txn[SIGMOD'14])

Related Work

LazyLog is inspired by the general idea of deferring work until needed

Exploited in other contexts like filesystems (Speculator[SOSP'05]) and databases (Lazy Eval Txn[SIGMOD'14])

Also in distributed systems

Related Work

LazyLog is inspired by the general idea of deferring work until needed

Exploited in other contexts like filesystems (Speculator[SOSP'05]) and databases (Lazy Eval Txn[SIGMOD'14])

Also in distributed systems

- Skyros[SOSP'21]: defer ordering within a single shard

Related Work

LazyLog is inspired by the general idea of deferring work until needed

Exploited in other contexts like filesystems (Speculator[SOSP'05]) and databases (Lazy Eval Txn[SIGMOD'14])

Also in distributed systems

- Skyros[SOSP'21]: defer ordering within a single shard
- Occult[NSDI'17]: defer ordering across shards, but only provides causal ordering

Related Work

LazyLog is inspired by the general idea of deferring work until needed

Exploited in other contexts like filesystems (Speculator[SOSP'05]) and databases (Lazy Eval Txn[SIGMOD'14])

Also in distributed systems

- Skyros[SOSP'21]: defer ordering within a single shard
- Occult[NSDI'17]: defer ordering across shards, but only provides causal ordering

- LazyLog: First shared log to offer linearizable ordering across shards with low latency by deferring ordering

Related Work

LazyLog is inspired by the general idea of deferring work until needed

Exploited in other contexts like filesystems (Speculator[SOSP'05]) and databases (Lazy Eval Txn[SIGMOD'14])

Also in distributed systems

- Skyros[SOSP'21]: defer ordering within a single shard
- Occult[NSDI'17]: defer ordering across shards, but only provides causal ordering

- LazyLog: First shared log to offer linearizable ordering across shards with low latency by deferring ordering
- Enabled by our new observations about modern shared-log applications

Outline

Introduction

Motivation

LazyLog Insight and Interface

LazyLog System Design

Performance Evaluation

System Design

System Design

Designed an implementation of the LazyLog interface: Erwin

System Design

Designed an implementation of the LazyLog interface: Erwin

Offers linearizable ordering across shards with 1-RTT appends

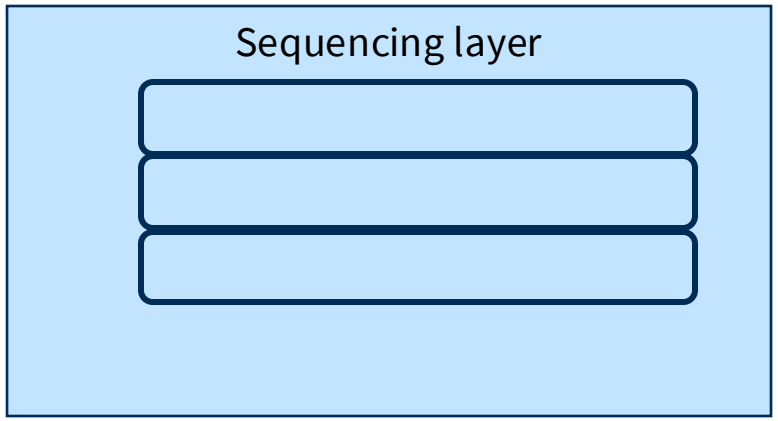
System Design

Designed an implementation of the LazyLog interface: Erwin

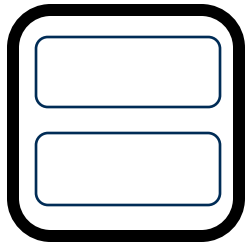
Offers linearizable ordering across shards with 1-RTT appends

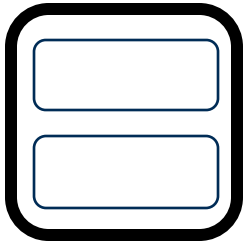
Offers about a million 4KB appends/sec on our testbed

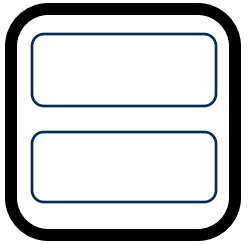
Erwin's Goal: 1-RTT Append



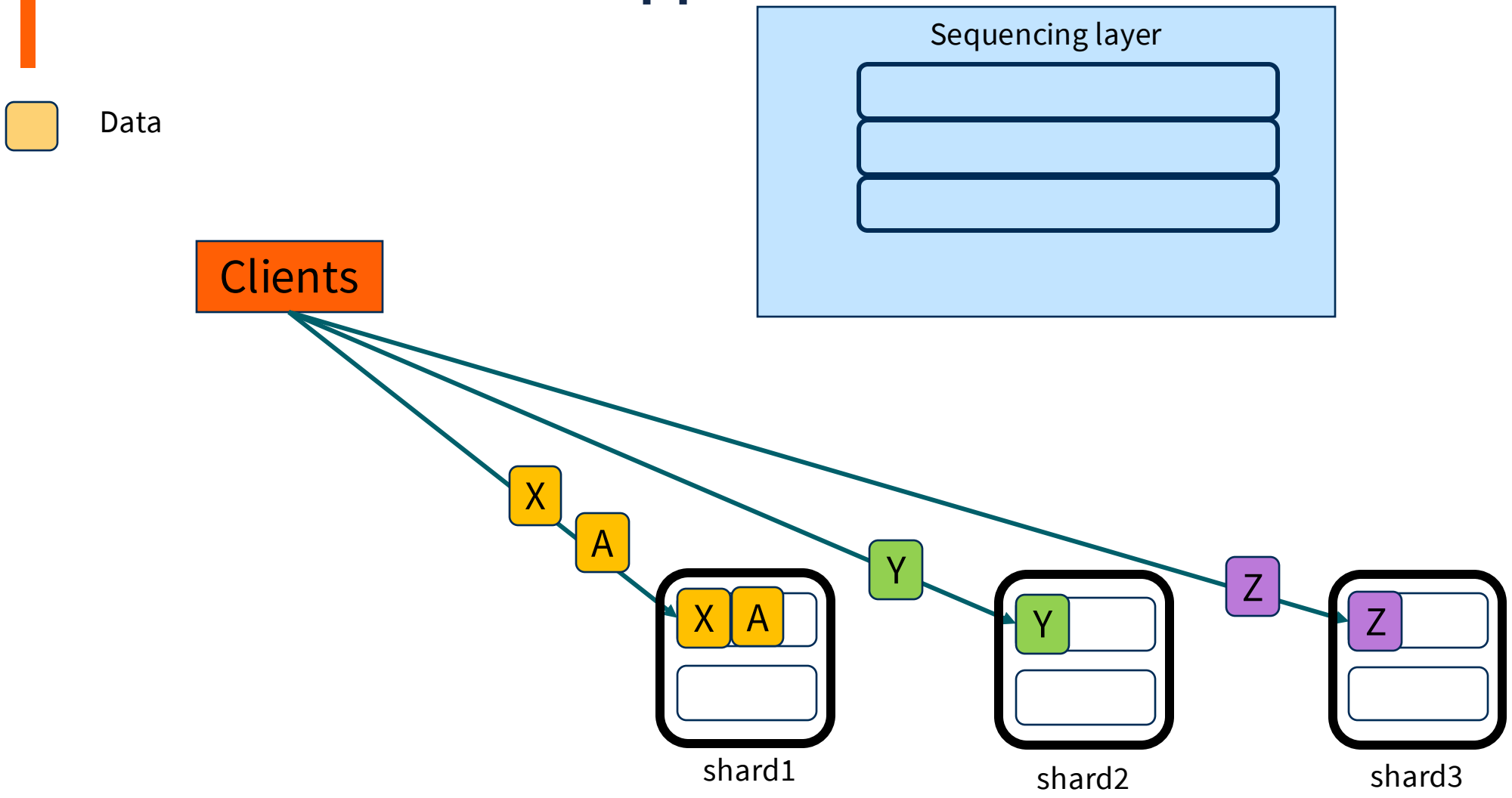
Clients


shard1

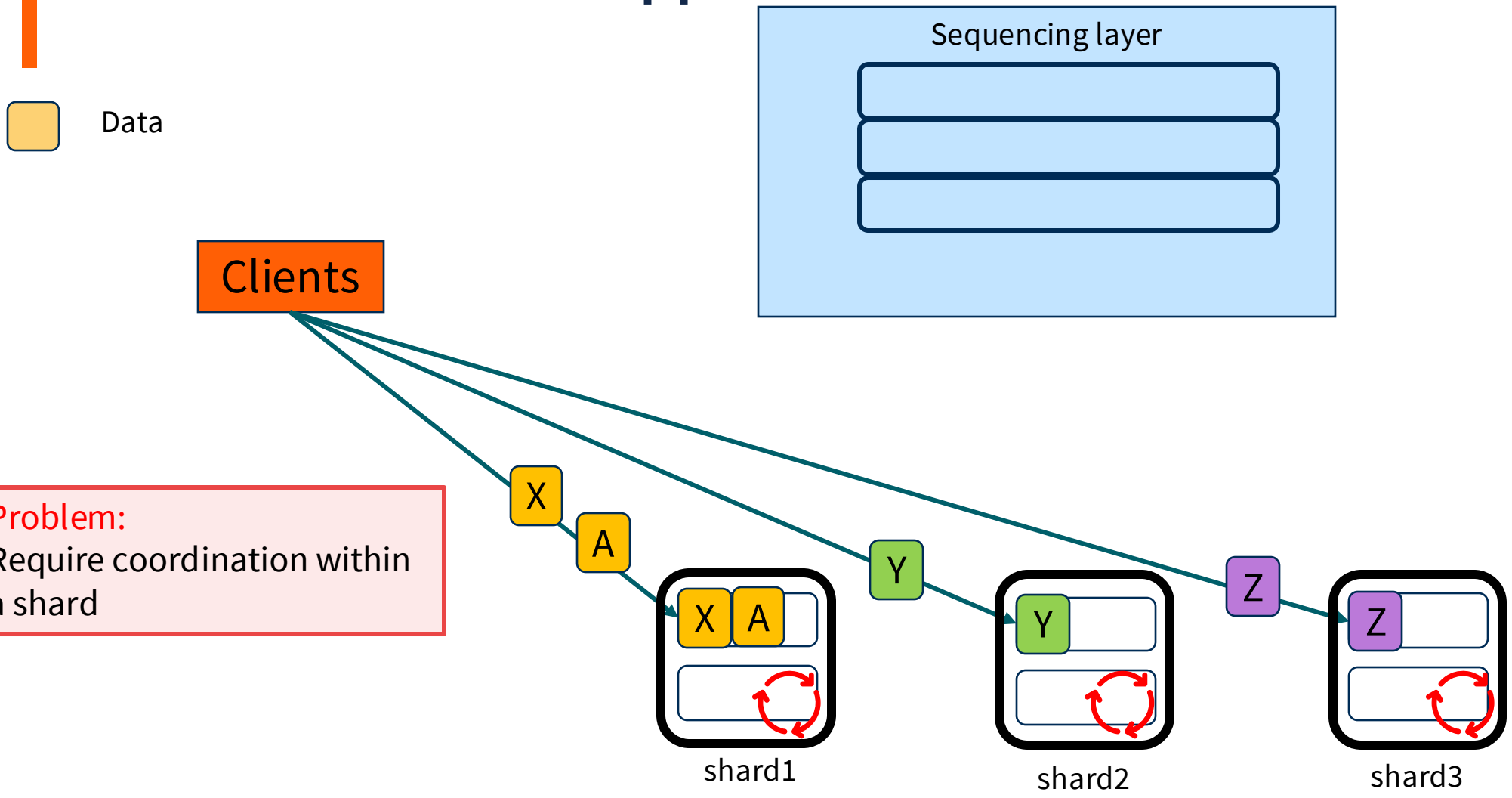

shard2


shard3

Erwin's Goal: 1-RTT Append

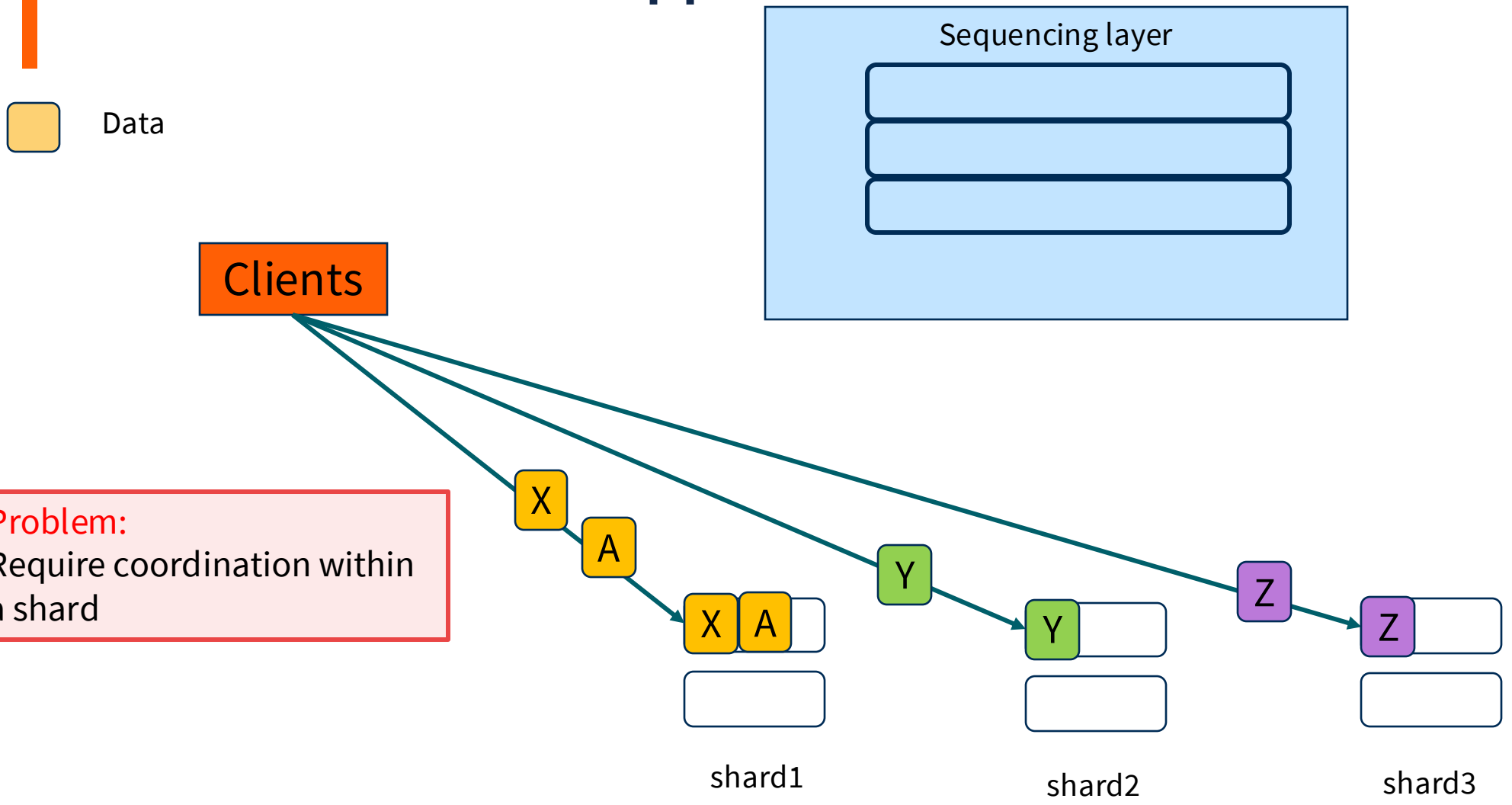


Erwin's Goal: 1-RTT Append



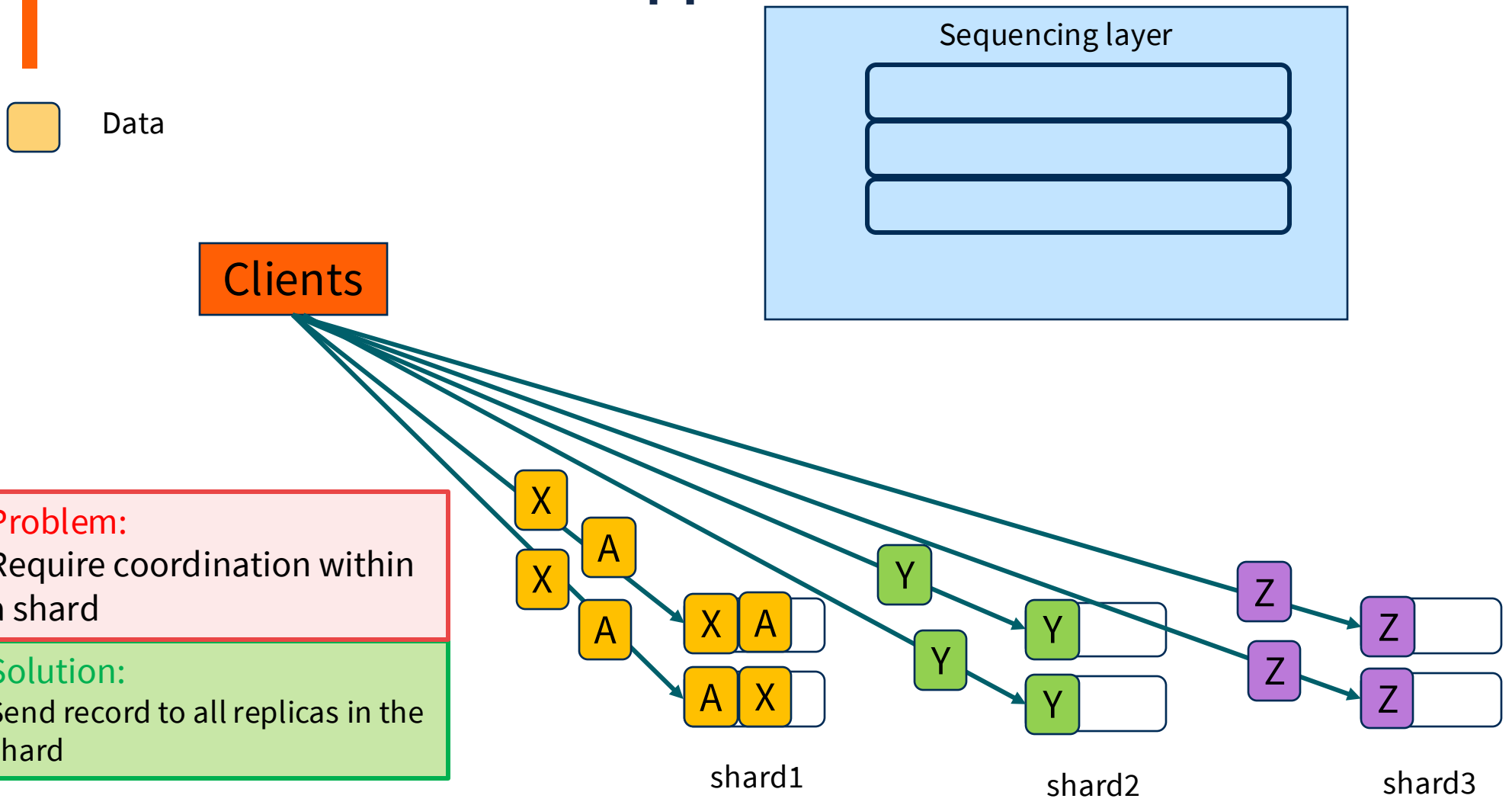
Problem:
Require coordination within a shard

Erwin's Goal: 1-RTT Append



Problem:
Require coordination within a shard

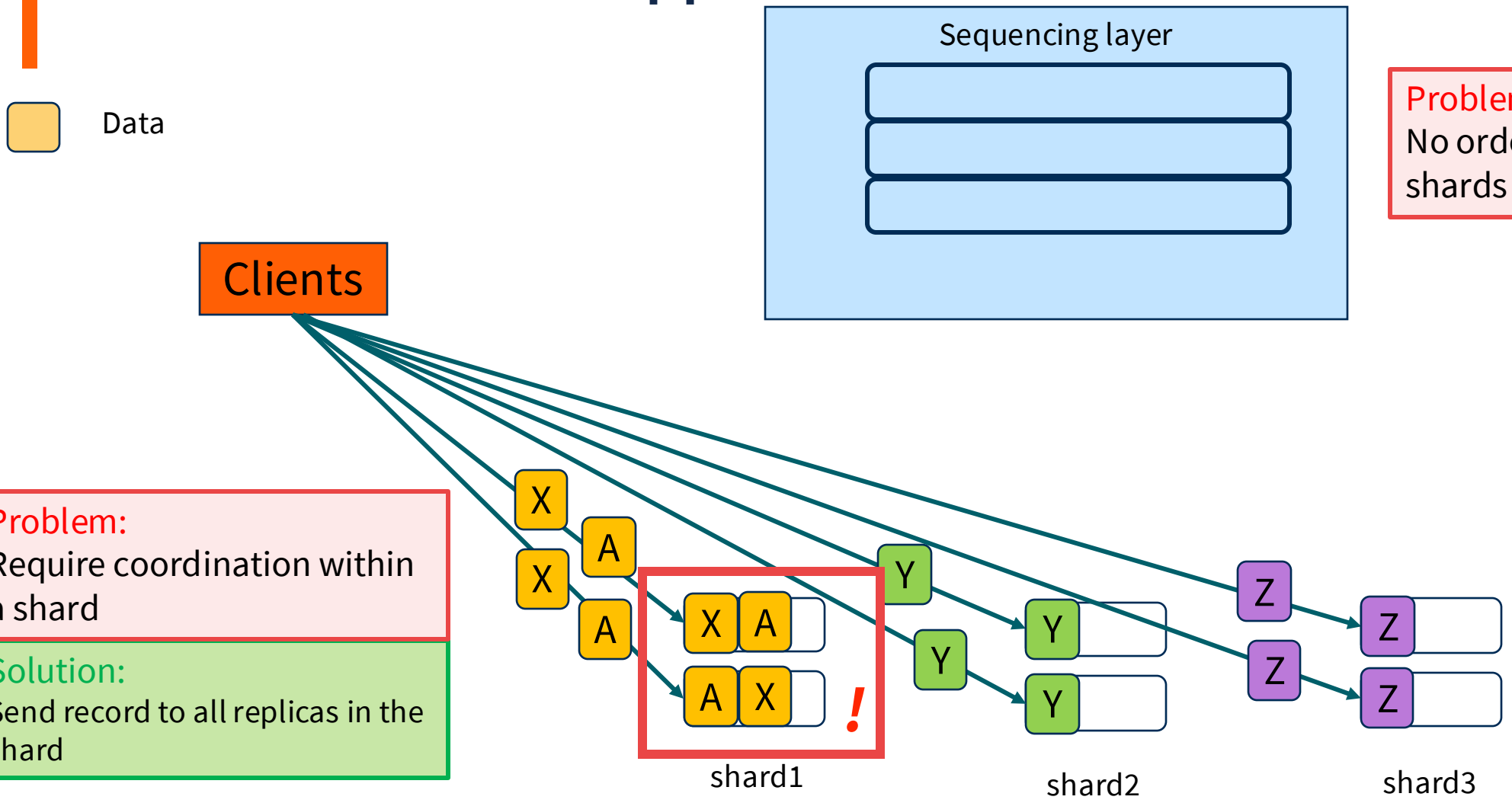
Erwin's Goal: 1-RTT Append



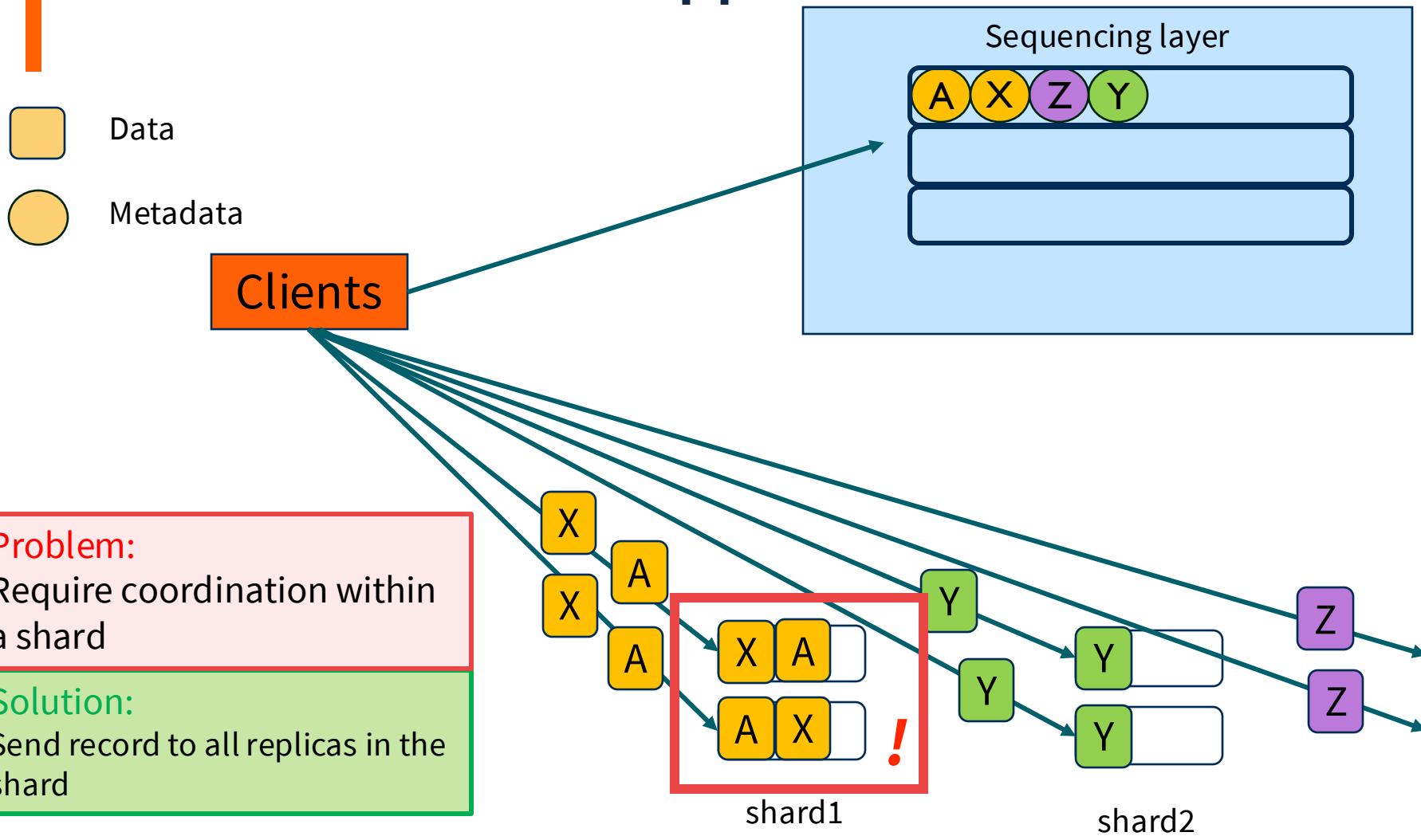
Problem:
Require coordination within a shard

Solution:
Send record to all replicas in the shard

Erwin's Goal: 1-RTT Append



Erwin's Goal: 1-RTT Append



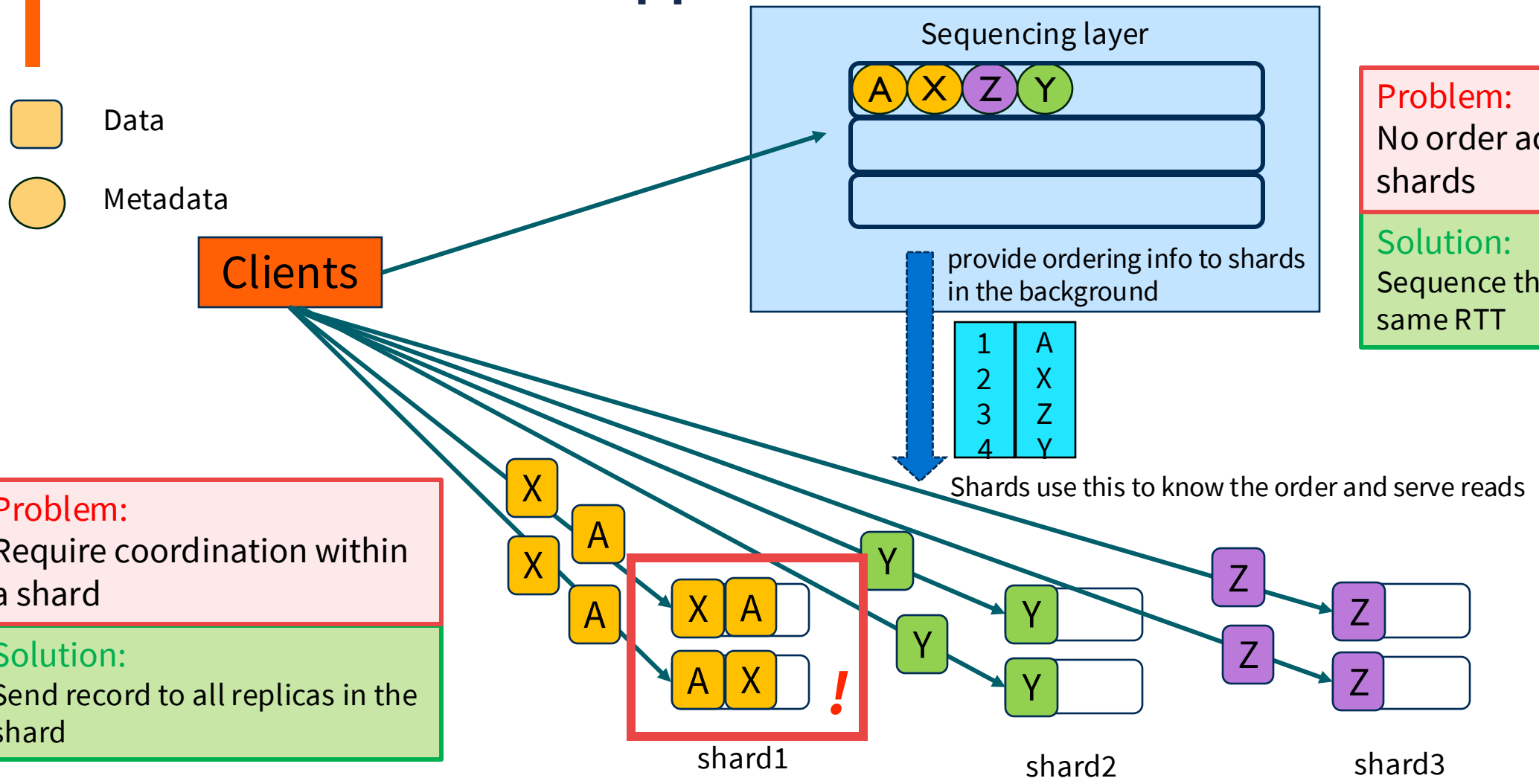
Problem:
No order across and within shards

Solution:
Sequence the metadata in the same RTT

Problem:
Require coordination within a shard

Solution:
Send record to all replicas in the shard

Erwin's Goal: 1-RTT Append



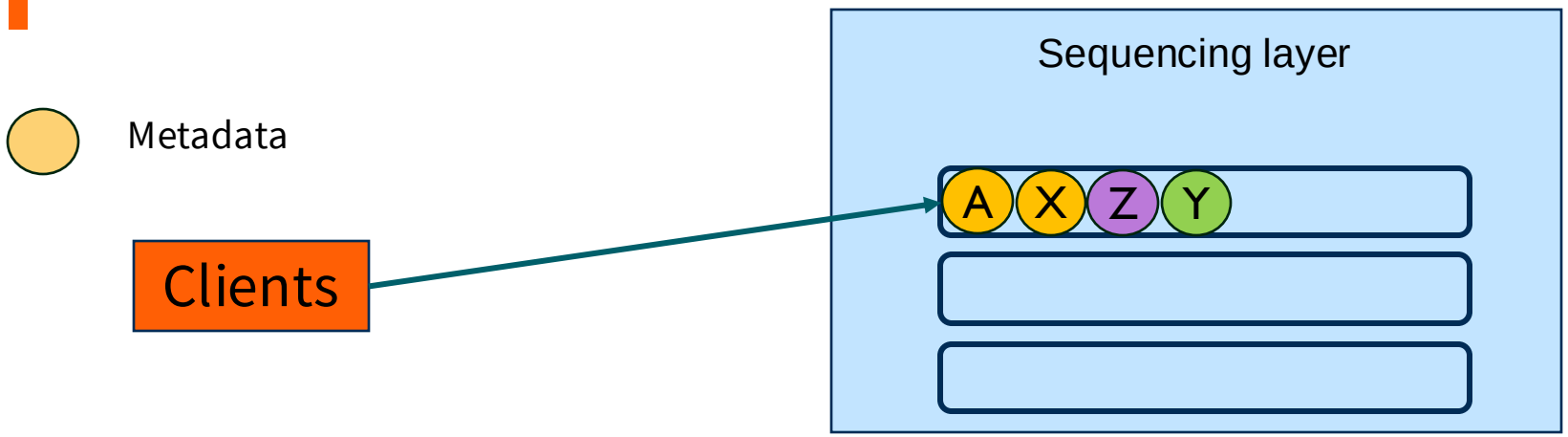
Problem:
No order across and within shards

Solution:
Sequence the metadata in the same RTT

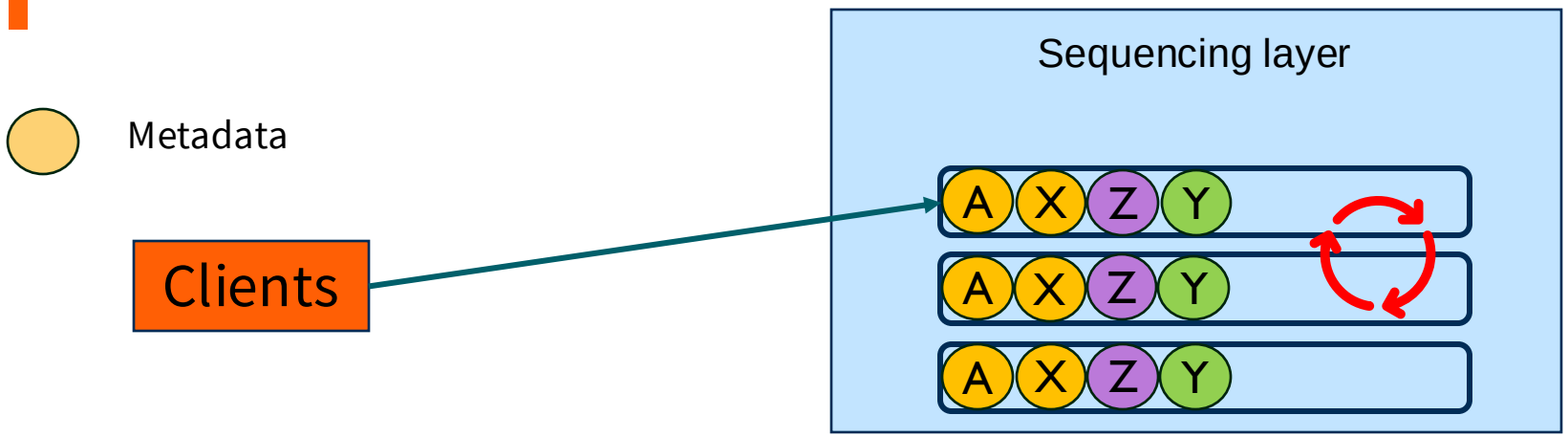
Problem:
Require coordination within a shard

Solution:
Send record to all replicas in the shard

Erwin: 1-RTT Append

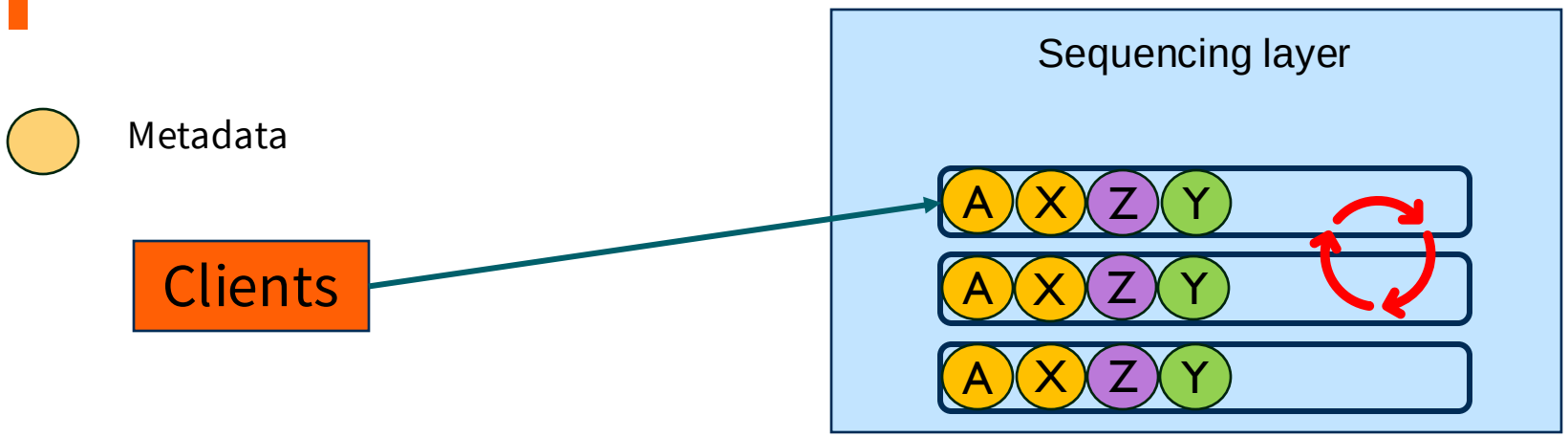


Erwin: 1-RTT Append



Problem:
Sequencing layer must run consensus to make ordering fault-tolerant → Incurs coordination within replicas

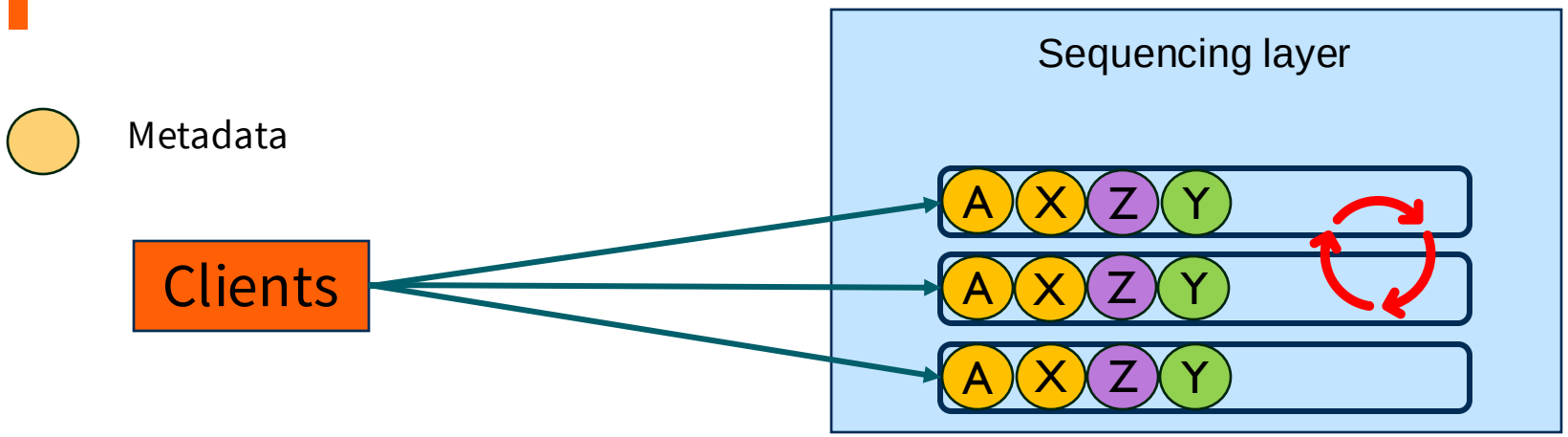
Erwin: 1-RTT Append



Problem:
Sequencing layer must run consensus to make ordering fault-tolerant → Incurs coordination within replicas

Solution:
Coordination-free sequencing

Erwin: 1-RTT Append

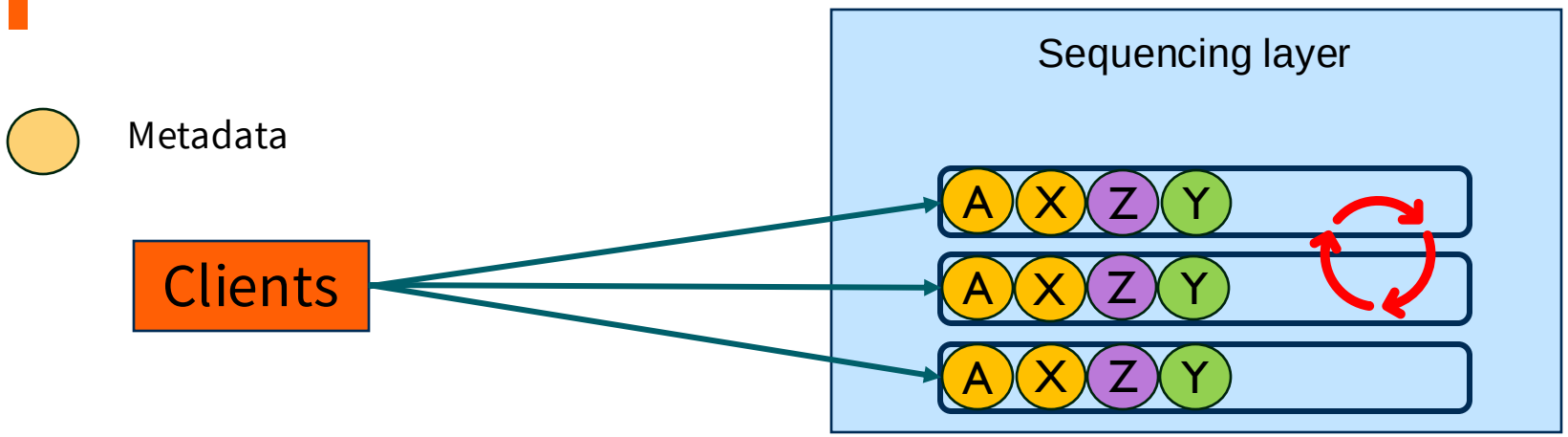


Problem:
Sequencing layer must run consensus to make ordering fault-tolerant → Incurs coordination within replicas

Solution:
Coordination-free sequencing

- Clients write to shard replicas in 1RTT; in same RTT, write metadata to **all** seq replicas

Erwin: 1-RTT Append

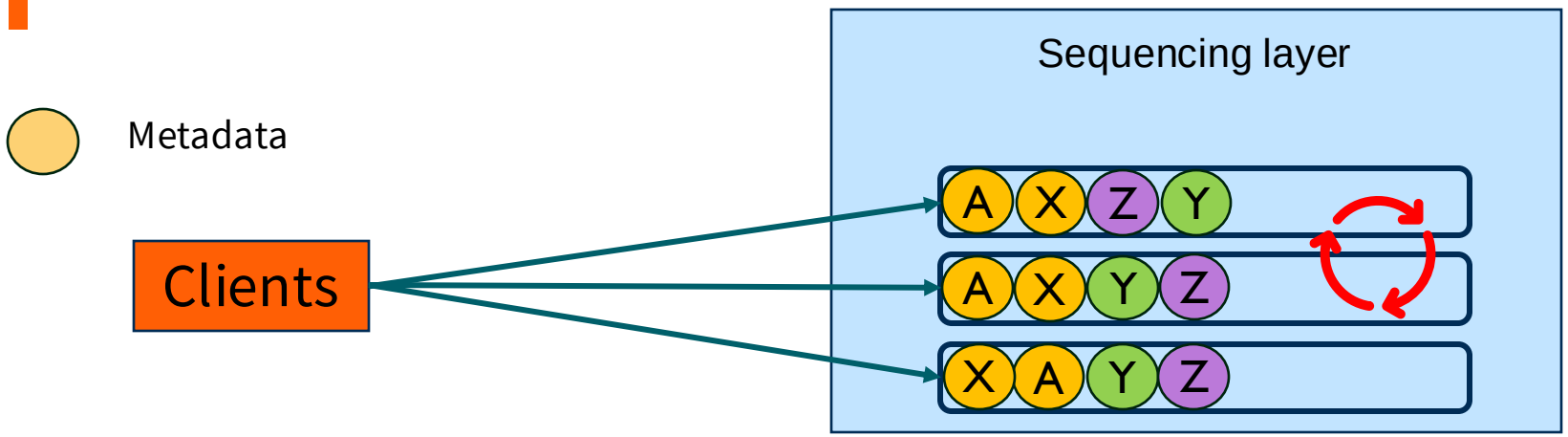


Problem:
Sequencing layer must run consensus to make ordering fault-tolerant → Incurs coordination within replicas

Solution:
Coordination-free sequencing

- Clients write to shard replicas in 1RTT; in same RTT, write metadata to **all** seq replicas
  - Appends complete in 1 RTT

Erwin: 1-RTT Append

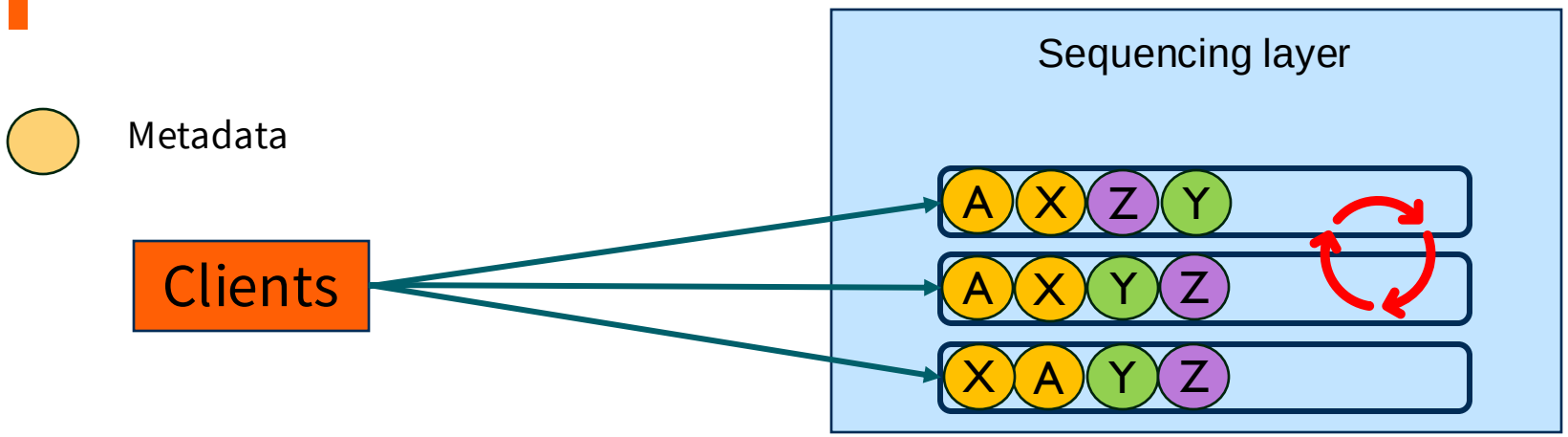


Problem:
Sequencing layer must run consensus to make ordering fault-tolerant → Incurs coordination within replicas

Solution:
Coordination-free sequencing

- Clients write to shard replicas in 1RTT; in same RTT, write metadata to **all** seq replicas
  - Appends complete in 1 RTT

Erwin: 1-RTT Append

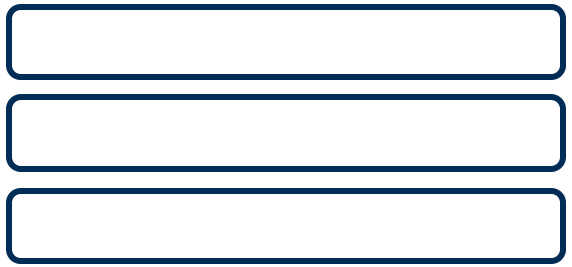


Problem:
Sequencing layer must run consensus to make ordering fault-tolerant → Incurs coordination within replicas

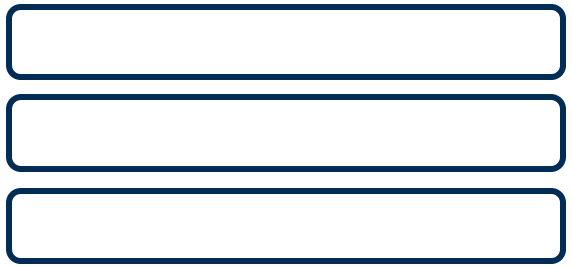
Solution:
Coordination-free sequencing

- Clients write to shard replicas in 1RTT; in same RTT, write metadata to **all** seq replicas
  - Appends complete in 1 RTT
- Erwin allows different orders across sequencing replicas
  - but without violating the linearizability

Correct Lazy Sequencing

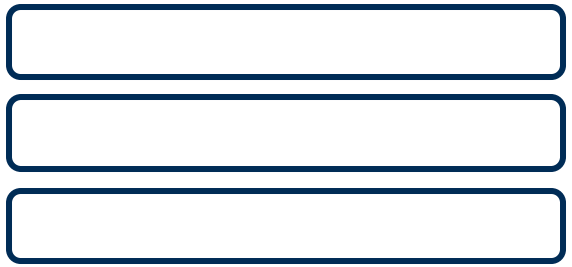


Correct Lazy Sequencing



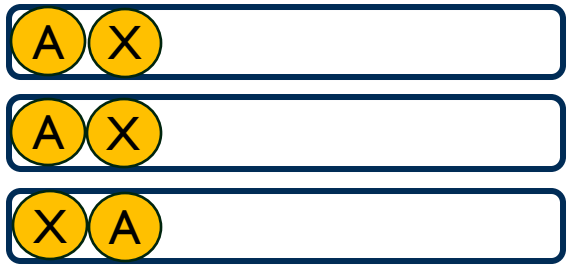
- Intuition: If append(B) follows append(A) in real-time, all logs will capture that dependency

Correct Lazy Sequencing



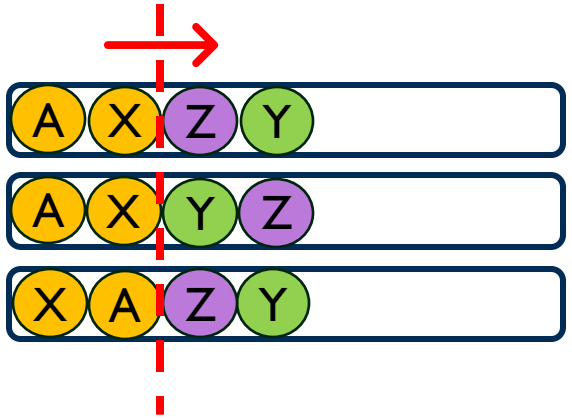
- Intuition: If append(B) follows append(A) in real-time, all logs will capture that dependency
- Example: actual ordering is (A|||X) → (Z|||Y): Captured by all replicas

Correct Lazy Sequencing



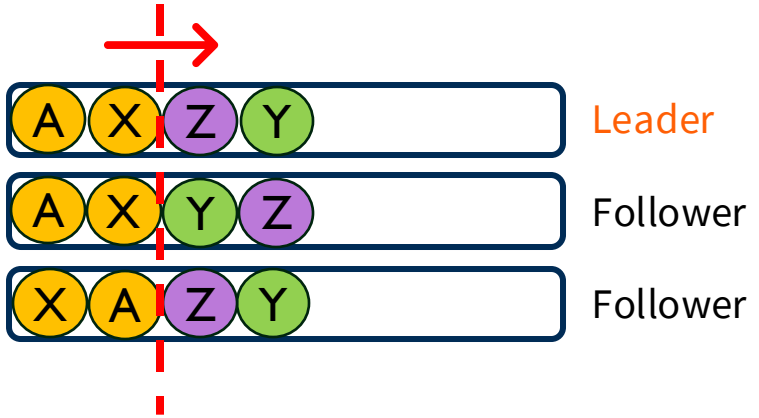
- Intuition: If append(B) follows append(A) in real-time, all logs will capture that dependency
- Example: actual ordering is (A|||X) → (Z|||Y): Captured by all replicas

Correct Lazy Sequencing



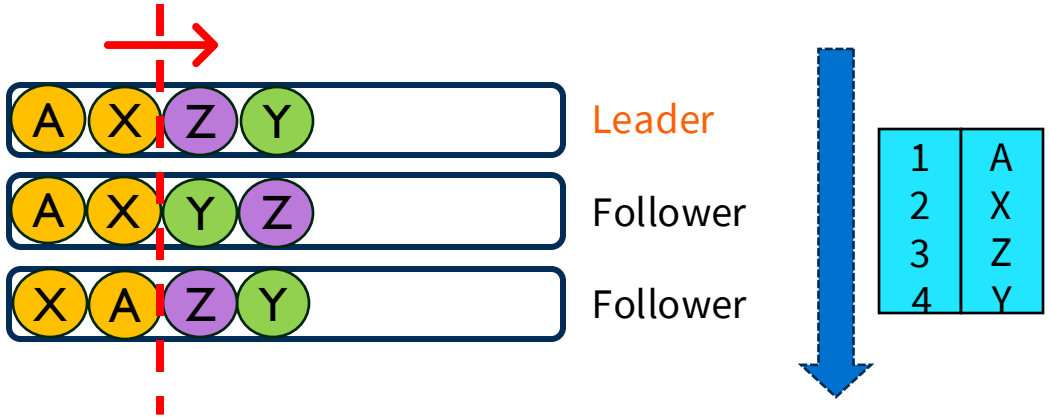
- Intuition: If append(B) follows append(A) in real-time, all logs will capture that dependency
- Example: actual ordering is (A|||X) → (Z|||Y): Captured by all replicas

Correct Lazy Sequencing



- Intuition: If append(B) follows append(A) in real-time, all logs will capture that dependency
- Example: actual ordering is (A|||X) → (Z|||Y): Captured by all replicas
- Assign one sequencing replica as leader to decide the order

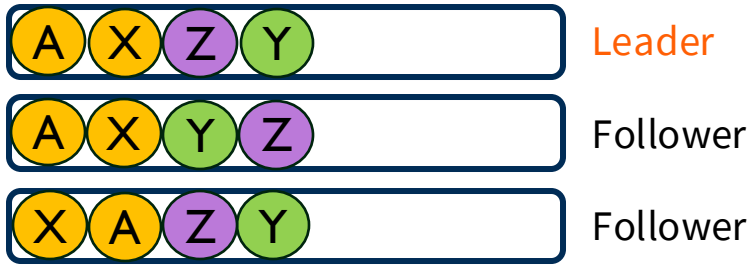
Correct Lazy Sequencing



- Intuition: If append(B) follows append(A) in real-time, all logs will capture that dependency
- Example: actual ordering is (A|||X) → (Z|||Y): Captured by all replicas
- Assign one sequencing replica as leader to decide the order
- Send leader's order to shards

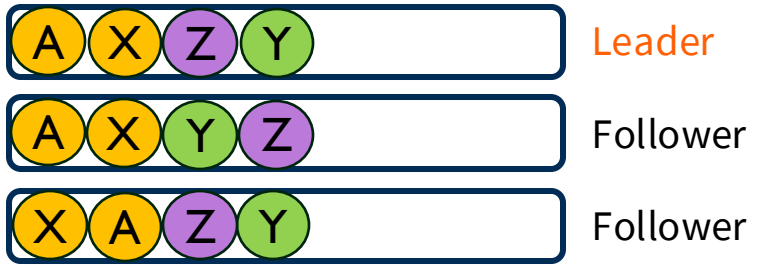
Handle Sequencing Leader Failure

Exposed order must be preserved upon failures!



Handle Sequencing Leader Failure

Exposed order must be preserved upon failures!

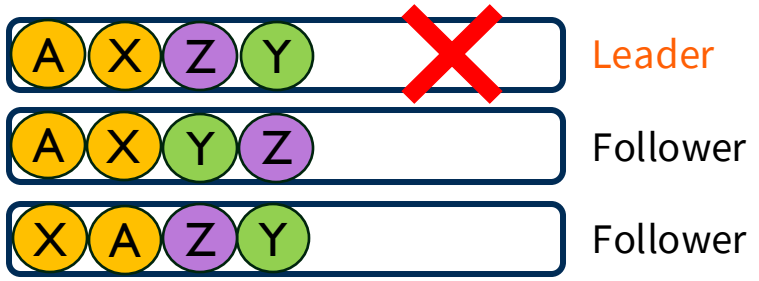


1	A
2	X
3	Z
4	Y

- Shards have served reads with this order

Handle Sequencing Leader Failure

Exposed order must be preserved upon failures!

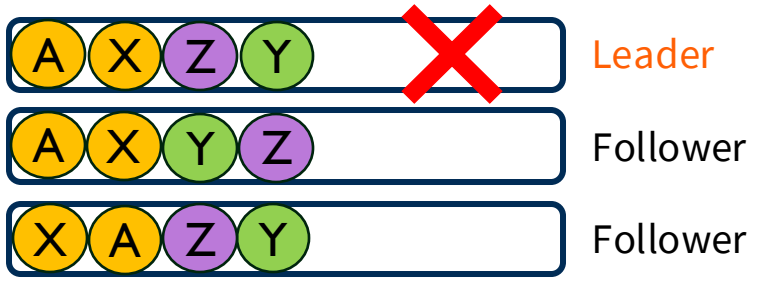


1	A
2	X
3	Z
4	Y

- Shards have served reads with this order

Handle Sequencing Leader Failure

Exposed order must be preserved upon failures!

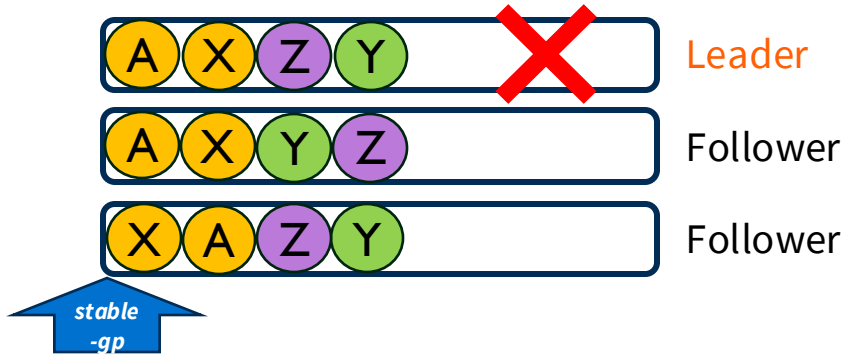


1	A
2	X
3	Z
4	Y

- Shards have served reads with this order
- Must preserve the exposed order for future reads

Handle Sequencing Leader Failure

Exposed order must be preserved upon failures!



1	A
2	X
3	Z
4	Y

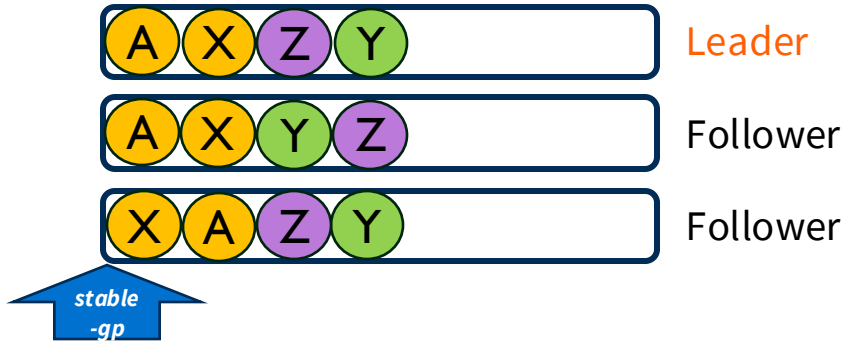
- Shards have served reads with this order
- Must preserve the exposed order for future reads

Intuition:

- ***stable-gp*** invariant: records for pos before ***stable-gp*** are stable and remain unchanged regardless of future failures
- Only positions up to ***stable-gp*** are exposed to readers

Handle Sequencing Leader Failure

Exposed order must be preserved upon failures!



1	A
2	X
3	Z
4	Y

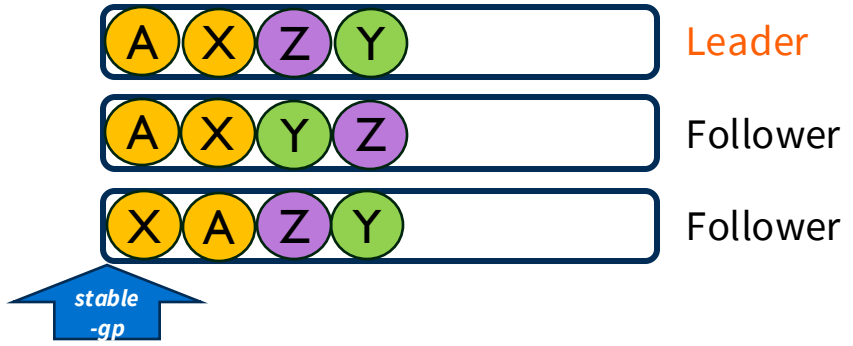
- Shards have served reads with this order
- Must preserve the exposed order for future reads

Intuition:

- ***stable-gp*** invariant: records for pos before ***stable-gp*** are stable and remain unchanged regardless of future failures
- Only positions up to ***stable-gp*** are exposed to readers

Handle Sequencing Leader Failure

Exposed order must be preserved upon failures!

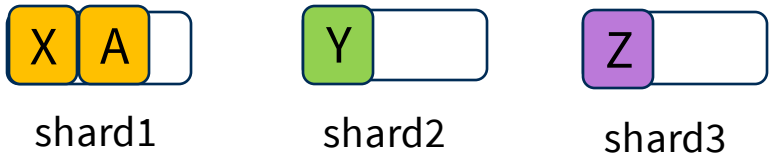


1	A
2	X
3	Z
4	Y

- Shards have served reads with this order
- Must preserve the exposed order for future reads

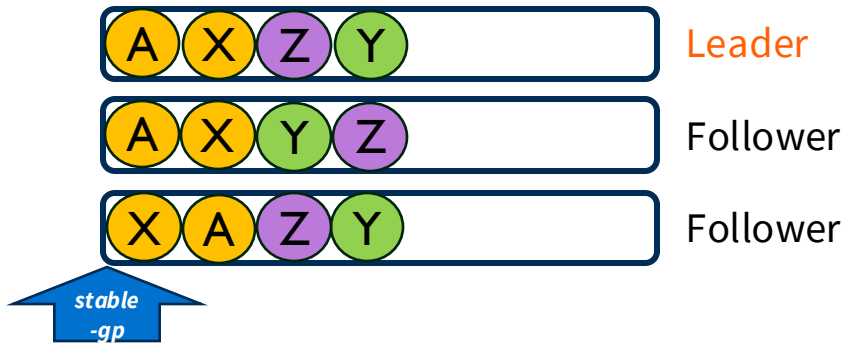
Intuition:

- ***stable-gp*** invariant: records for pos before ***stable-gp*** are stable and remain unchanged regardless of future failures
- Only positions up to ***stable-gp*** are exposed to readers
- Advance ***stable-gp*** only after



Handle Sequencing Leader Failure

Exposed order must be preserved upon failures!

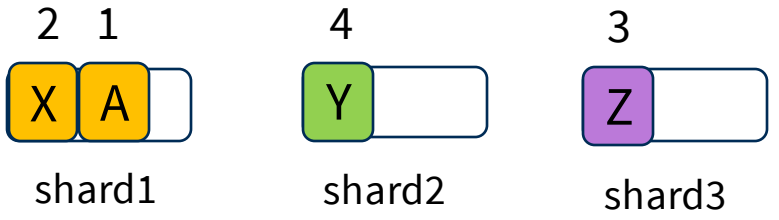


1	A
2	X
3	Z
4	Y

- Shards have served reads with this order
- Must preserve the exposed order for future reads

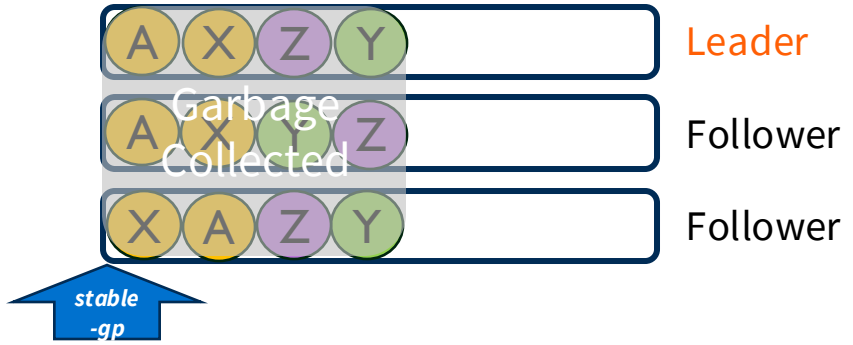
Intuition:

- ***stable-gp*** invariant: records for pos before ***stable-gp*** are stable and remain unchanged regardless of future failures
- Only positions up to ***stable-gp*** are exposed to readers
- Advance ***stable-gp*** only after
  - shards acknowledge order and



Handle Sequencing Leader Failure

Exposed order must be preserved upon failures!

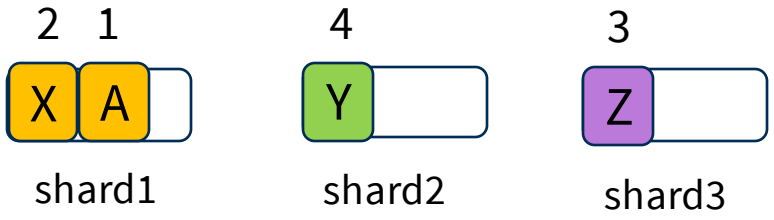


1	A
2	X
3	Z
4	Y

- Shards have served reads with this order
- Must preserve the exposed order for future reads

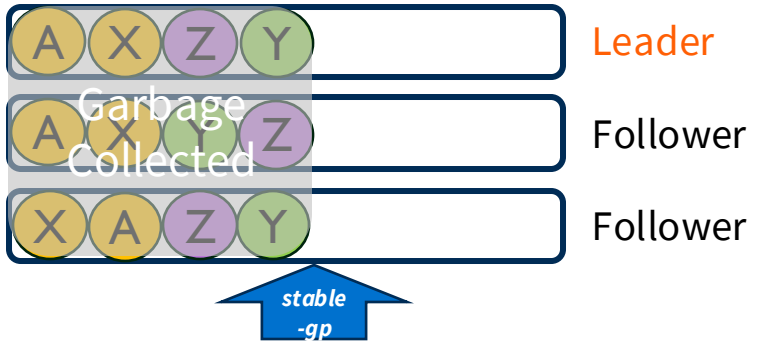
Intuition:

- ***stable-gp*** invariant: records for pos before ***stable-gp*** are stable and remain unchanged regardless of future failures
- Only positions up to ***stable-gp*** are exposed to readers
- Advance ***stable-gp*** only after
  - shards acknowledge order and
  - entries on all sequencing replicas are garbage-collected



Handle Sequencing Leader Failure

Exposed order must be preserved upon failures!

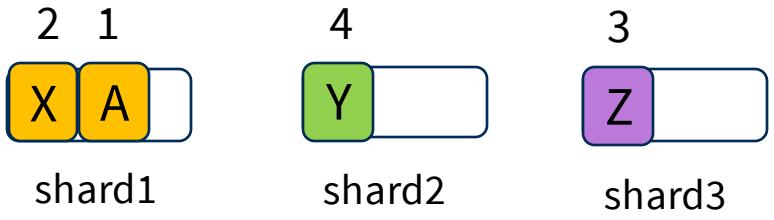


1	A
2	X
3	Z
4	Y

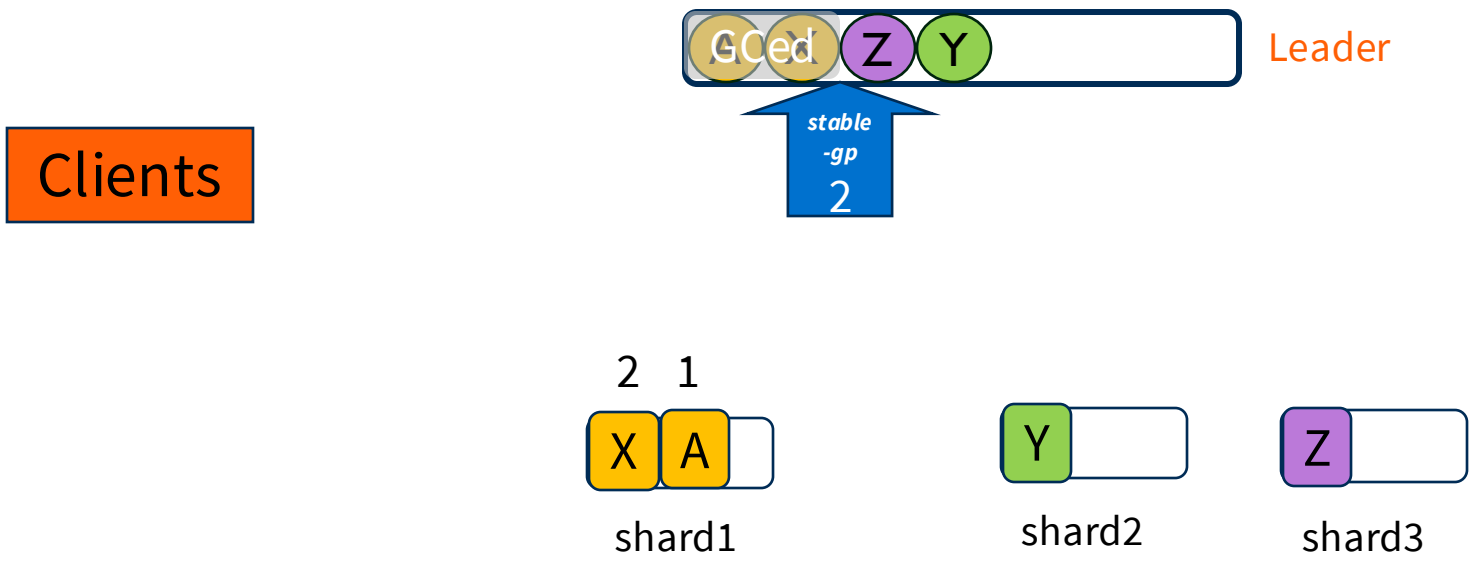
- Shards have served reads with this order
- Must preserve the exposed order for future reads

Intuition:

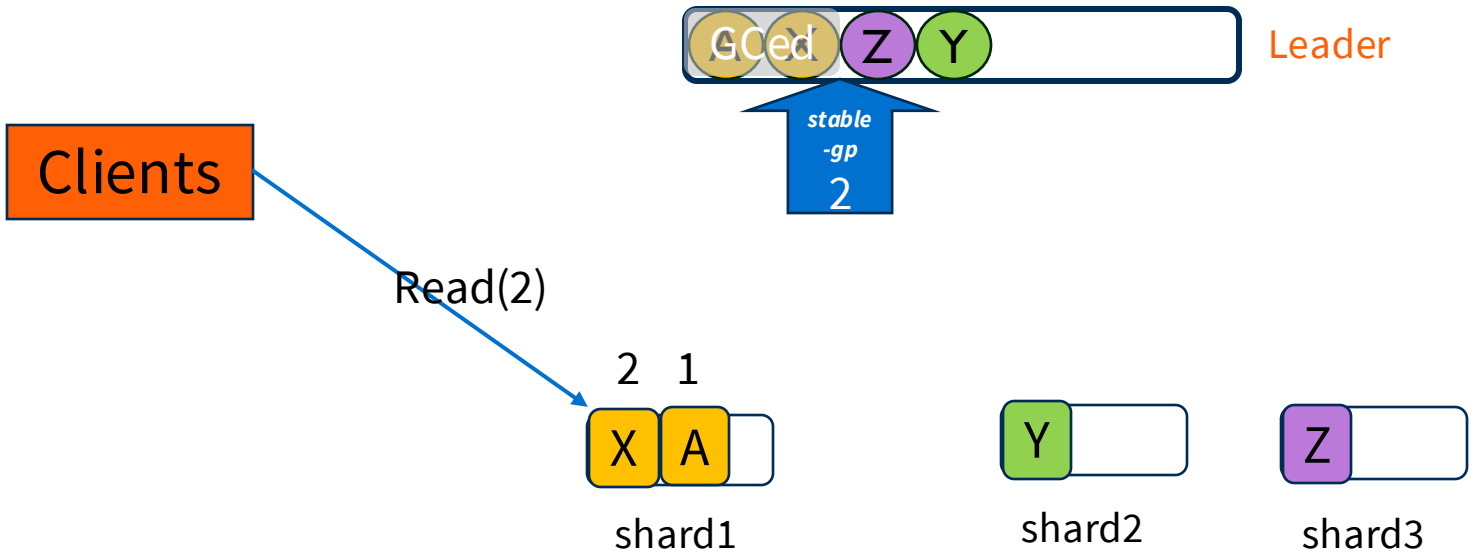
- ***stable-gp*** invariant: records for pos before ***stable-gp*** are stable and remain unchanged regardless of future failures
- Only positions up to ***stable-gp*** are exposed to readers
- Advance ***stable-gp*** only after
  - shards acknowledge order and
  - entries on all sequencing replicas are garbage-collected



Erwin: Read

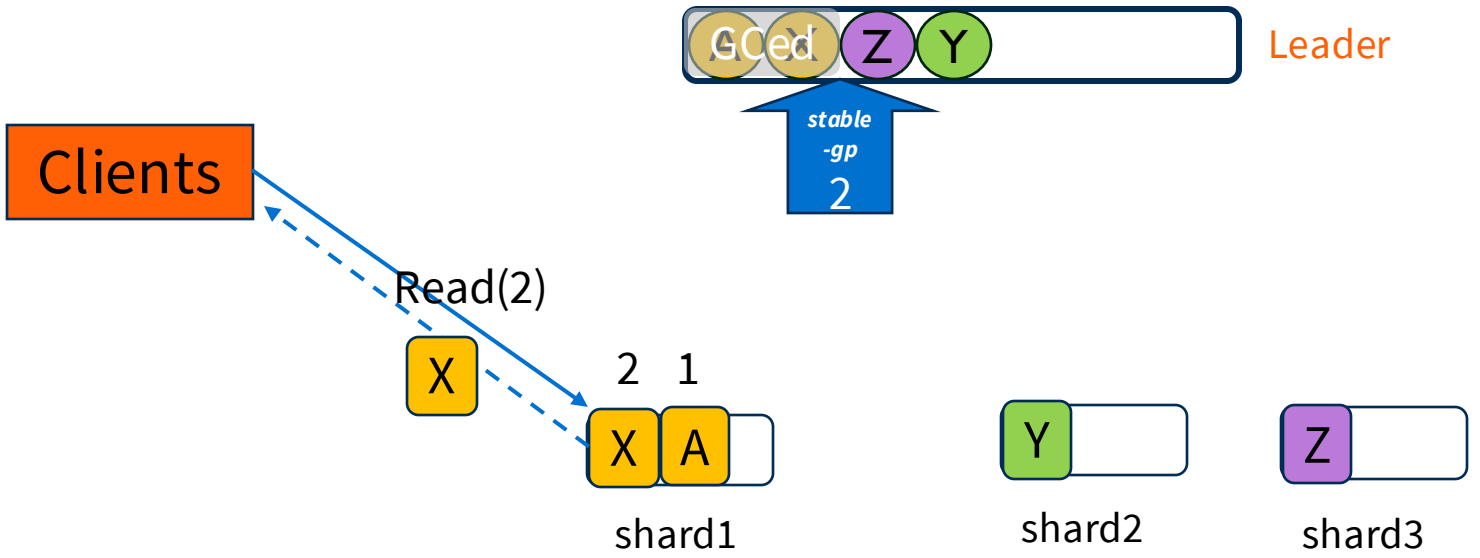

Clients
GCed
Z
Y
Leader
stable
-gp
2
2
1
X
A
shard1
Y
shard2
Z
shard3

Erwin: Read



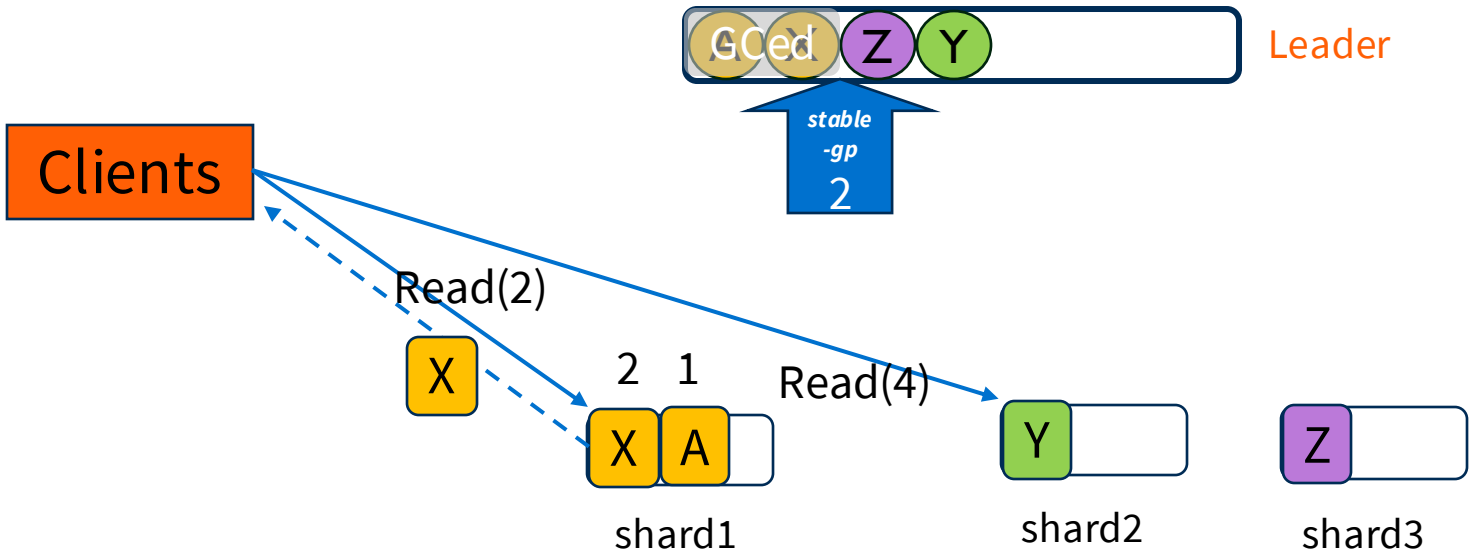
- Reading ordered position (fast read): entry returned directly

Erwin: Read



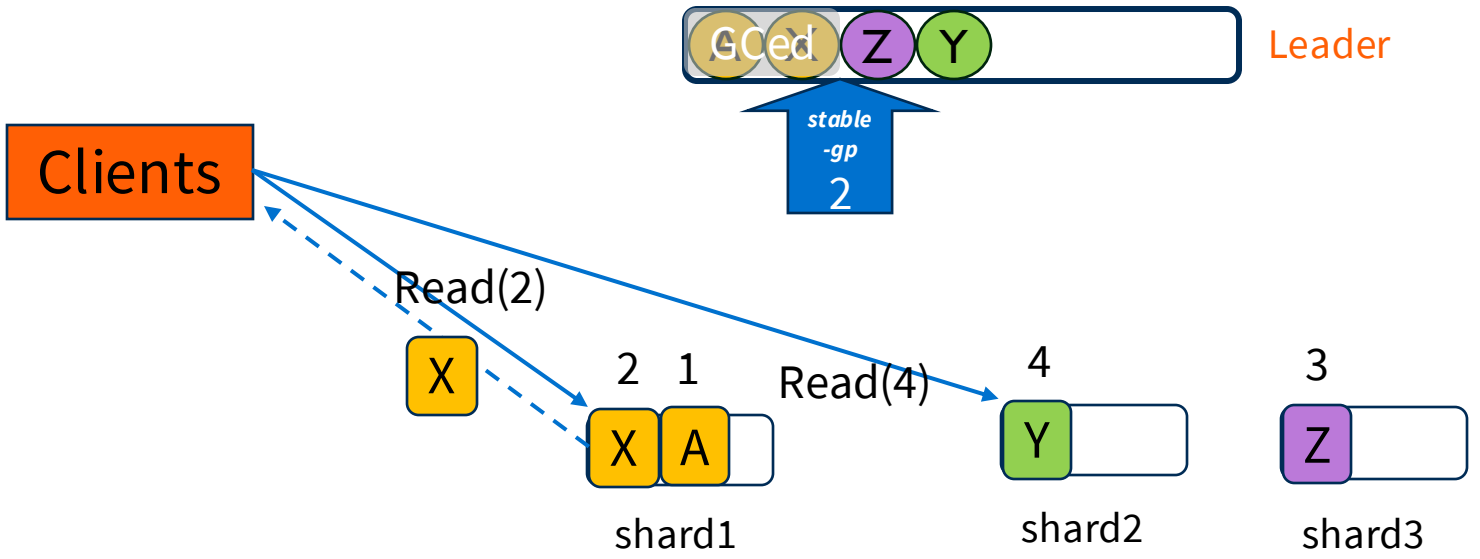
- Reading ordered position (fast read): entry returned directly

Erwin: Read



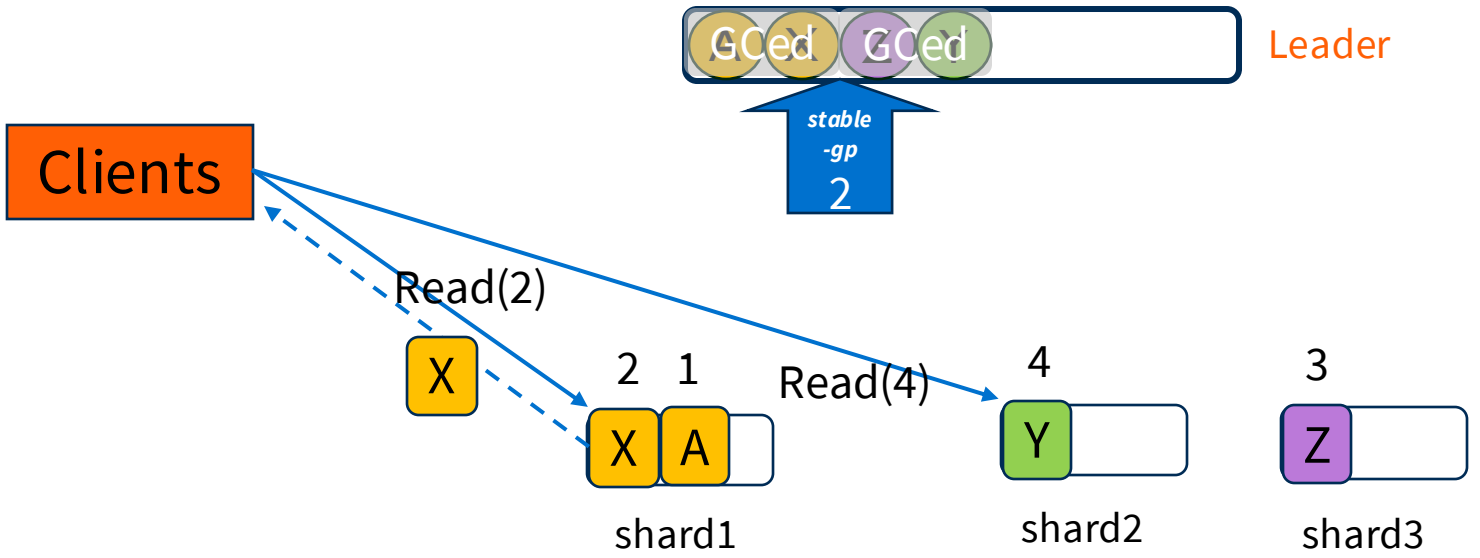
- Reading ordered position (fast read): entry returned directly
- Reading unordered position (slow read): must wait until ***stable-gp*** is advanced to the read position

Erwin: Read



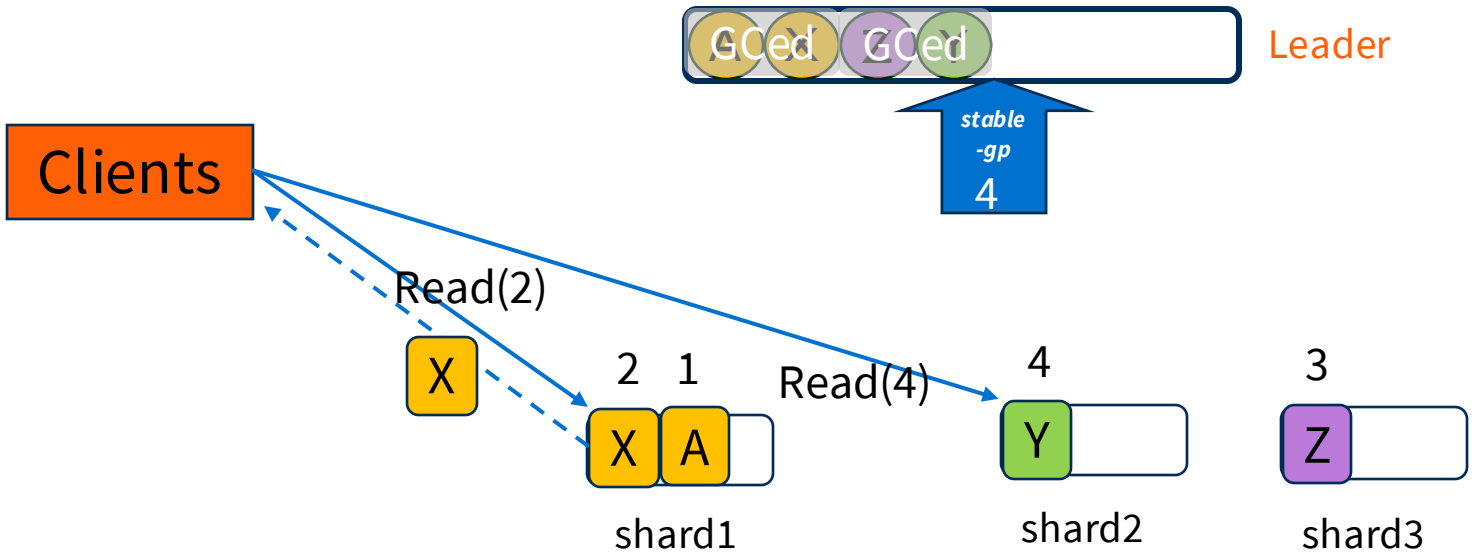
- Reading ordered position (fast read): entry returned directly
- Reading unordered position (slow read): must wait until ***stable-gp*** is advanced to the read position

Erwin: Read



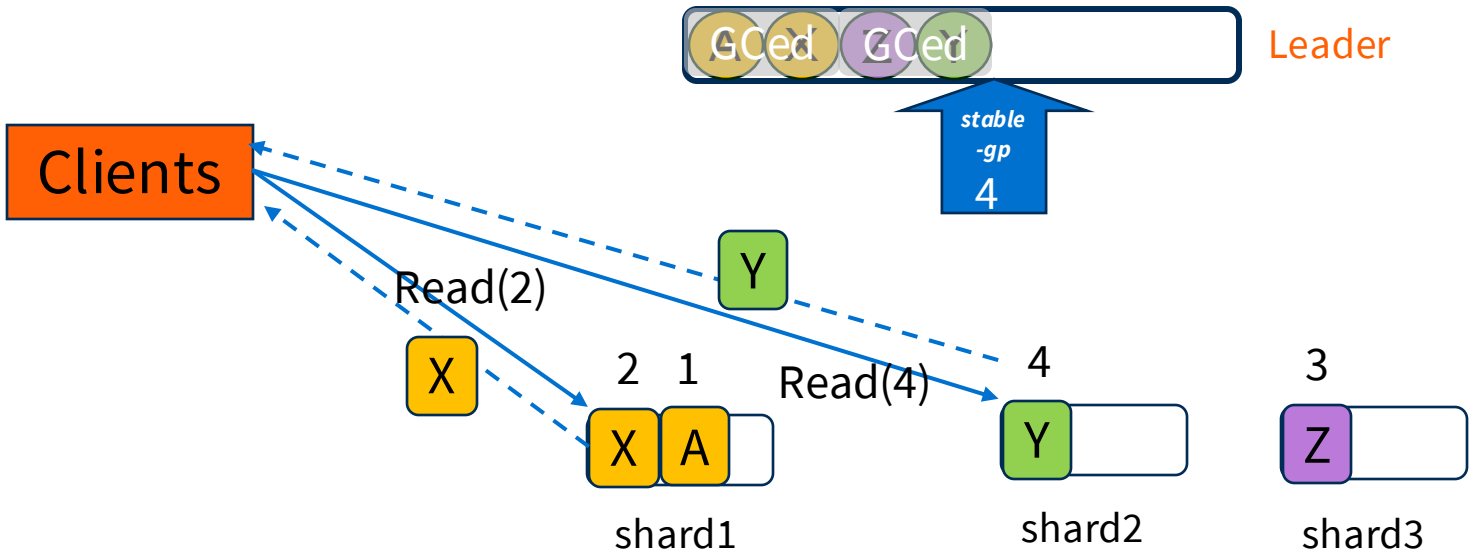
- Reading ordered position (fast read): entry returned directly
- Reading unordered position (slow read): must wait until ***stable-gp*** is advanced to the read position

Erwin: Read



- Reading ordered position (fast read): entry returned directly
- Reading unordered position (slow read): must wait until ***stable-gp*** is advanced to the read position

Erwin: Read



- Reading ordered position (fast read): entry returned directly
- Reading unordered position (slow read): must wait until ***stable-gp*** is advanced to the read position

Outline

Introduction

Motivation

LazyLog Insight and Interface

LazyLog System Design

Performance Evaluation

Performance Evaluation

- What's the latency benefit of lazy ordering?
- How do reads perform in LazyLog?
- Do end applications benefit?

What’s the Latency Benefit of Lazy Ordering?

What’s the Latency Benefit of Lazy Ordering?

Workload: 4KB record append-only

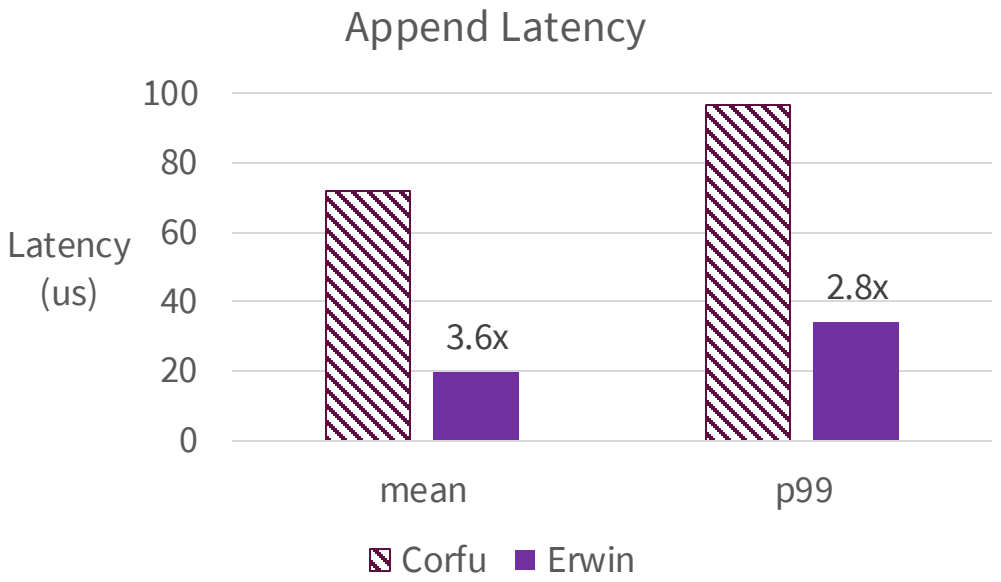
3 replicas per shard with 5 shards

What's the Latency Benefit of Lazy Ordering?

Workload: 4KB record append-only

3 replicas per shard with 5 shards

Erwin reduces append latency



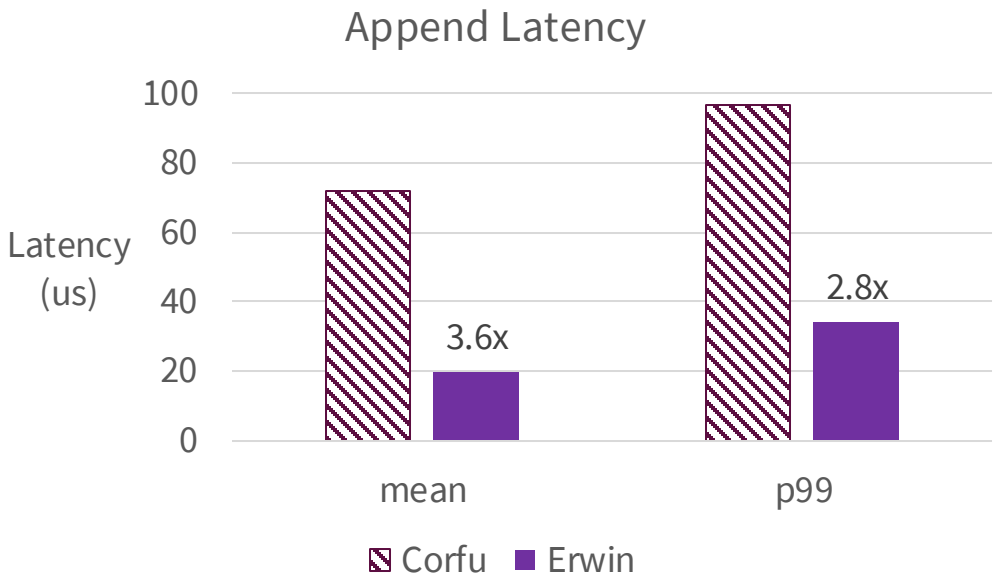
What's the Latency Benefit of Lazy Ordering?

Workload: 4KB record append-only

3 replicas per shard with 5 shards

Erwin reduces append latency

- Avg: By 3.6x compared to Corfu
- P99: By 2.8x compared to Corfu



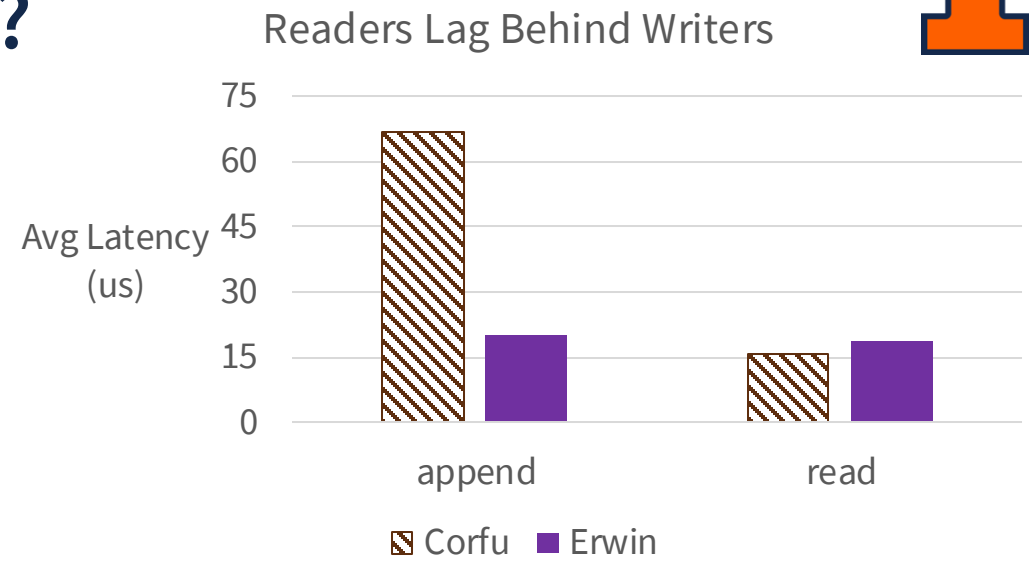
How Do Reads Perform in LazyLog?

How Do Reads Perform in LazyLog?

4KB record read after append

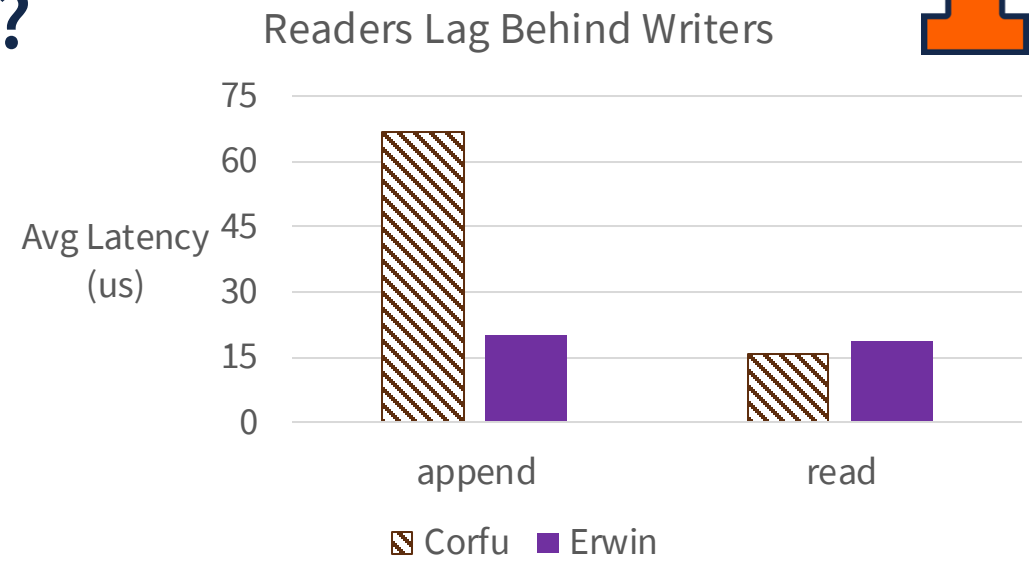
How Do Reads Perform in LazyLog?

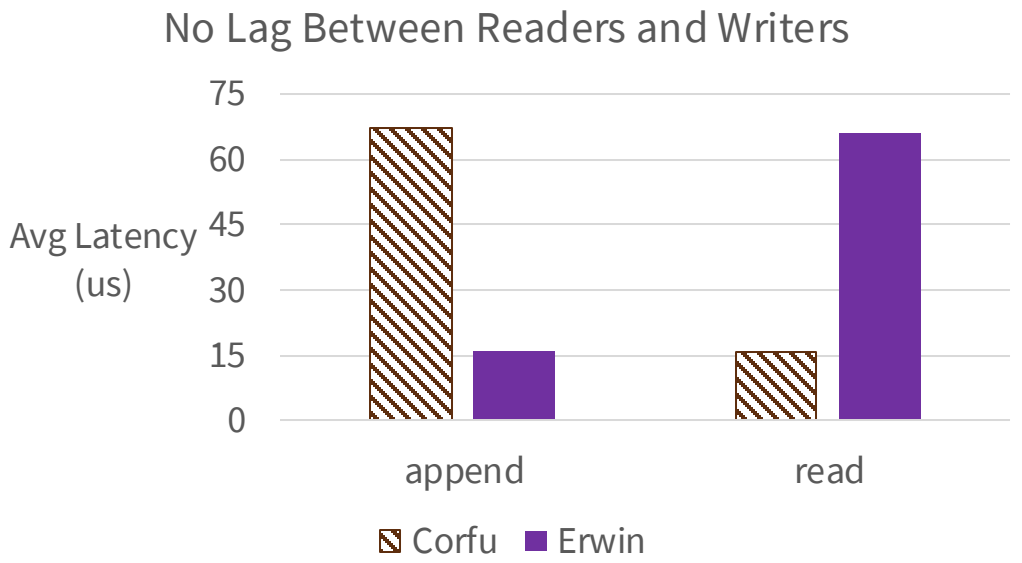
4KB record read after append



How Do Reads Perform in LazyLog?

4KB record read after append

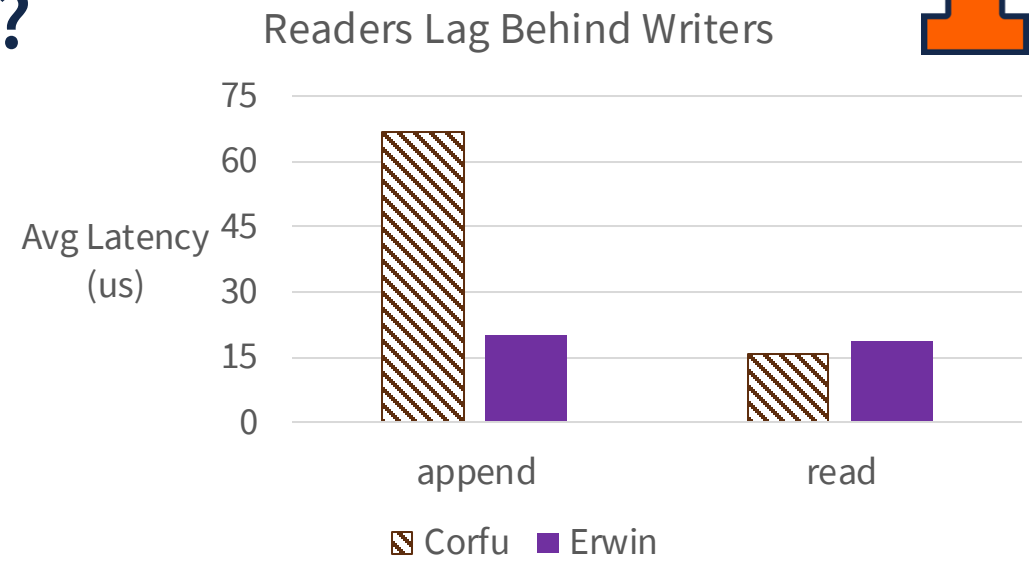


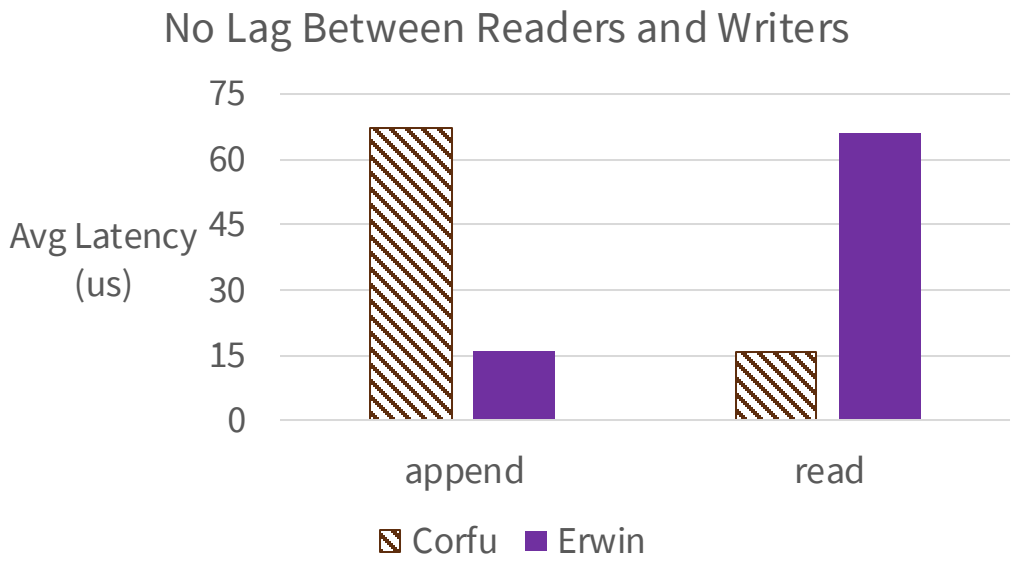


How Do Reads Perform in LazyLog?

4KB record read after append

For many applications in which reads lag behind writes:



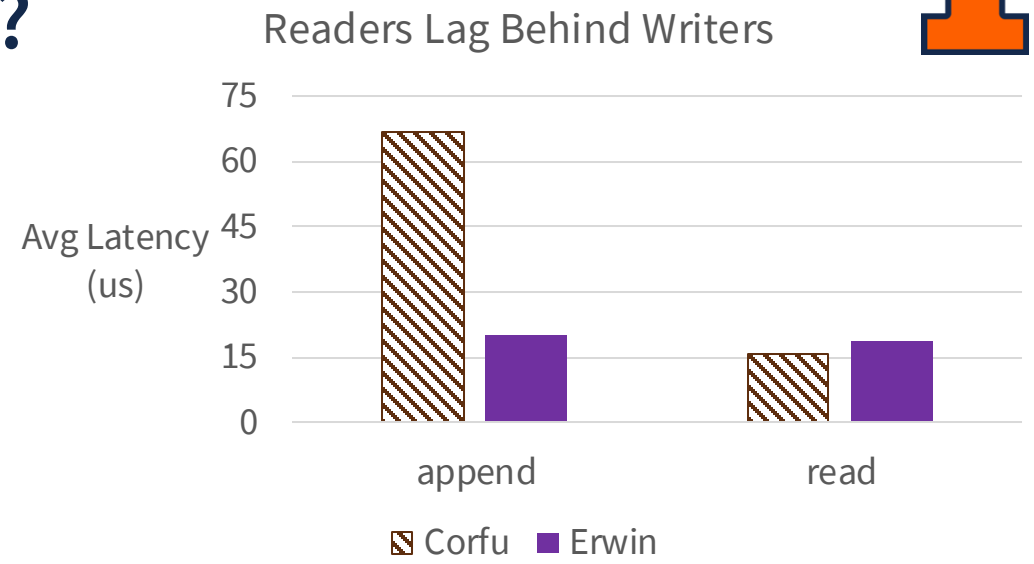


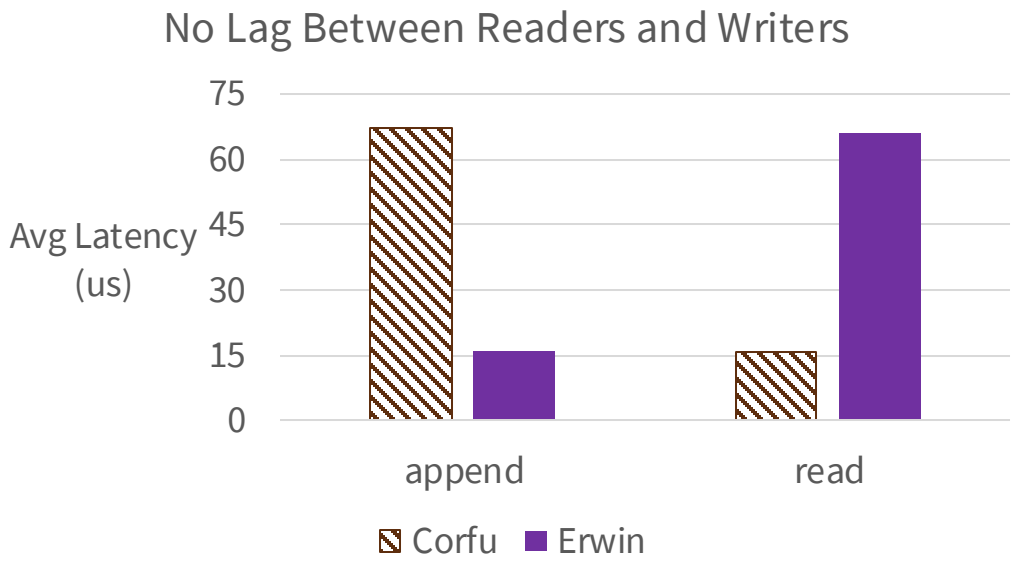
How Do Reads Perform in LazyLog?

4KB record read after append

For many applications in which reads lag behind writes:

- Erwin achieves low append latency and read latency





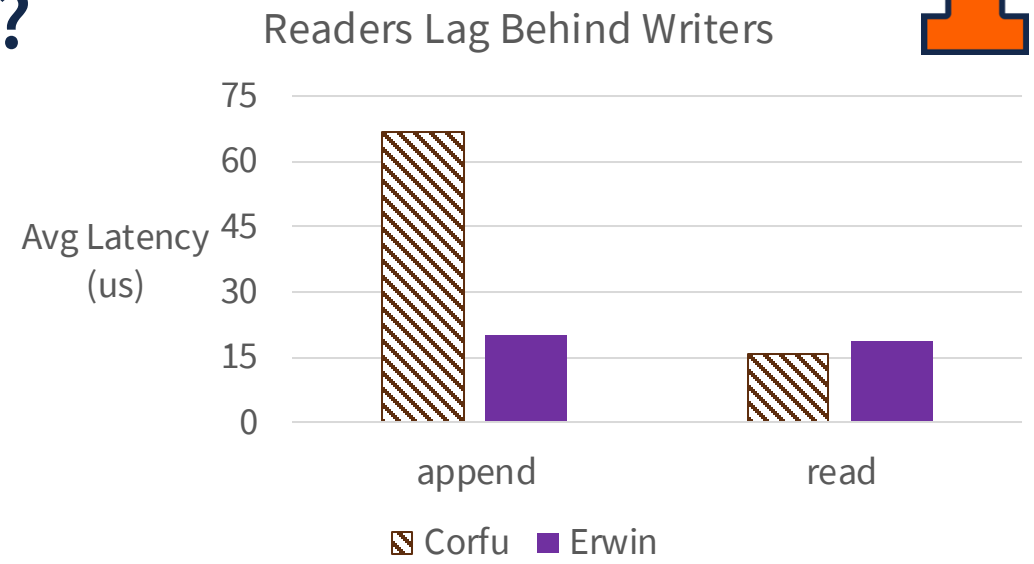
How Do Reads Perform in LazyLog?

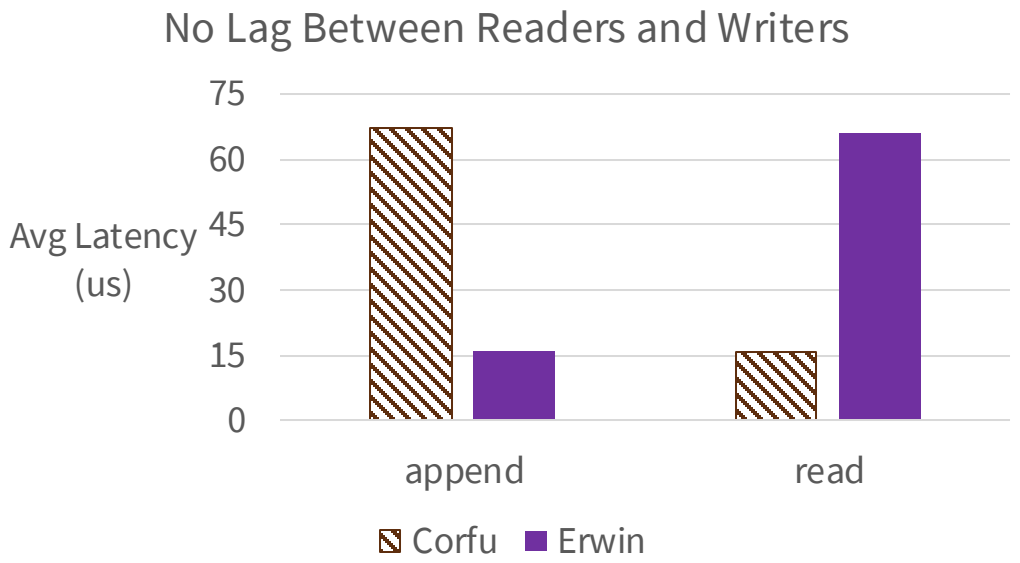
4KB record read after append

For many applications in which reads lag behind writes:

- Erwin achieves low append latency and read latency

In the worst case when there is *no* lag:





How Do Reads Perform in LazyLog?

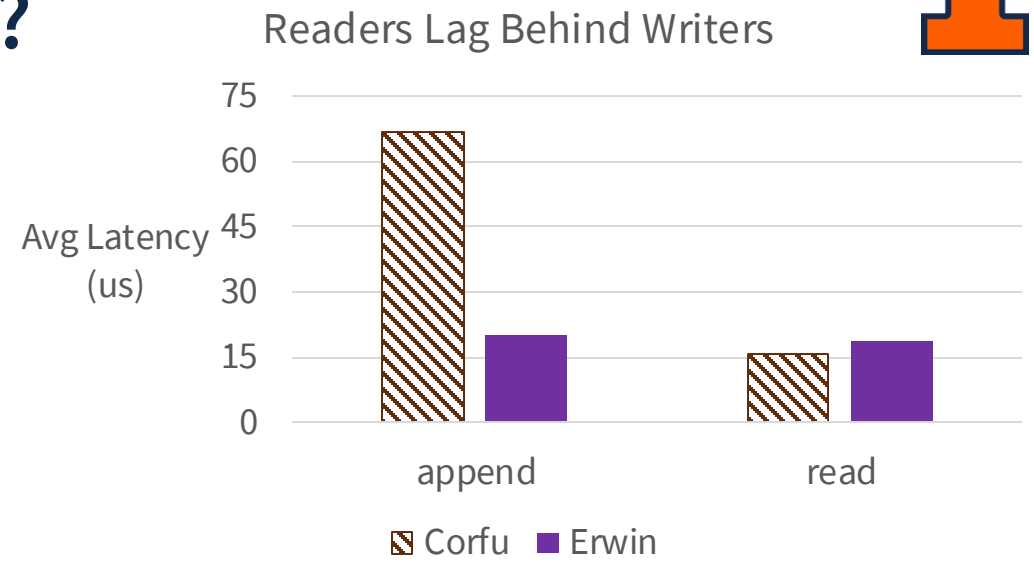
4KB record read after append

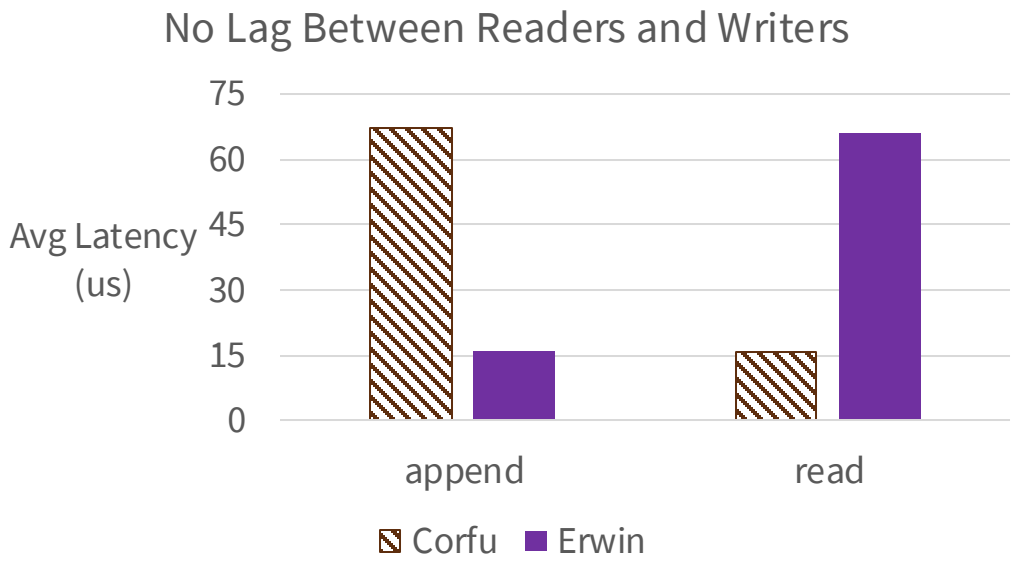
For many applications in which reads lag behind writes:

- Erwin achieves low append latency and read latency

In the worst case when there is *no* lag:

- Erwin shifts ordering cost from append to read





How Do Reads Perform in LazyLog?

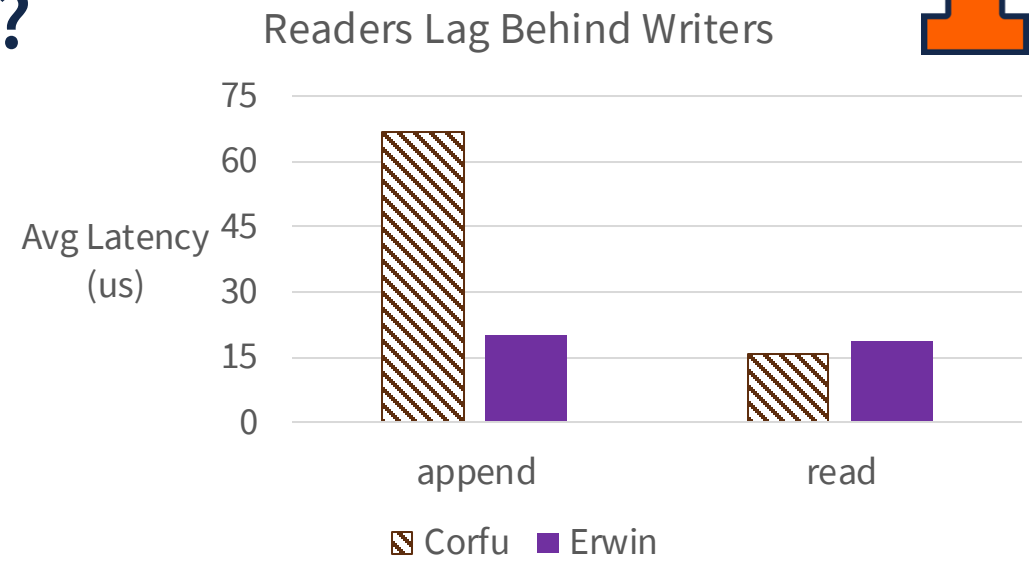
4KB record read after append

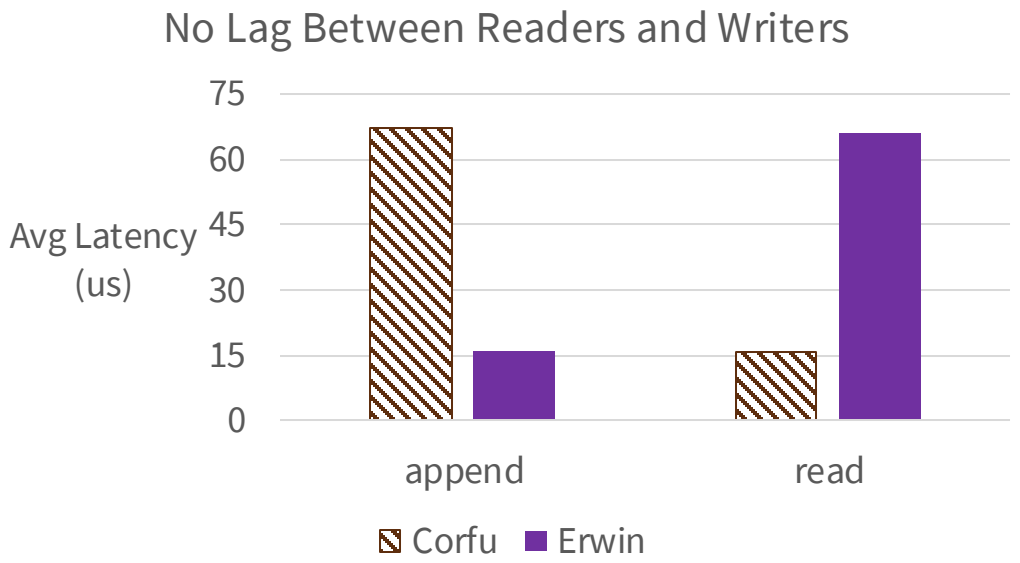
For many applications in which reads lag behind writes:

- Erwin achieves low append latency and read latency

In the worst case when there is *no* lag:

- Erwin shifts ordering cost from append to read
- Append + read latency remains the same





Do End Apps Benefit from LazyLog?

Do End Apps Benefit from LazyLog?

Built 3 Apps: *KV Store*, *Audit Log*, and *Journal for stream processing system*

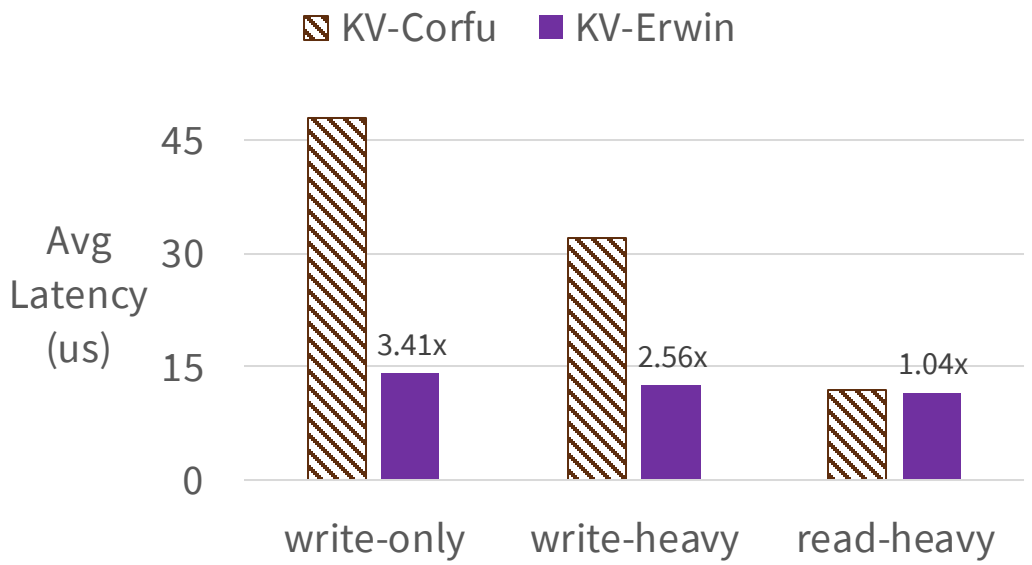
Do End Apps Benefit from LazyLog?

Built 3 Apps: *KV Store*, *Audit Log*, and *Journal for stream processing system*

KV Store (decoupled WR-er and RD-er):

Append to log on PUTs

Reader reads log, constructs state, serves GETs



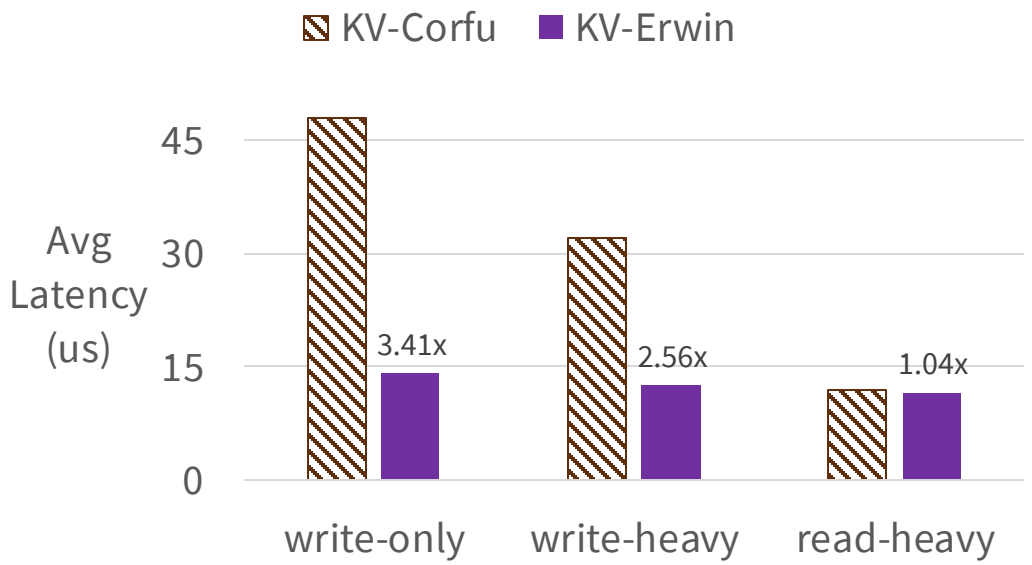
Do End Apps Benefit from LazyLog?

Built 3 Apps: *KV Store*, *Audit Log*, and *Journal for stream processing system*

KV Store (decoupled WR-er and RD-er):

Append to log on PUTs

Reader reads log, constructs state, serves GETs



Do End Apps Benefit from LazyLog?

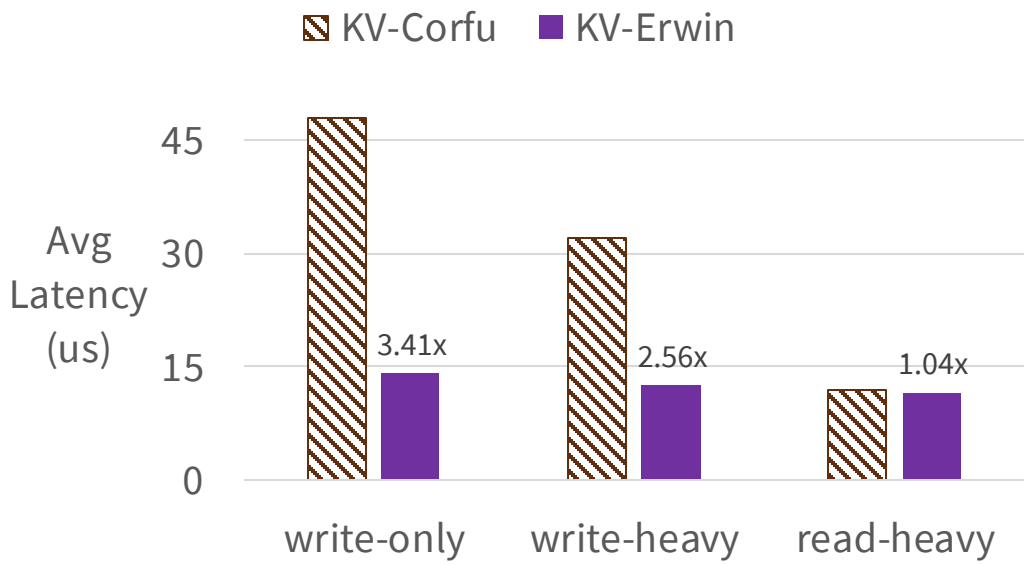
Built 3 Apps: *KV Store*, *Audit Log*, and *Journal for stream processing system*

KV Store (decoupled WR-er and RD-er):

Append to log on PUTs

Reader reads log, constructs state, serves GETs

Erwin benefits applications by reducing ingestion latency



Do End Apps Benefit from LazyLog?

Built 3 Apps: *KV Store*, *Audit Log*, and *Journal for stream processing system*

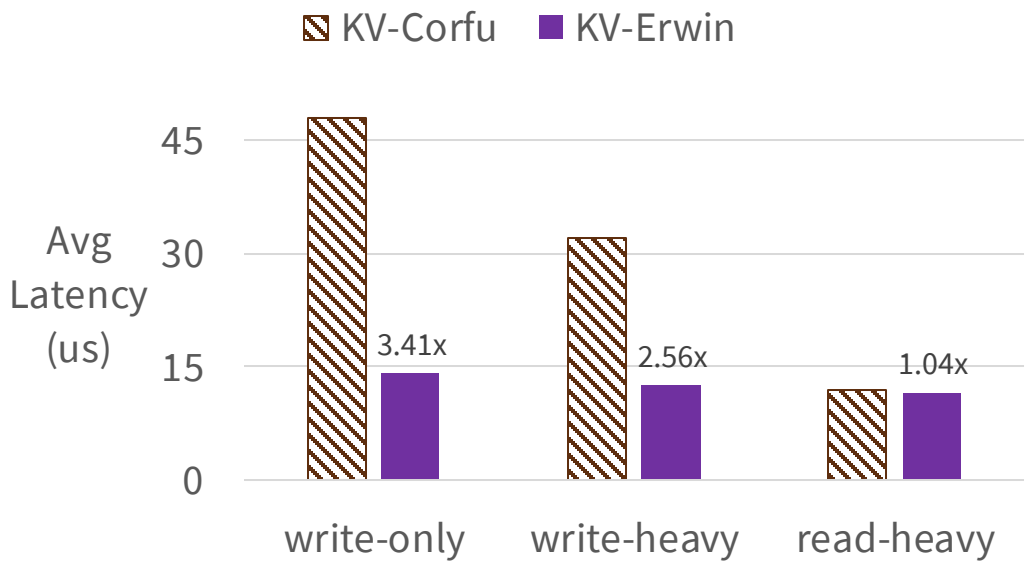
KV Store (decoupled WR-er and RD-er):

Append to log on PUTs

Reader reads log, constructs state, serves GETs

Erwin benefits applications by reducing ingestion latency

- Benefit is more pronounced when shared-log interaction takes significant partition of app request execution



More in the Paper

- More experiments in the paper
- Another implementation:
  - Erwin-bb (black-box): Treat shards as black boxes. Can work with any PB/Raft shard or even Kafka.

See our paper for more details

Summary

Summary

- Eager-ordering shared logs incur high latencies, impacts app performance

Summary

- Eager-ordering shared logs incur high latencies, impacts app performance
- Eager ordering is not needed for many applications and readers are time-decoupled from writers

Summary

- Eager-ordering shared logs incur high latencies, impacts app performance
- Eager ordering is not needed for many applications and readers are time-decoupled from writers
- LazyLog – a new shared-log abstraction that defers ordering

Summary

- Eager-ordering shared logs incur high latencies, impacts app performance
- Eager ordering is not needed for many applications and readers are time-decoupled from writers
- LazyLog – a new shared-log abstraction that defers ordering
- Low ingestion latency with little overhead upon reads

Summary

- Eager-ordering shared logs incur high latencies, impacts app performance
- Eager ordering is not needed for many applications and readers are time-decoupled from writers
- LazyLog – a new shared-log abstraction that defers ordering
- Low ingestion latency with little overhead upon reads
- LazyLog systems deliver benefits for applications

Summary

- Eager-ordering shared logs incur high latencies, impacts app performance
- Eager ordering is not needed for many applications and readers are time-decoupled from writers
- LazyLog – a new shared-log abstraction that defers ordering
- Low ingestion latency with little overhead upon reads
- LazyLog systems deliver benefits for applications

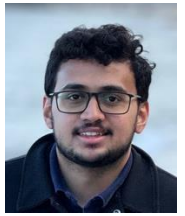
Available on GitHub:
https://github.com/dassl-uiuc/LazyLog-Artifact

Summary

- Eager-ordering shared logs incur high latencies, impacts app performance
- Eager ordering is not needed for many applications and readers are time-decoupled from writers
- LazyLog – a new shared-log abstraction that defers ordering
- Low ingestion latency with little overhead upon reads
- LazyLog systems deliver benefits for applications

Available on GitHub:
https://github.com/dassl-uiuc/LazyLog-Artifact


Xuhao Luo


Shreesha G. Bhat*


Jiyu Hu*


Ram Alagappan


Aishwarya Ganesan