LazyLog: A Shared Log Abstraction for Low-Latency Applications

Xuhao Luo, Shreesha G. Bhat, Jiyu Hu, Ram Alagappan, Aishwarya Ganesan

University of Illinois Urbana-Champaign

ANKITA SINGH
2024MCS2891

ABHIYA M JOSE
2024JCS2046

VADDI JHANCY
2024JCS2040

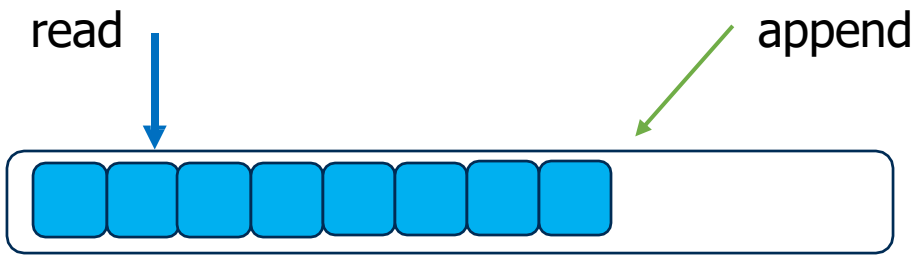
Shared Log: Abstraction and Interface

Abstraction:

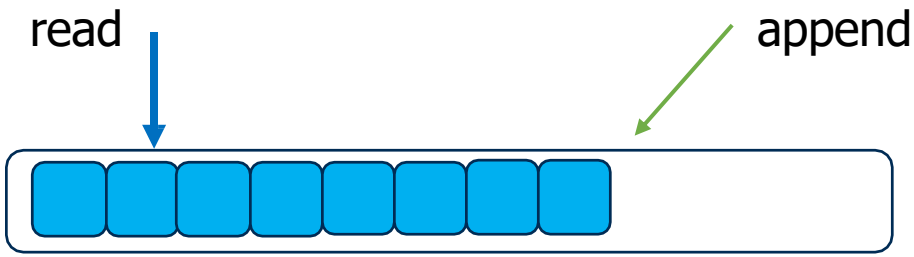
Fault-tolerant, durable,
linearizably ordered sequence of records

Interface:

append
read



Shared Log: Abstraction and Interface




```
// 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);
```

The Problem with Current Shared Logs

- high append latencies
  - Append takes multiple RTTs
- Low append latency is critical to applications
- Eager ordering nature of shared logs:

The Problem with Current Shared Logs

- high ingestion latencies
  - Append takes multiple RTTs

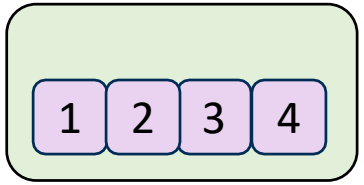
- Low ingestion latency is critical to applications

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

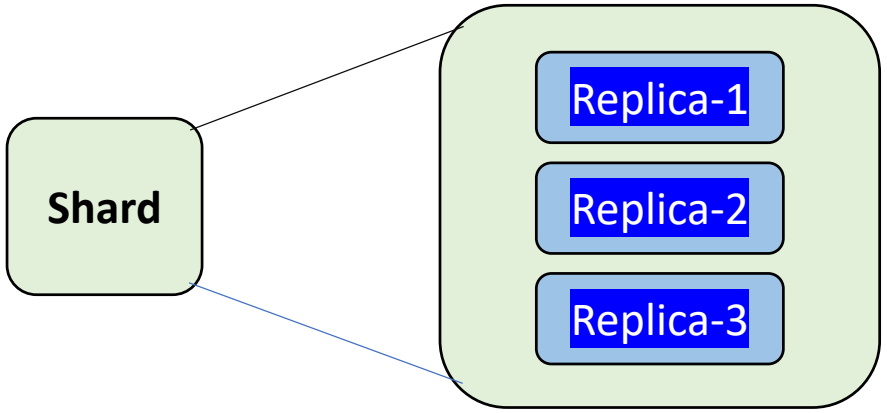
Total Ordering in Shared Logs

Linearizable order:

if append(B) starts after append(A) completes, then B appears after A in the shared log



Shard-1

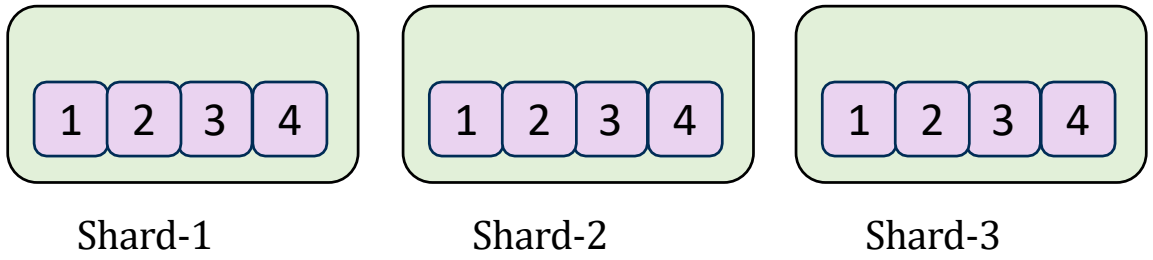


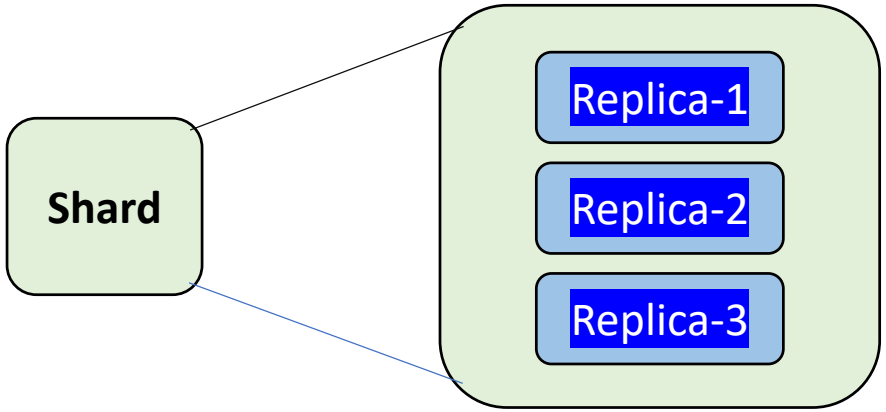
Fault Tolerance via Replication

Total Ordering in Shared Logs

Linearizable order:

if append(B) starts after append(A) completes, then B appears after A in the shared log





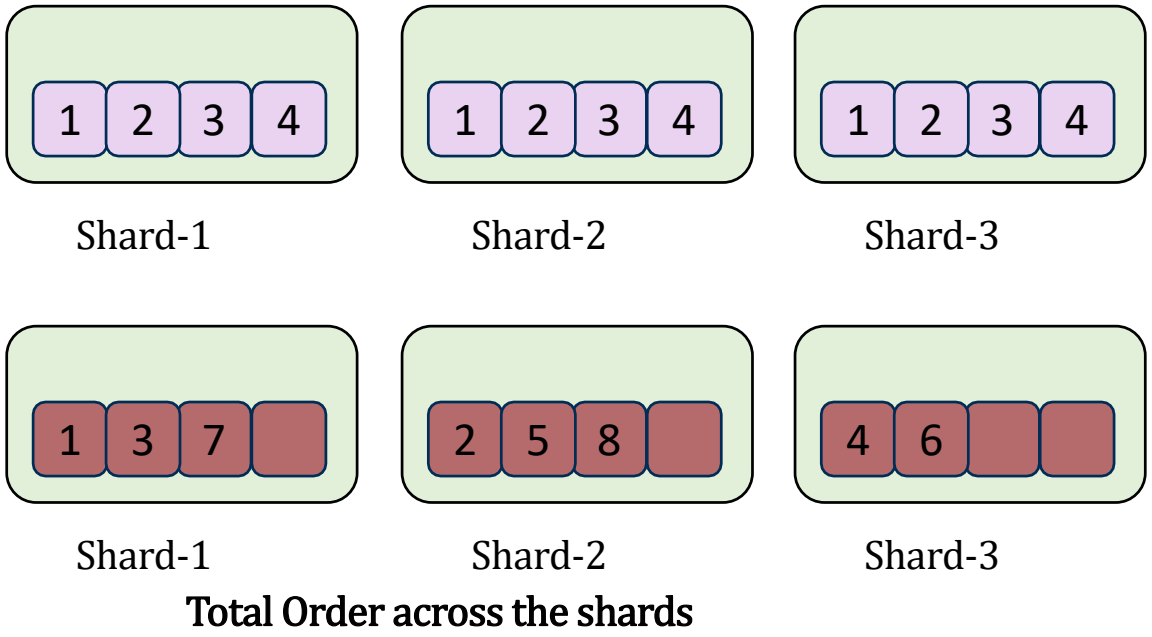
Fault Tolerance via Replication

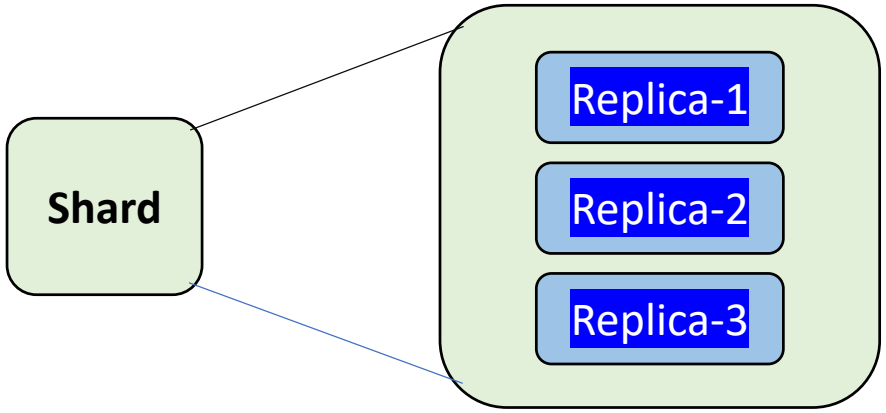
Requires Total Ordering across the shards

Total Ordering in Shared Logs

Linearizable order:

if append(B) starts after append(A) completes, then B appears after A in the shared log

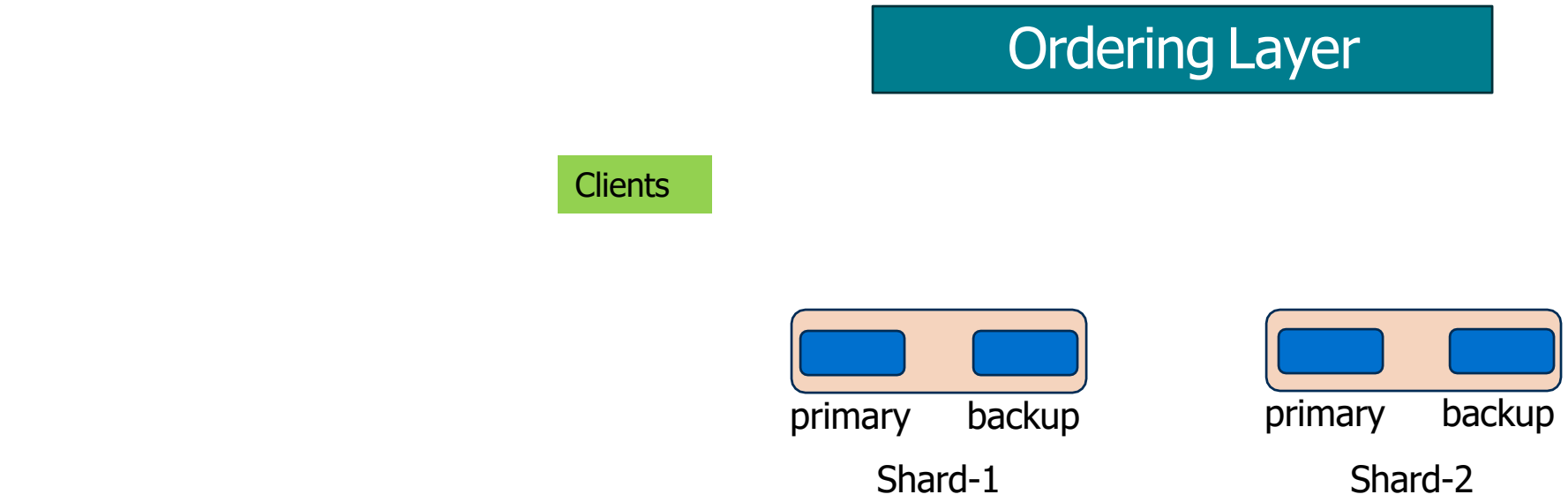




Fault Tolerance via Replication

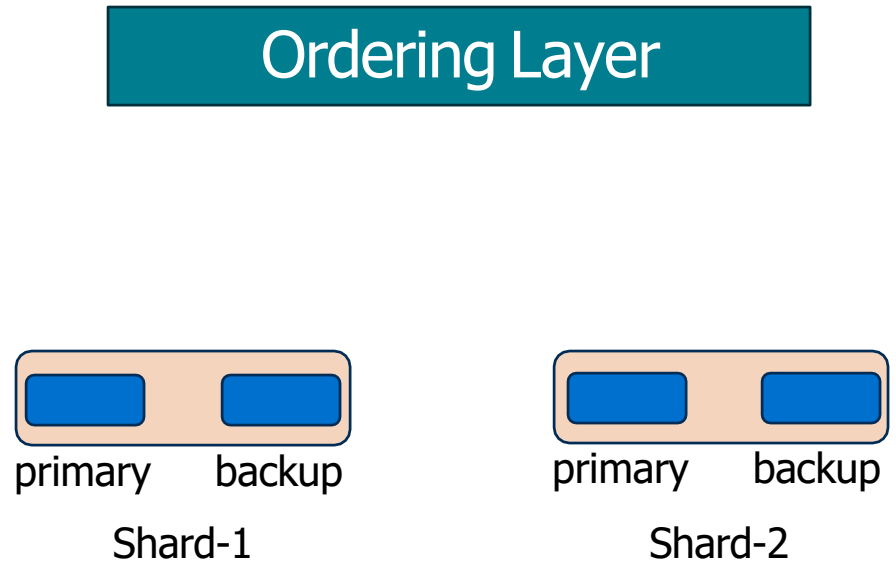
Scalability with total order: Scalog & Corfu

Eager Ordering → High Latency



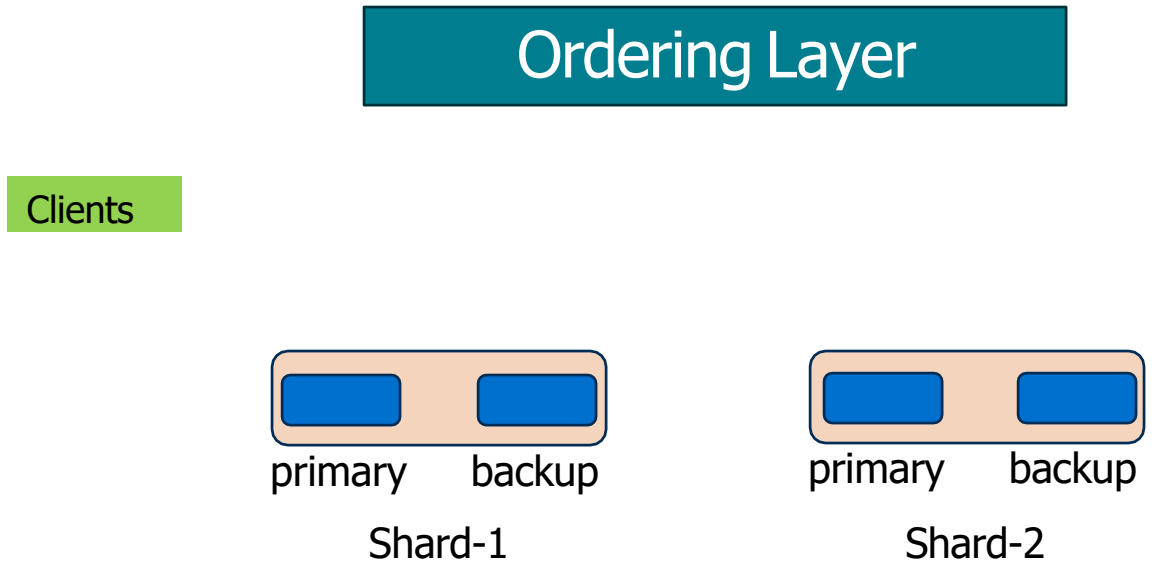
Eager Ordering → High Latency

- Incur high append latency



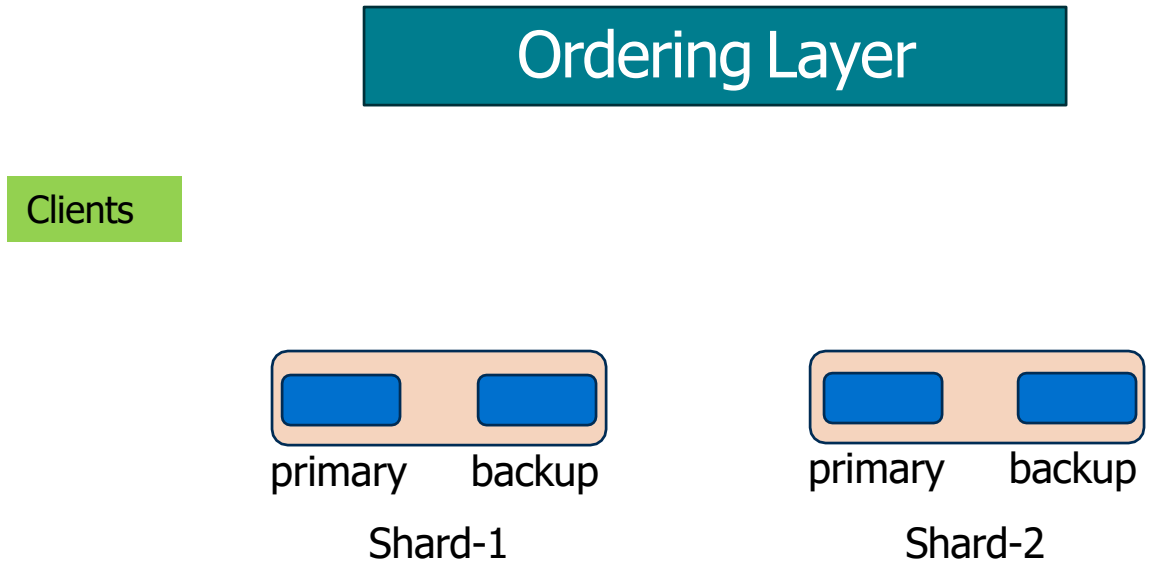
Eager Ordering → High Latency

- Incur high append latency
- Rooted in eager ordering



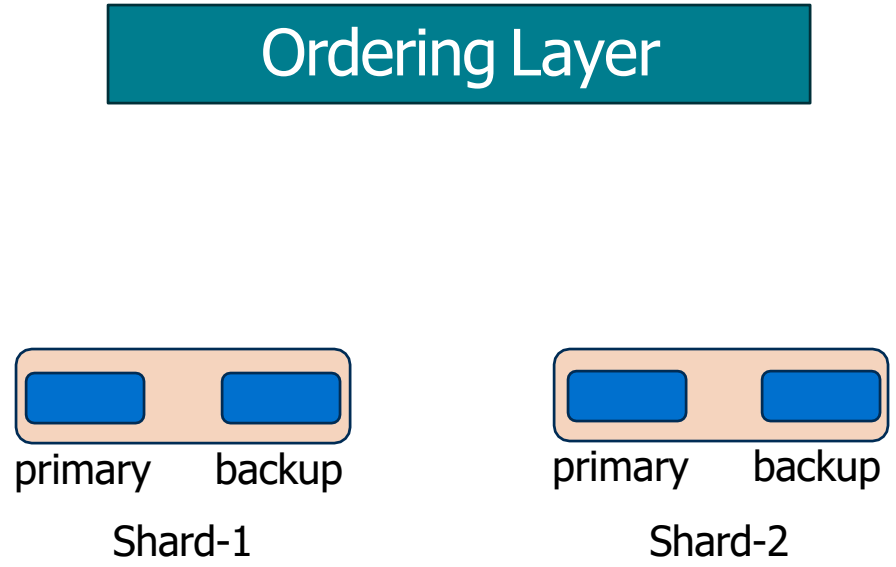
Eager Ordering → High Latency

- Incur high append latency
- Rooted in eager ordering
- Both durability and global ordering are completed before getting back to clients



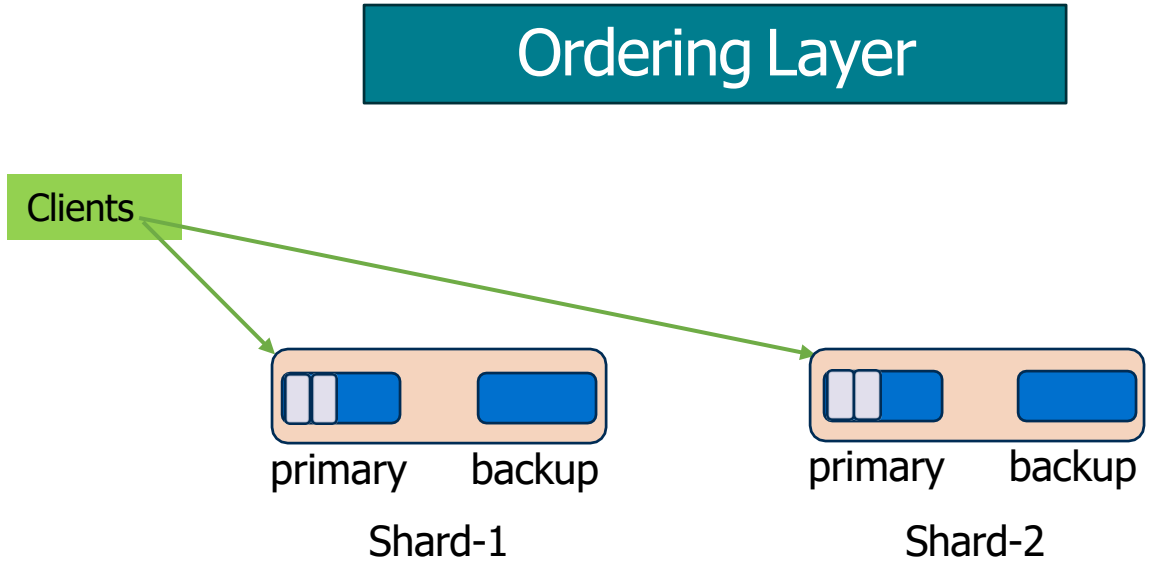
Eager Ordering → High Latency

- **Scalog**



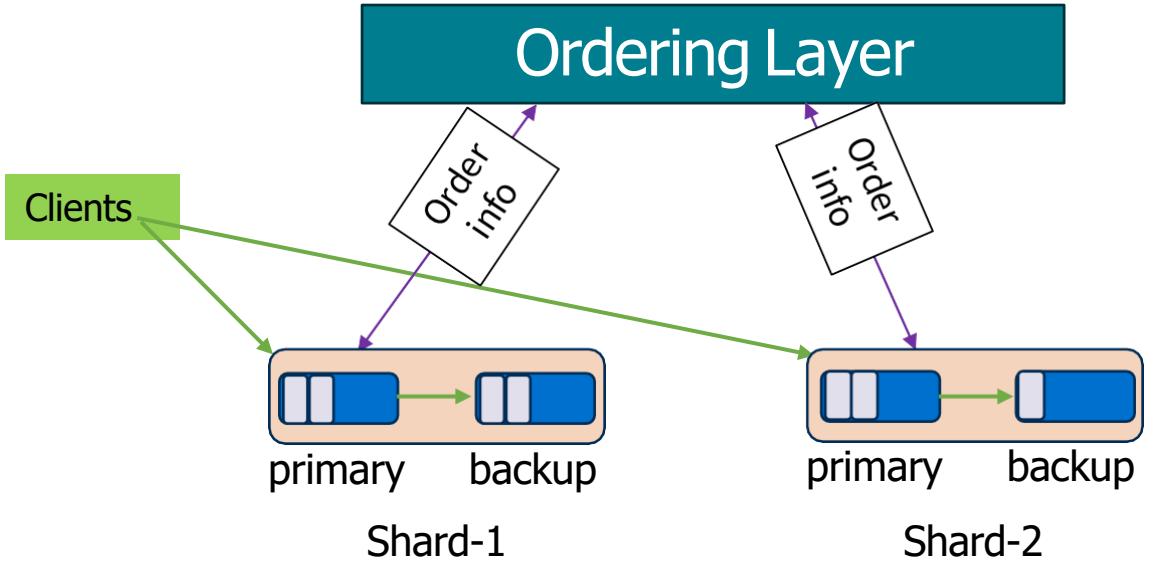
Eager Ordering → High Latency

- Scalog
  - durability first



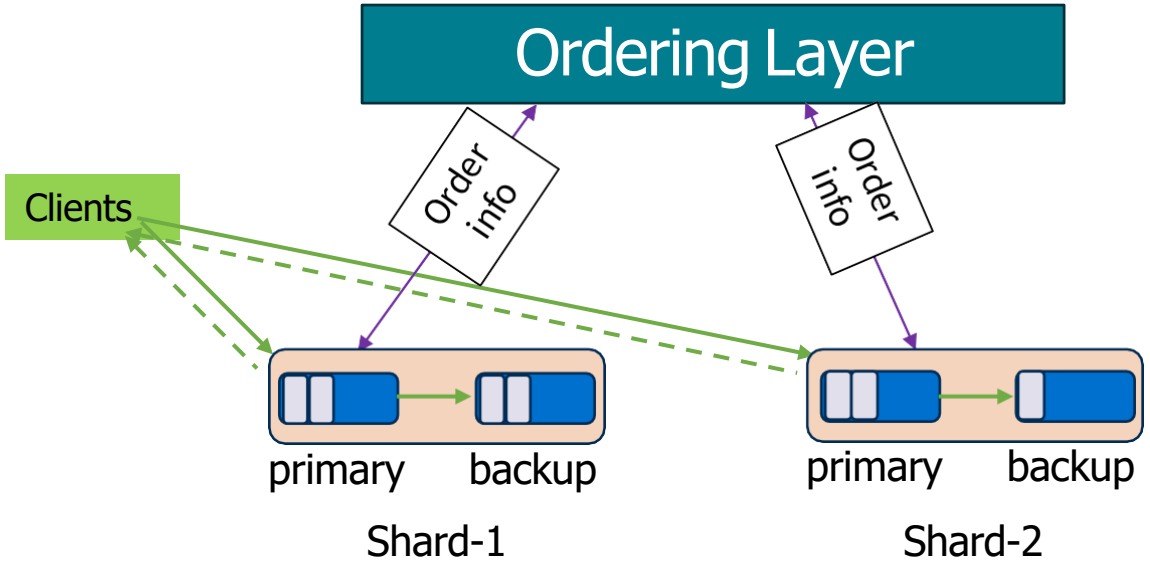
Eager Ordering → High Latency

- **Scalog**
  - durability first
  - then global ordering



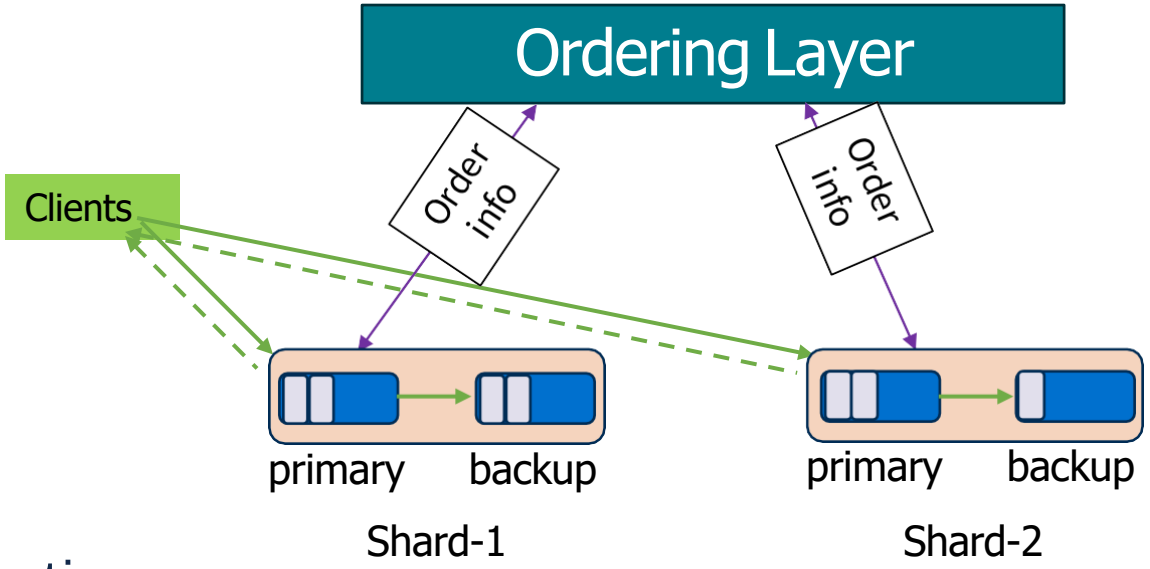
Eager Ordering → High Latency

- **Scalog**
  - durability first
  - then global ordering



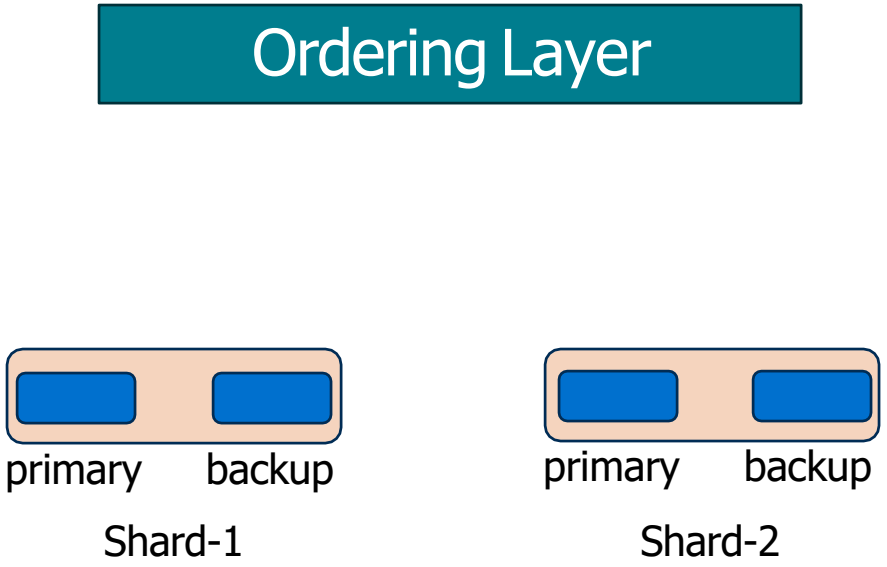
Eager Ordering → High Latency

- **Scalog**
  - durability first
  - then global ordering
  - multiple RTT +batch interval
  - Results in high ingestion latency for applicati ons



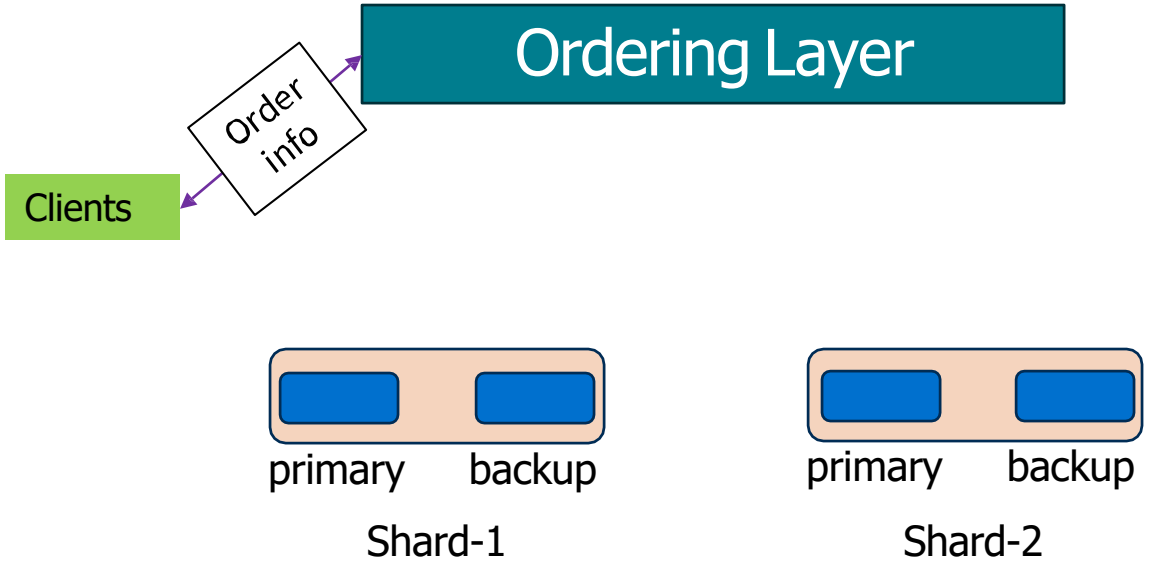
Eager Ordering → High Latency

- **Corfu**



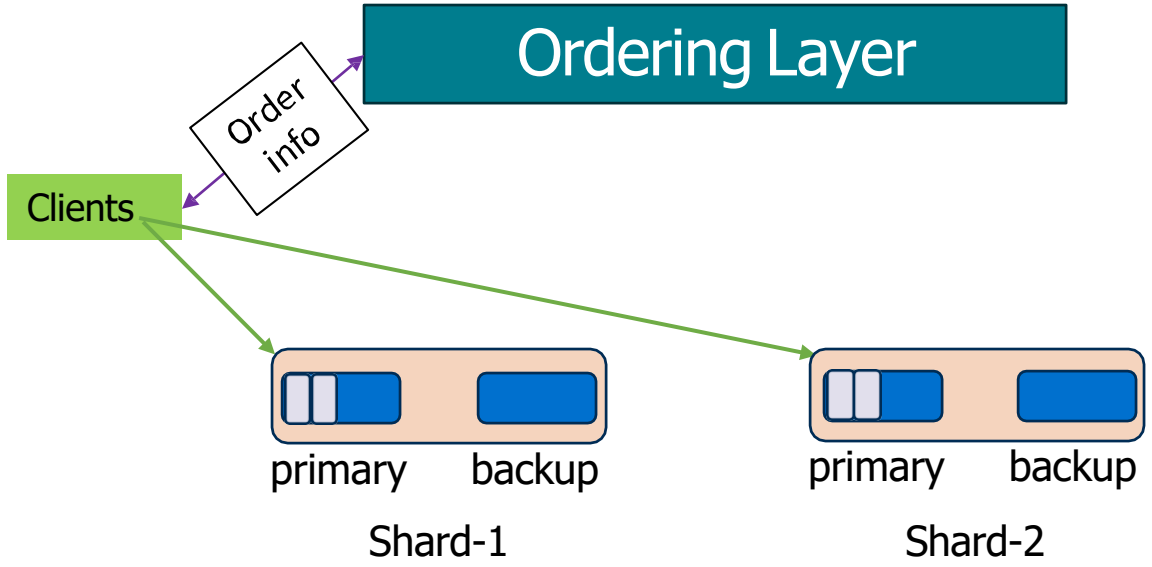
Eager Ordering → High Latency

- **Corfu**
  - global ordering first



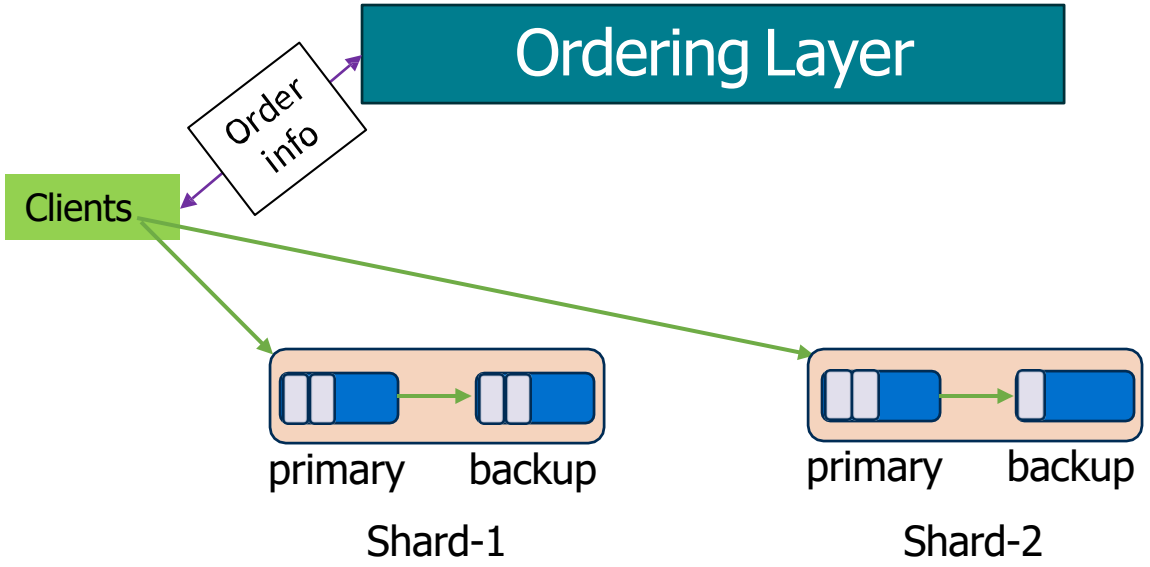
Eager Ordering → High Latency

- **Corfu**
  - global ordering first
  - then durability



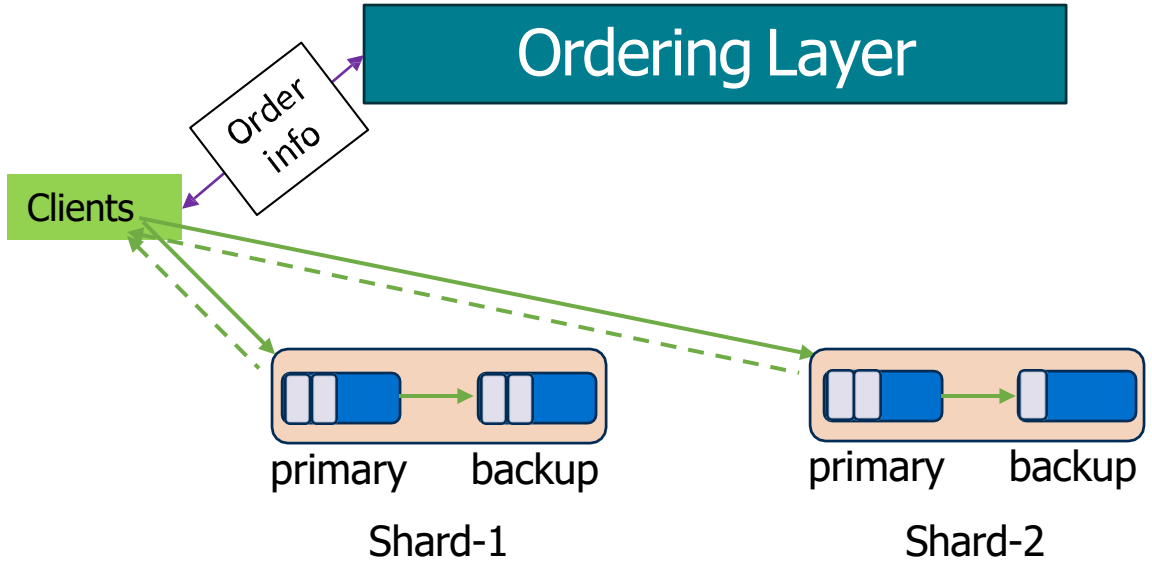
Eager Ordering → High Latency

- Corfu
  - global ordering first
  - then durability



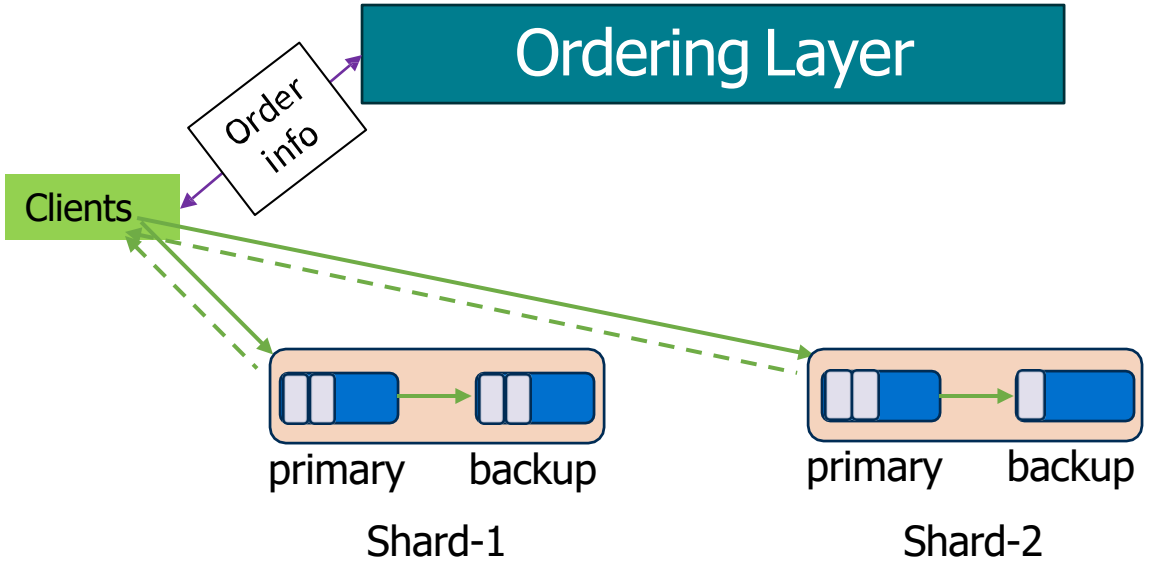
Eager Ordering → High Latency

- **Corfu**
  - global ordering first
  - then durability



Eager Ordering → High Latency

- **Corfu**
  - global ordering first
  - then durability
  - multiple RTT
  - Results in high ingestion latency for applications



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

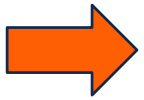
LazyLog: Idea and Abstraction

Idea

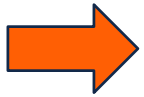
- No need for eagerly bind upon an append
  - But only make it durable

LazyLog: Idea and Abstraction

Idea



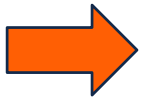
- No need for eagerly bind upon an append
  - But only make it durable



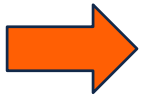
- Bind the records lazily, it must enforce ordering before positions can be read

LazyLog: Idea and Abstraction

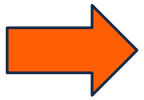
Idea



- No need for eagerly bind upon an append
  - But only make it durable



- Bind the records lazily, it must enforce ordering before positions can be read

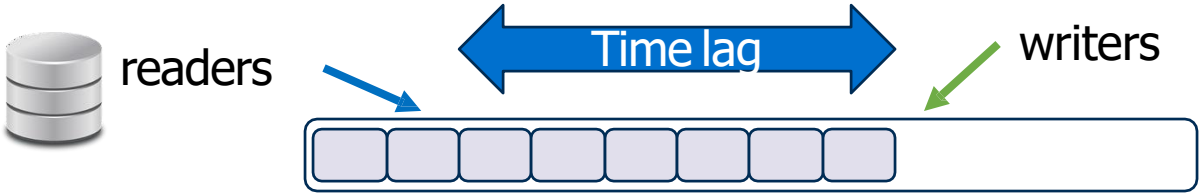


- Shared log can do the ordering comfortably in the background

Insight

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

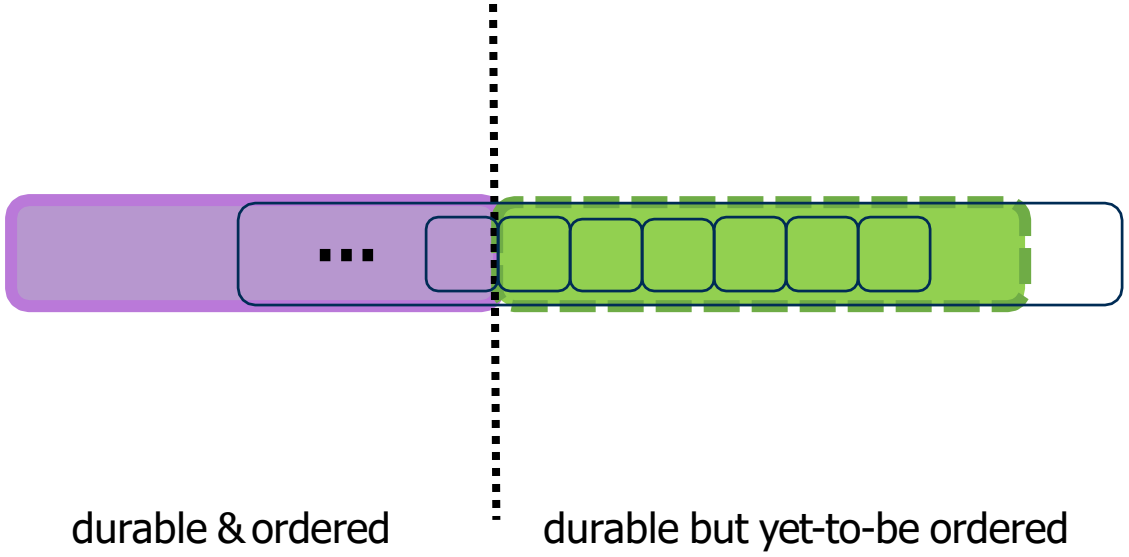
Holds for Many Apps



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

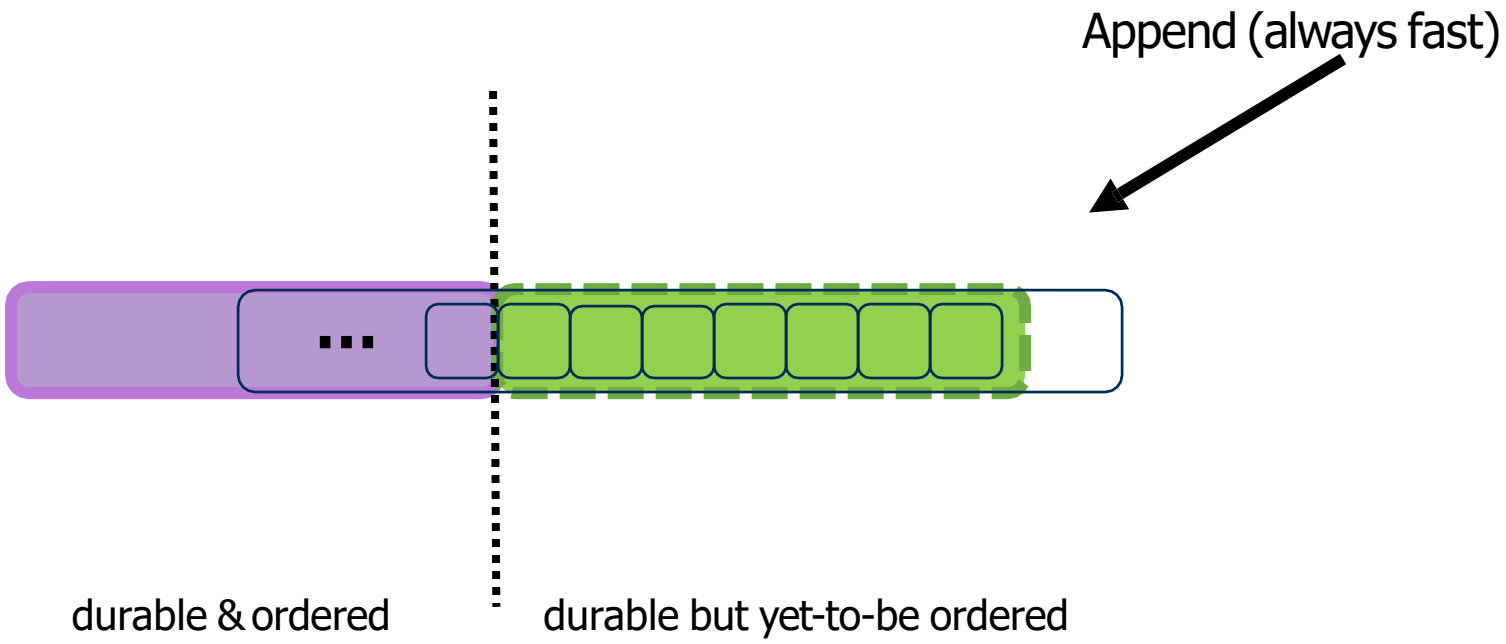
Performance Property

- Cannot be too lazy – keep ordering in the background



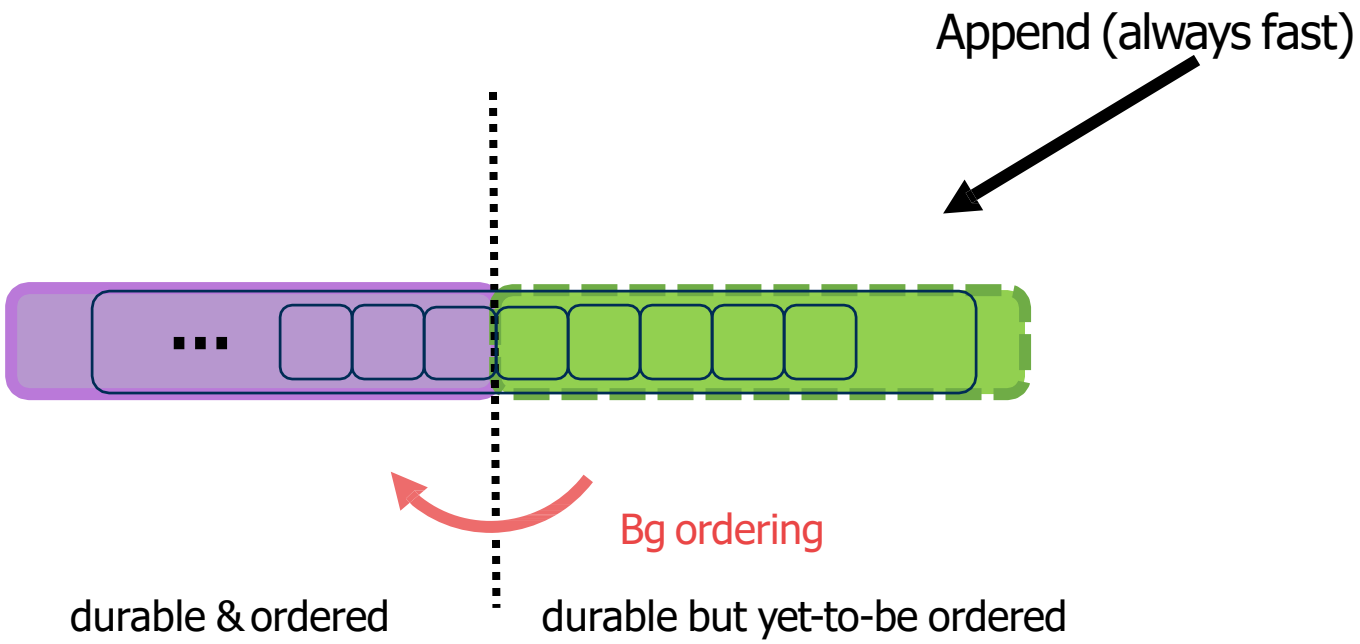
Performance Property

- Cannot be too lazy – keep ordering in the background



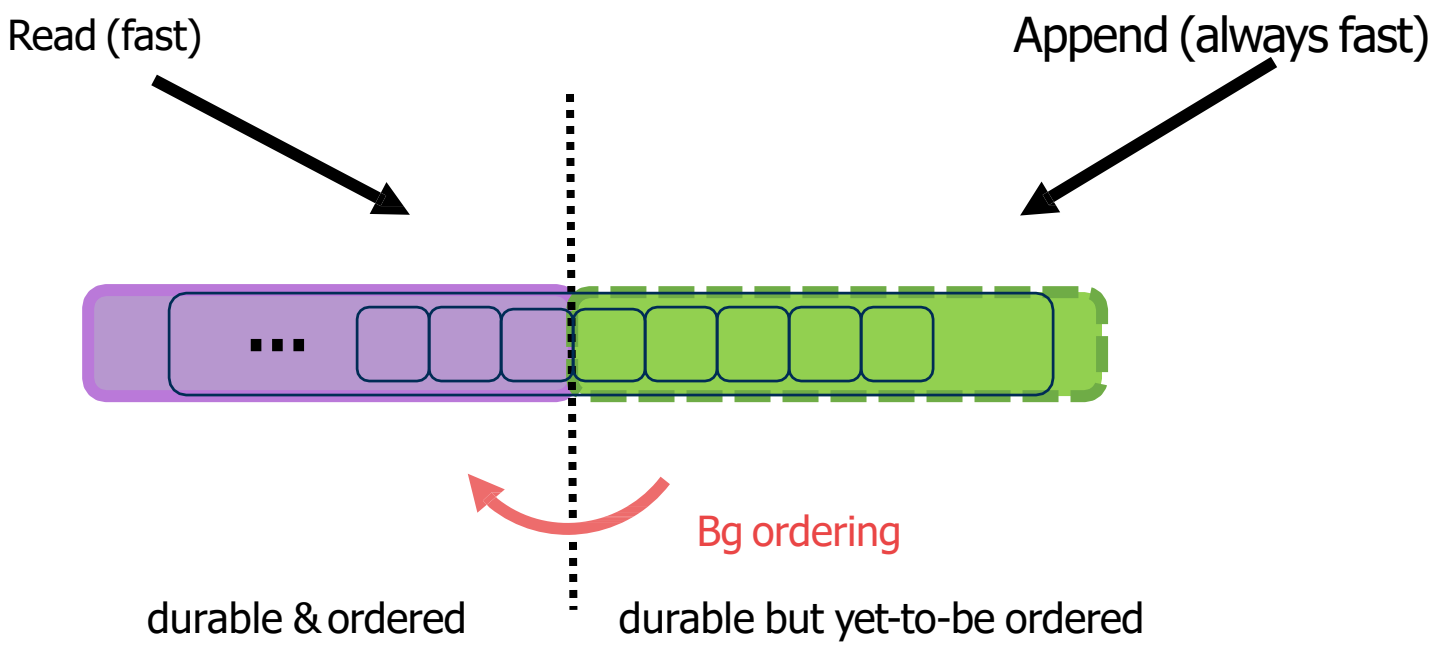
Performance Property

- Cannot be too lazy – keep ordering in the background



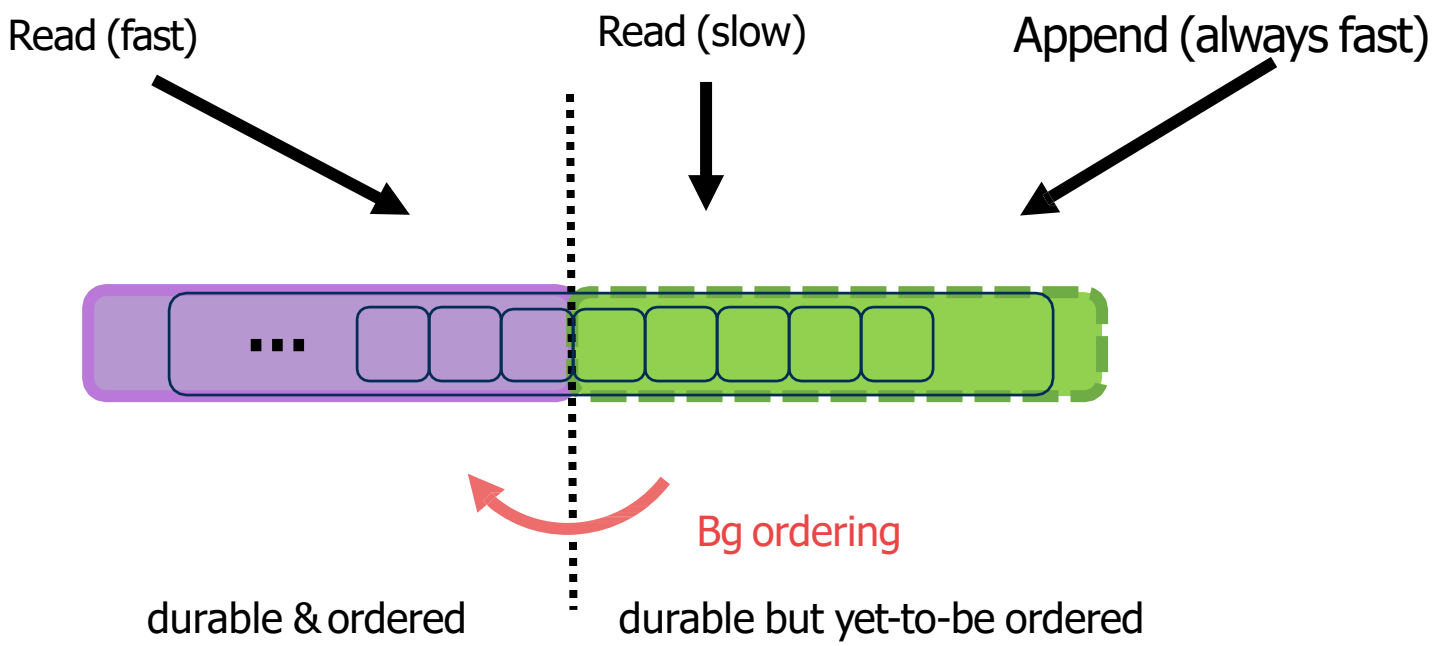
Performance Property

- Cannot be too lazy – keep ordering in the background



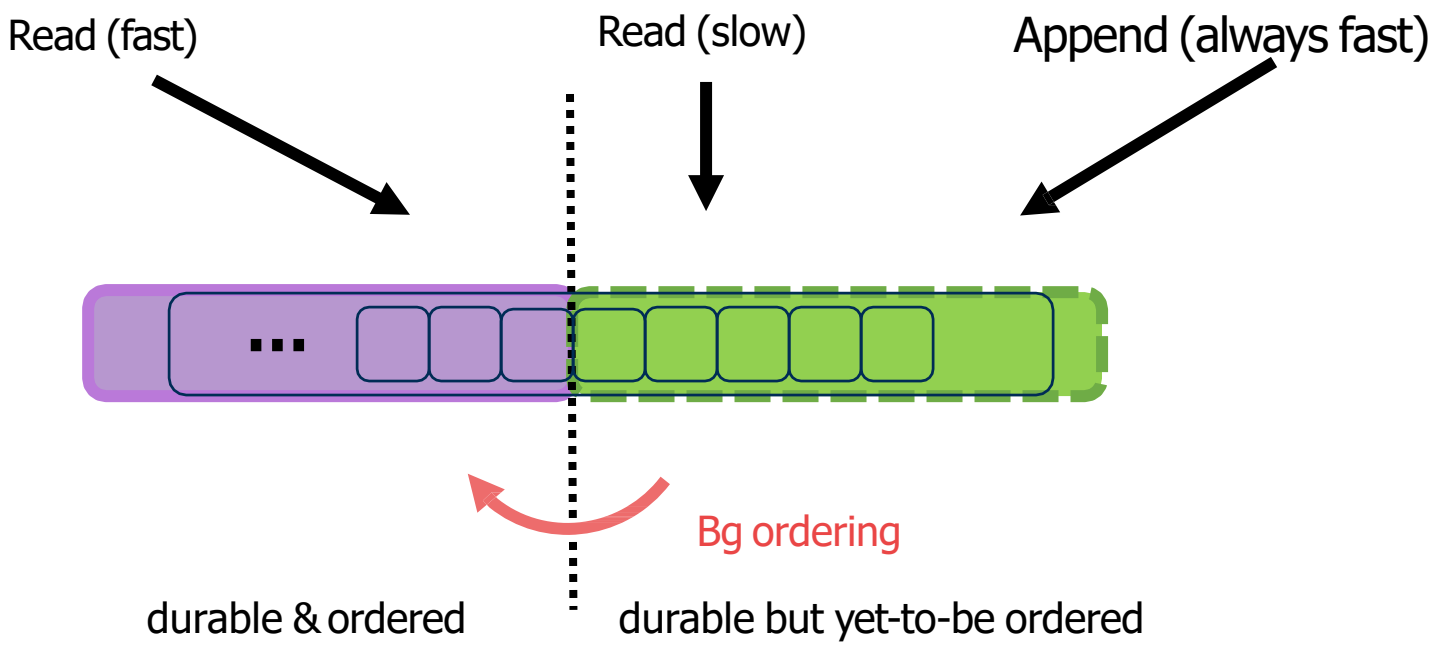
Performance Property

- Cannot be too lazy – keep ordering in the background



Performance Property

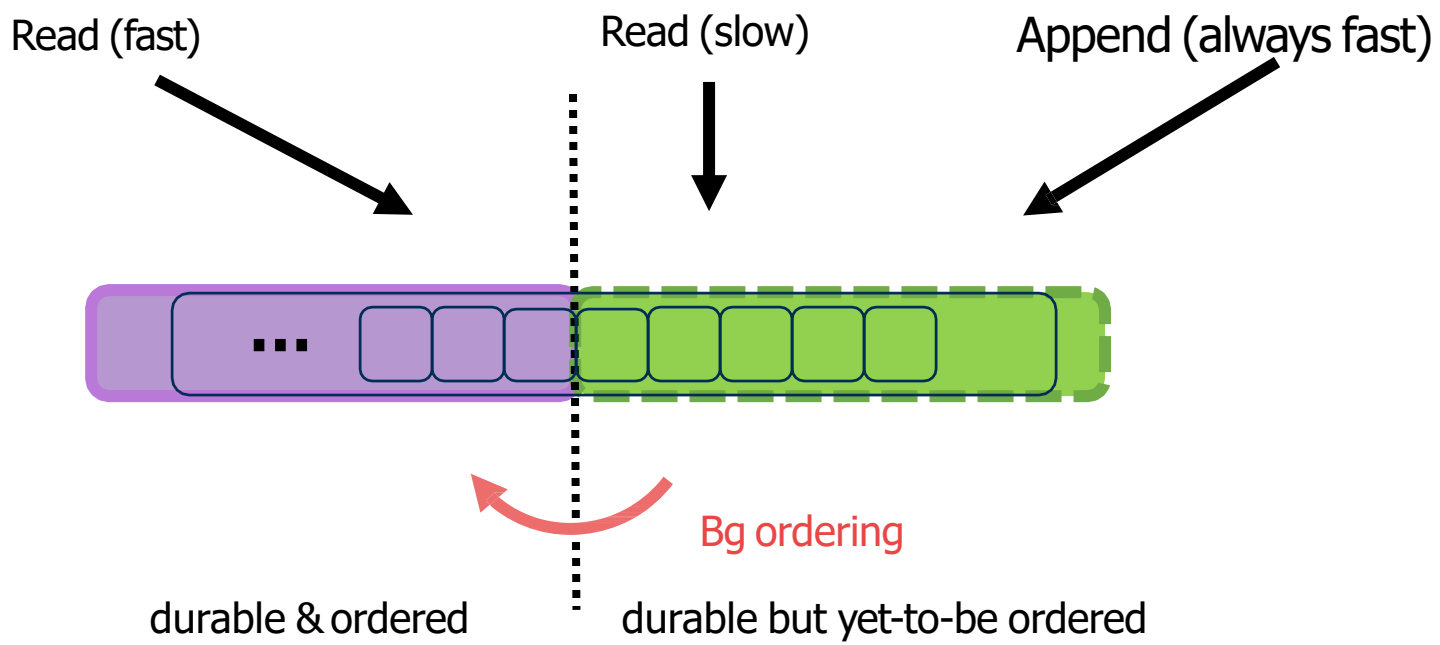
- Cannot be too lazy – keep ordering in the background



- For many apps – reads are always fast

Performance Property

- Cannot be too lazy – keep ordering in the background



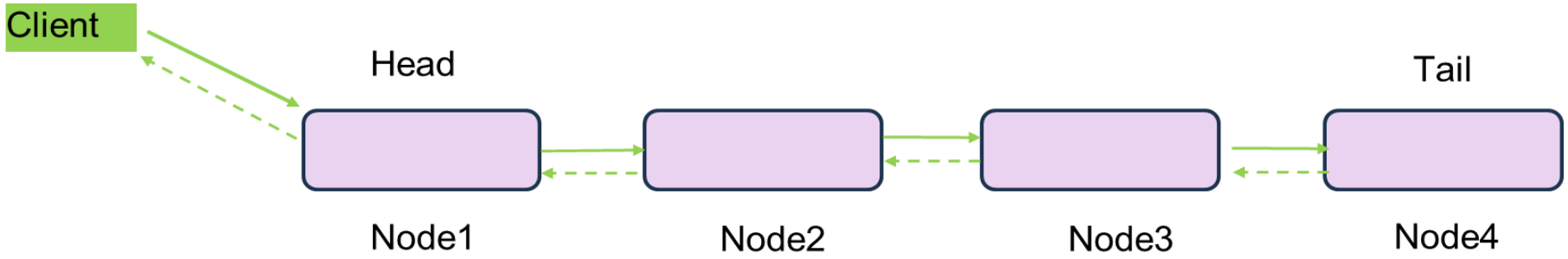
- For many apps – reads are always fast
- For immediate read, LazyLog preserves the performance of eager shared logs
  - never worse than an eager-ordering shared log!

Insight

Why 1-RTT Appends are hard to achieve in Systems like Chain Paxos or Raft?

Chain Paxos

Requires multi-step coordination between replicas:



Client → Node1

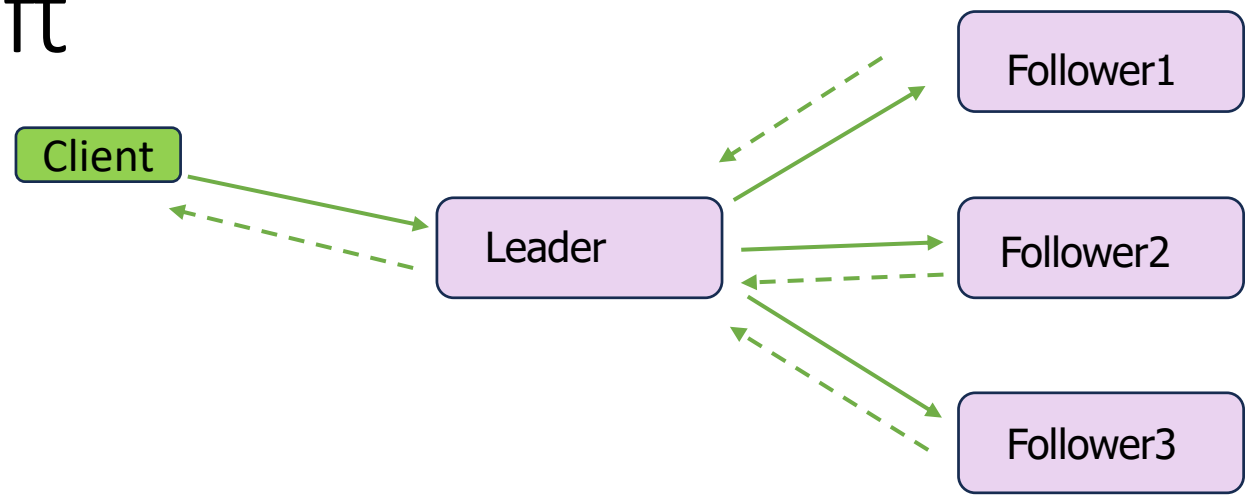
Node1 → Node2

Node2 → Node3

Node3 → Node4

Node4 → Client

Raft



Client → Leader (Send Request)

Leader → Followers (Replicate Entry)

Followers → Leader (Acks)

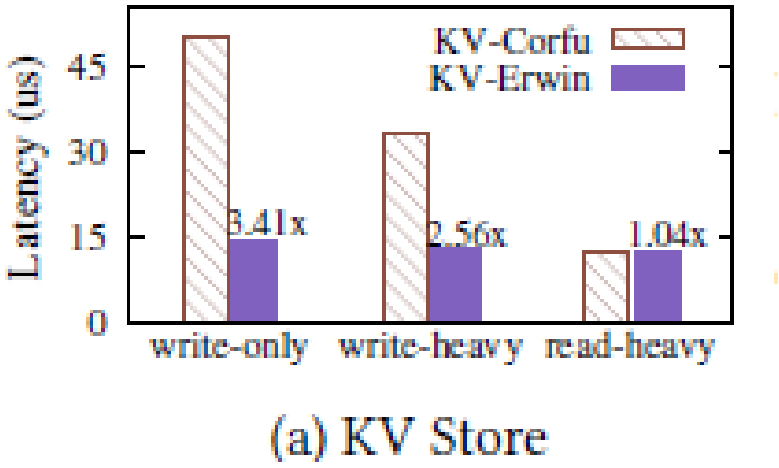
Leader → Client (Reply)

Insight

Application: Lazy-Log

Application: Key-Value Store

- Shared-log based key-value store
- Readers – Writers are decoupled
- Write-processing server- handles append request
- Read server – consume the log



(a) KV Store

- **One writer server, one reader server**.
- Shared log with **1 shard (3 replicas)**.
- Compared **Corfu** vs **Erwin** (LazyLog)

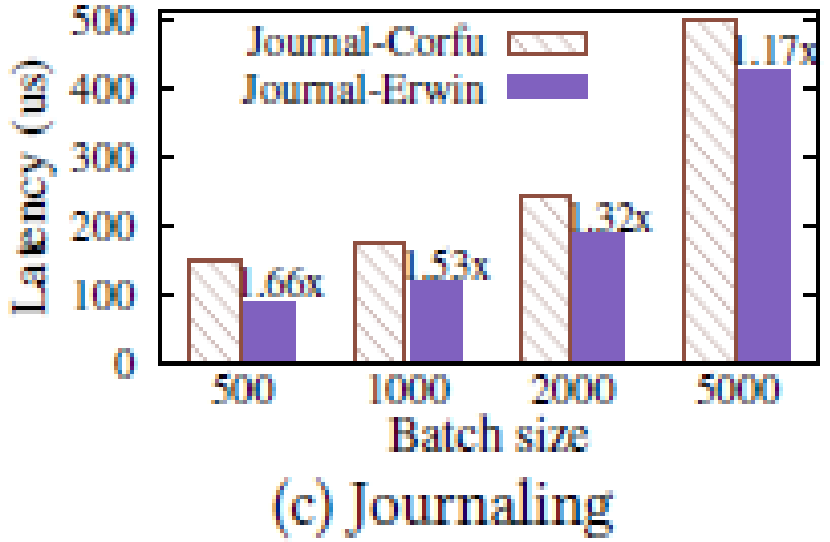
3 Tests

• YCSB	100% writes	write only	3.4× lower latency
• YCSB-A	50% writes & 50% reads	write heavy	2.5× lower latency
• YCSB-B	5% writes & 95% reads	read heavy	No change

Application: Journaling for Stream Processing

- Task Worker-
  reads the incoming records
  use shared log to store their states
  output the result
  processes in batch

To recover during failure



(c) Journaling

Latency = Record Reading + Processing + checkpointing + emitting
Compared **Corfu** vs **Erwin**

Tests: stream-processing word count app with 5 workers For Different batches

- 5,000 records more computation **1.17×** faster
- 500 records logging dominates **1.66×** faster

System Design

Designed an implementation of the LazyLog interface: Erwin

Goal- 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

Treats shards as black boxes and requires them to support:

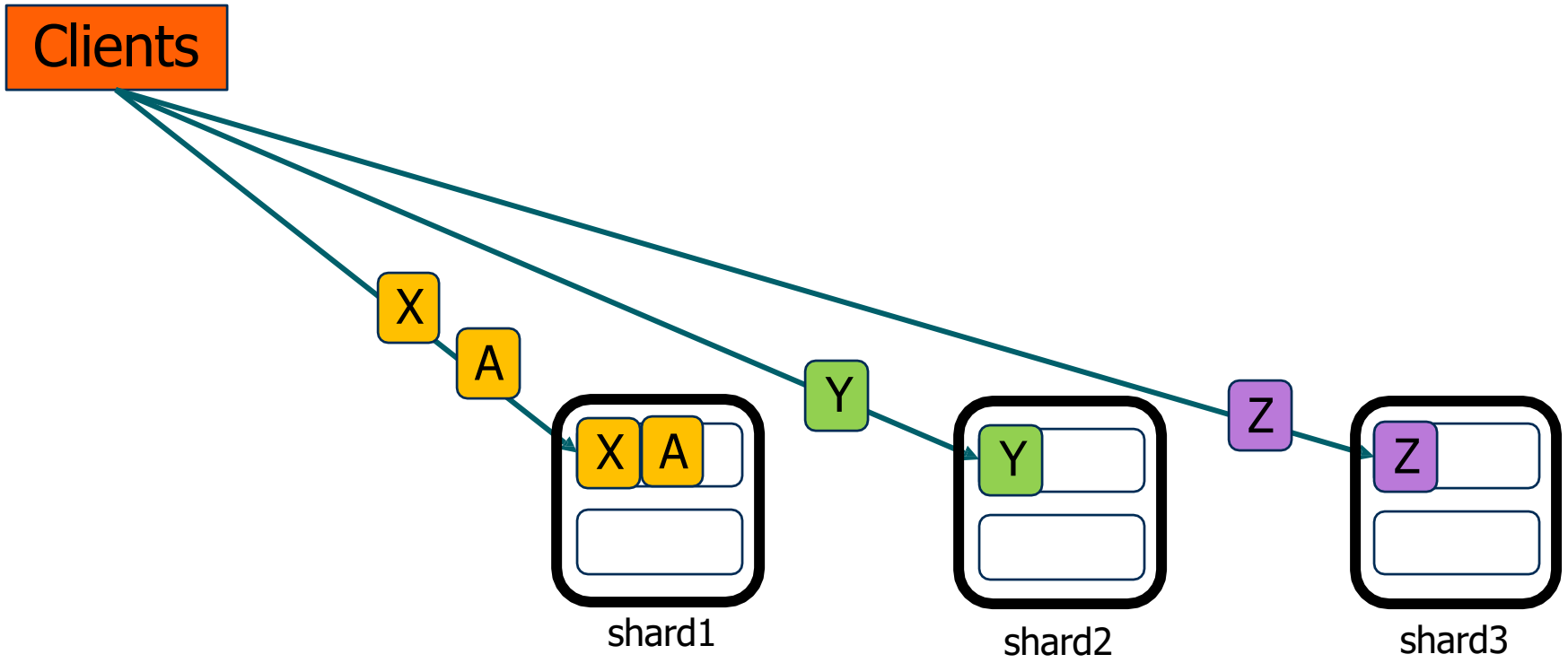
- append an entry and
- read the entry at a specified index / position

Why Erwin?

Motivation

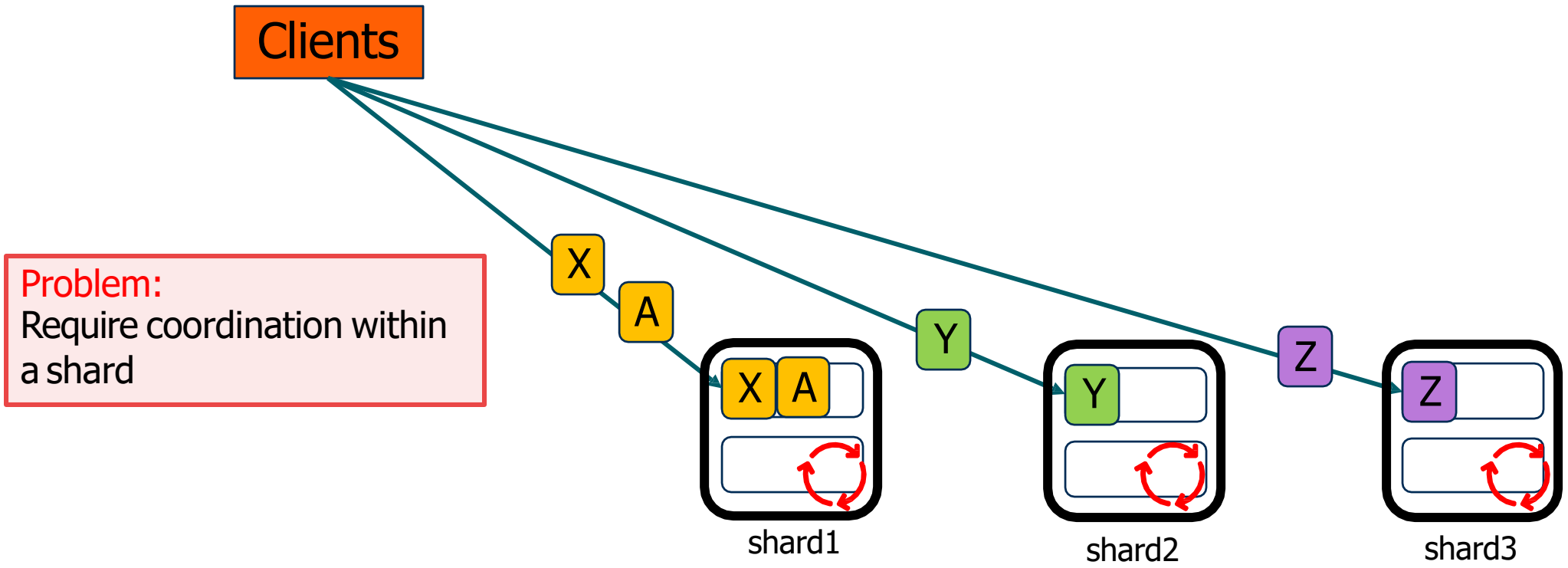
Motivation

Record



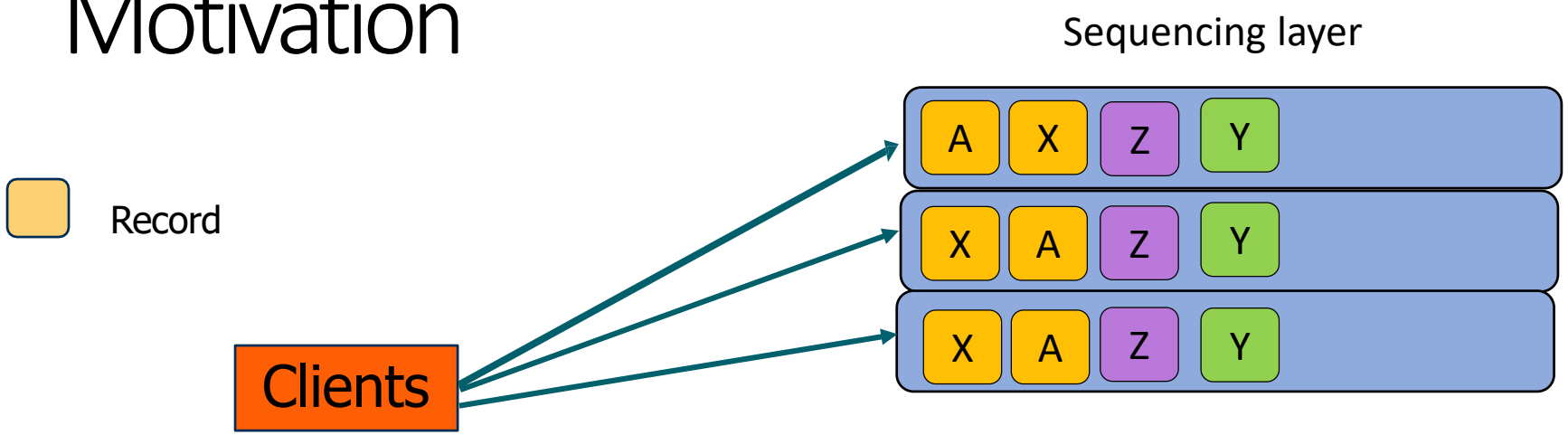
Motivation

Record



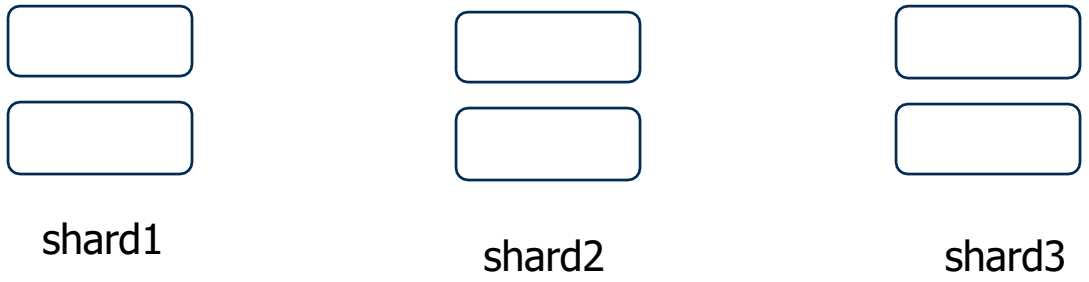
Problem:
Require coordination within a shard

Motivation

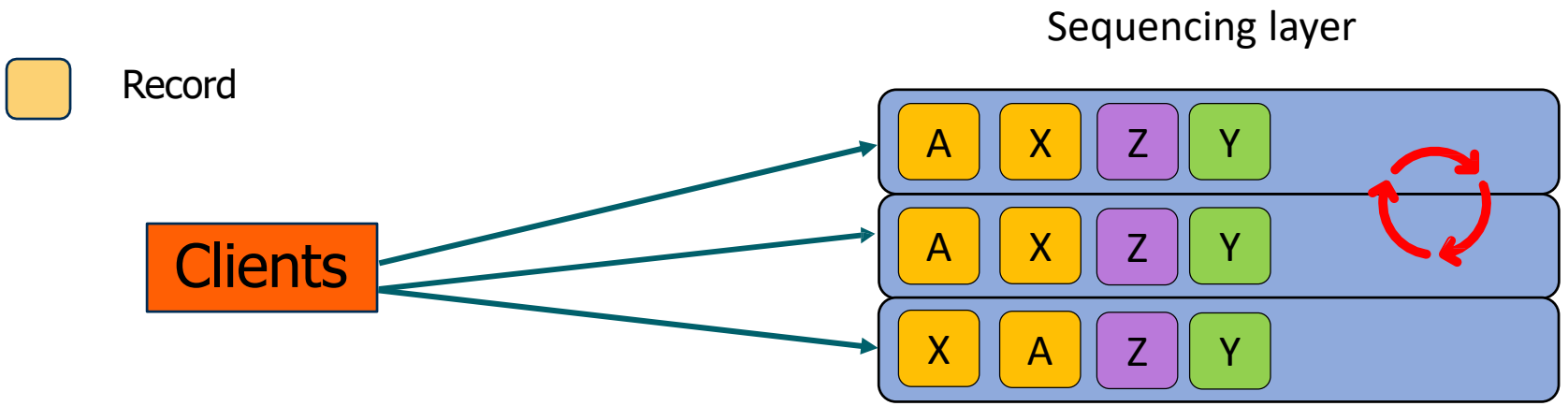


Problem:
Require coordination within a shard

Solution:
Use a buffer and send requests to this buffer

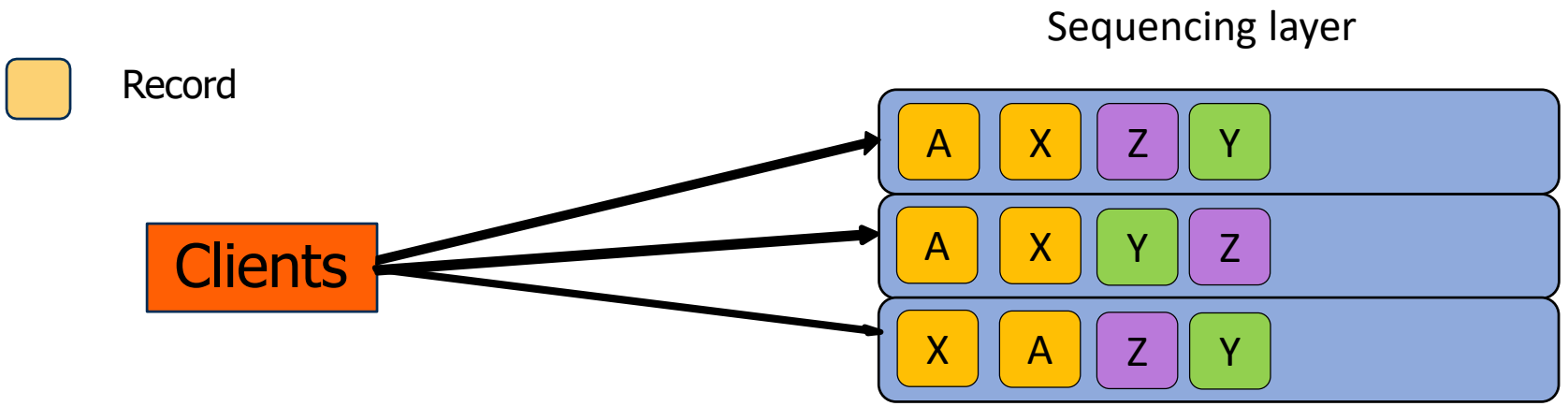


Erwin: 1-RTT Append



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

- 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

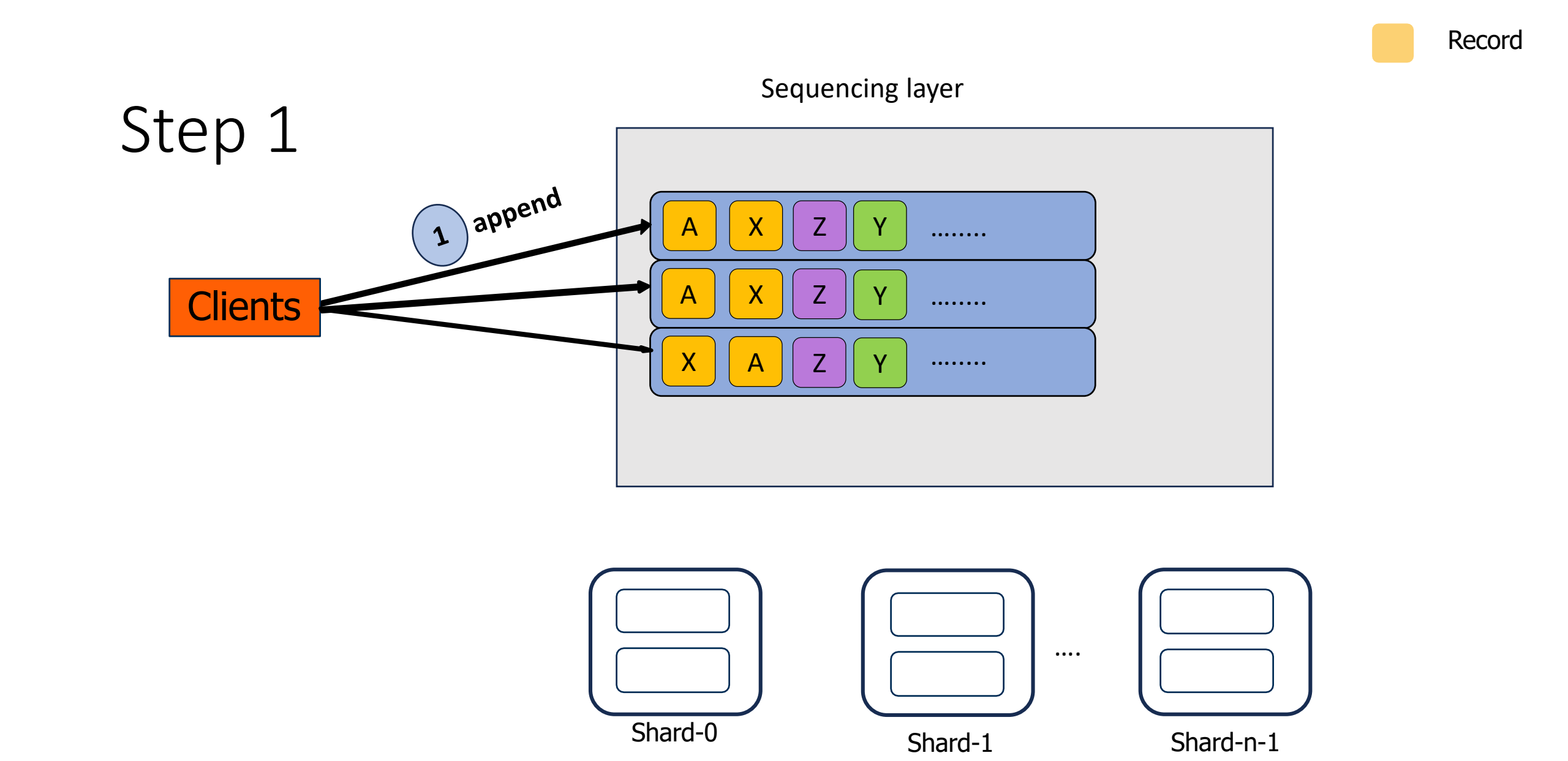
Erwin's Architecture

- Has set of unmodified primary-backup shards and **Sequencing layer**
- **Sequencing layer:**
  - Fault-tolerant : f+1 replicas to tolerate f failures
  - Coordination-free
  - Provide only short-term durability

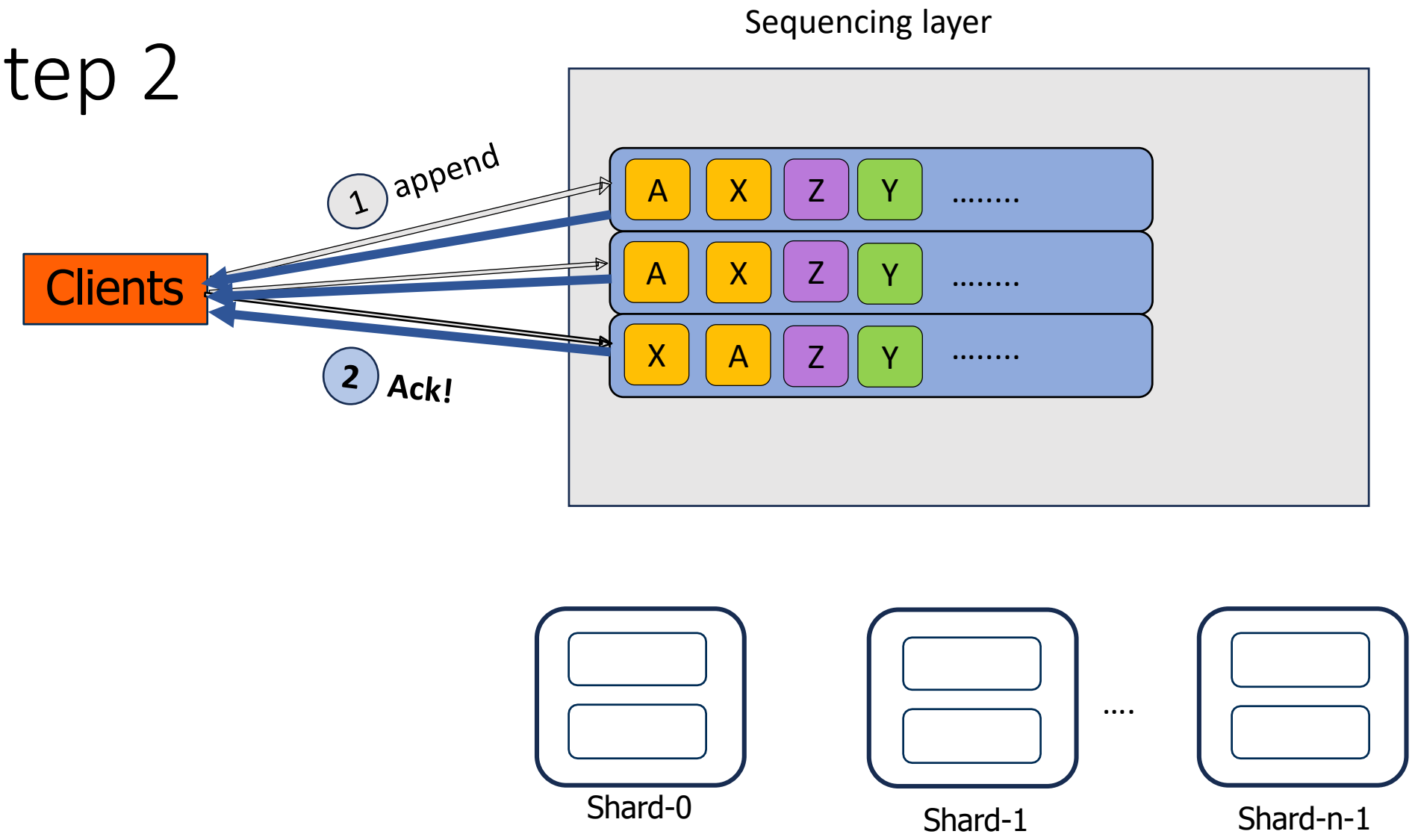
How does Erwin Append work?

How is Lazy ordering done?

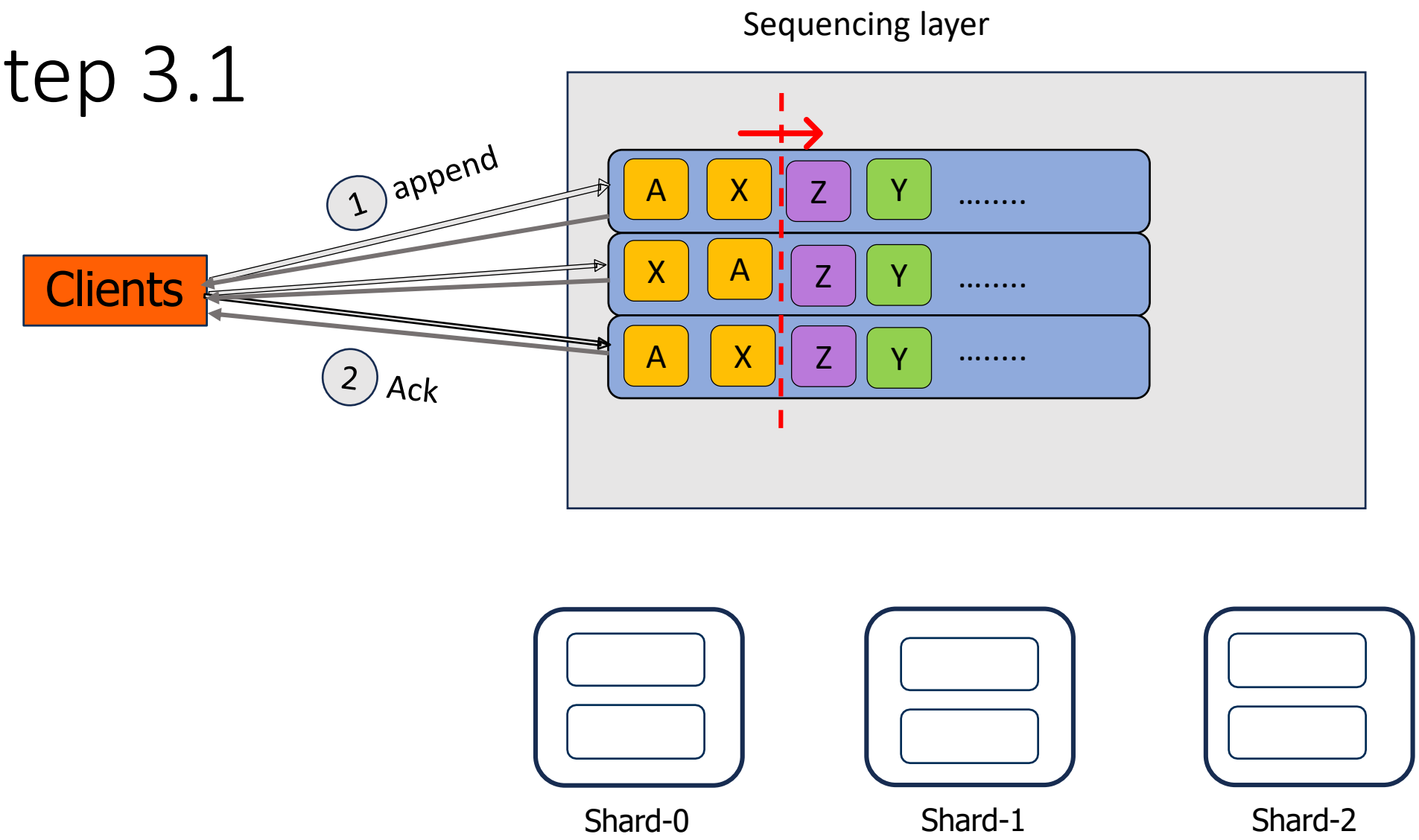
Step by step


Record
Step 1
Sequencing layer
1
append
Clients
A X Z Y .....
A X Z Y .....
X A Z Y .....
....
Shard-0
Shard-1
Shard-n-1

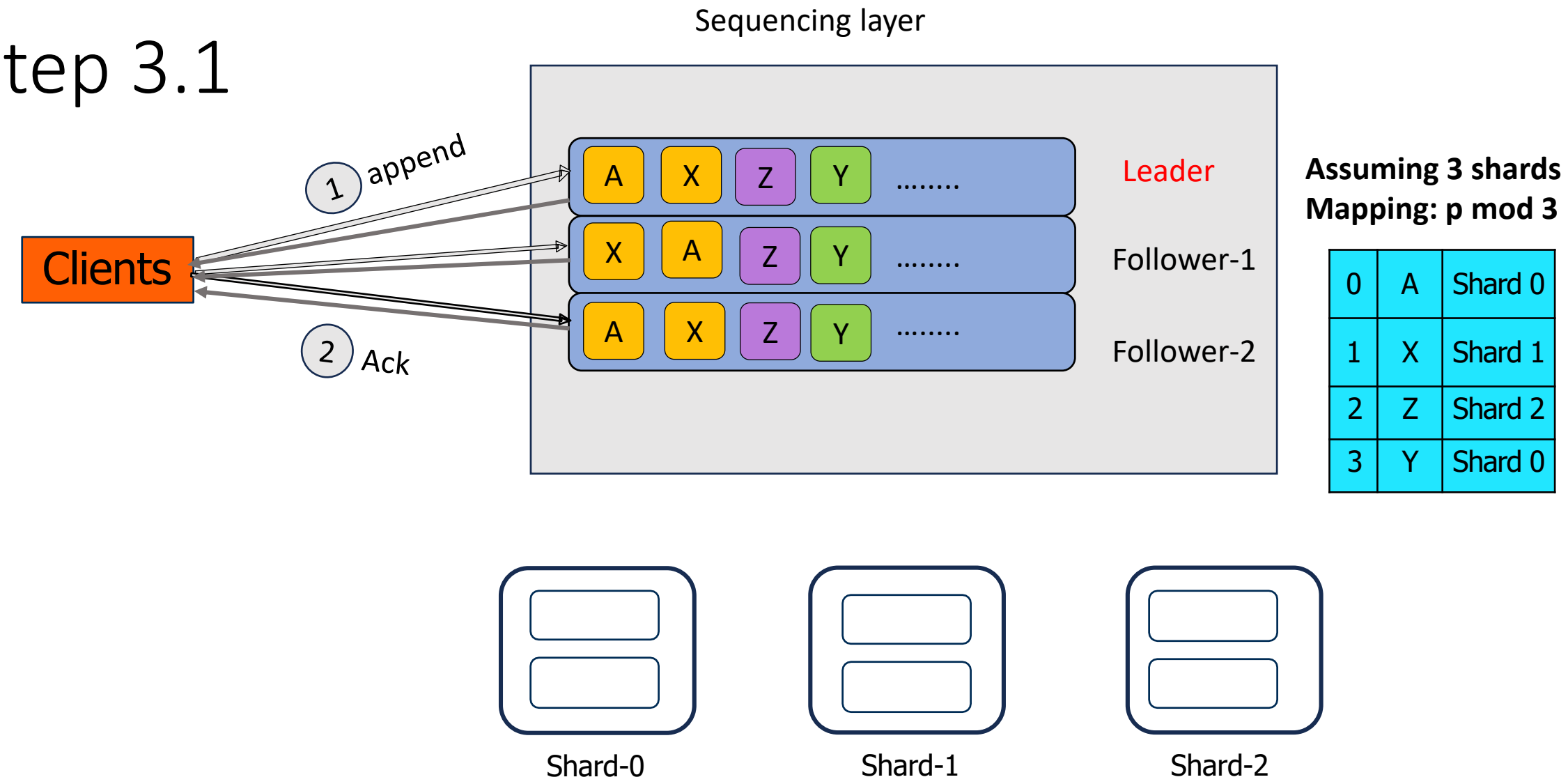
Step 2


Record
Sequencing layer
1 append
Clients
2 Ack!
A X Z Y .....
A X Z Y .....
X A Z Y .....
....
Shard-0
Shard-1
Shard-n-1

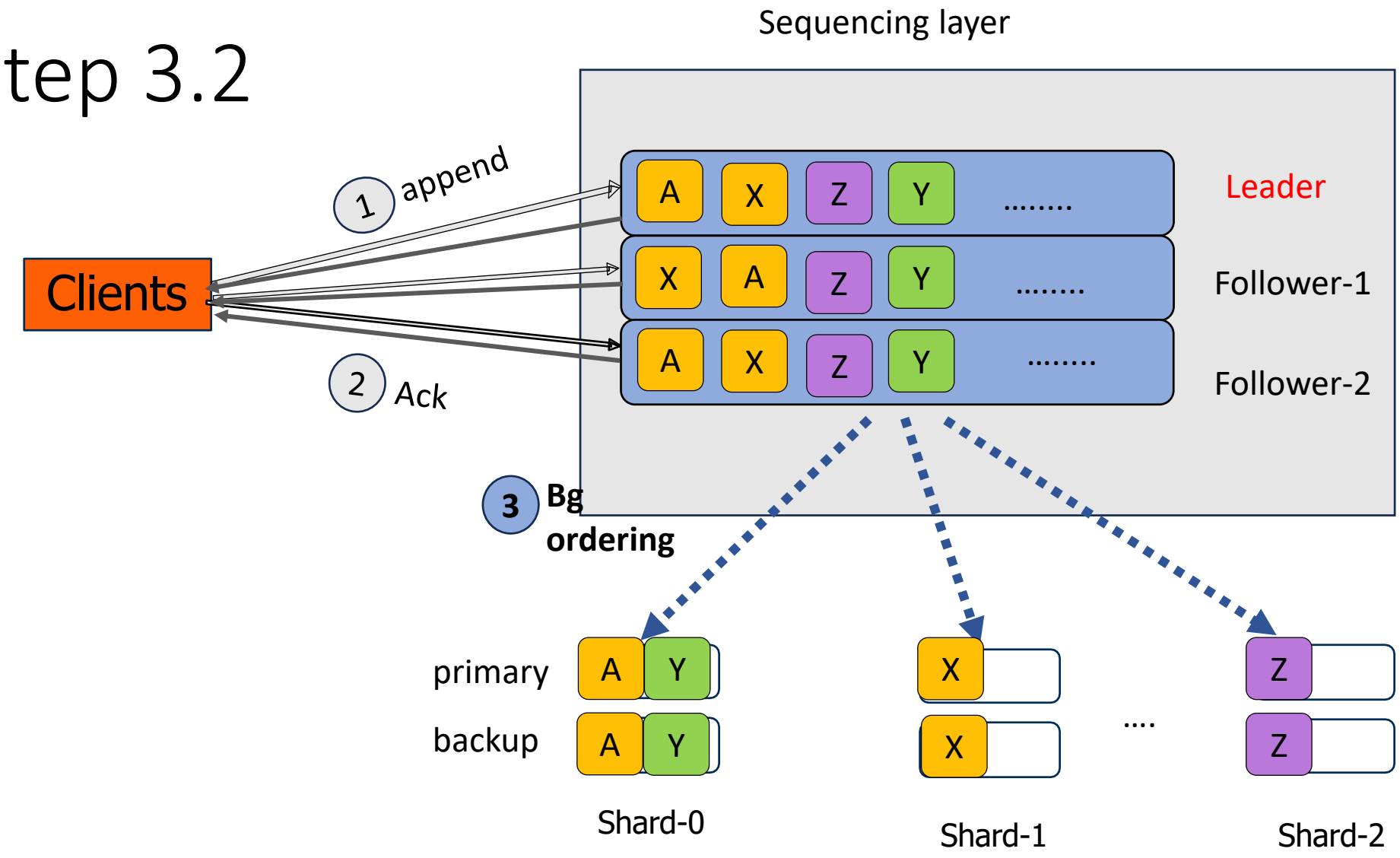
Step 3.1


Record
Sequencing layer
1 append
Clients
2 Ack
A X Z Y .....
X A Z Y .....
A X Z Y .....
Shard-0
Shard-1
Shard-2

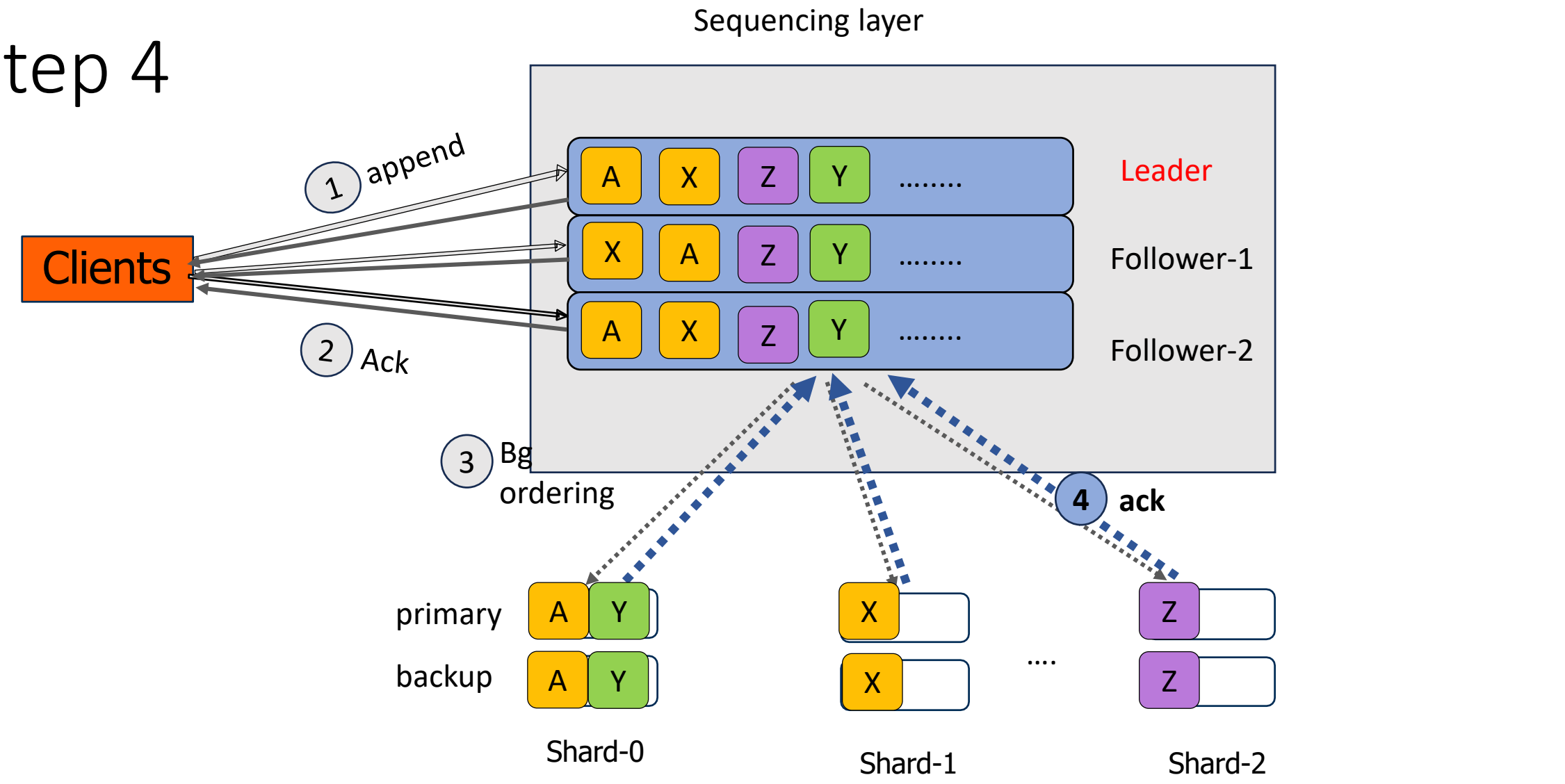
Step 3.1


Record
Sequencing layer
1 append
Clients
2 Ack
A X Z Y .....
X A Z Y .....
A X Z Y .....
Leader
Follower-1
Follower-2
Assuming 3 shards
Mapping: p mod 3
0 A Shard 0
1 X Shard 1
2 Z Shard 2
3 Y Shard 0
Shard-0
Shard-1
Shard-2

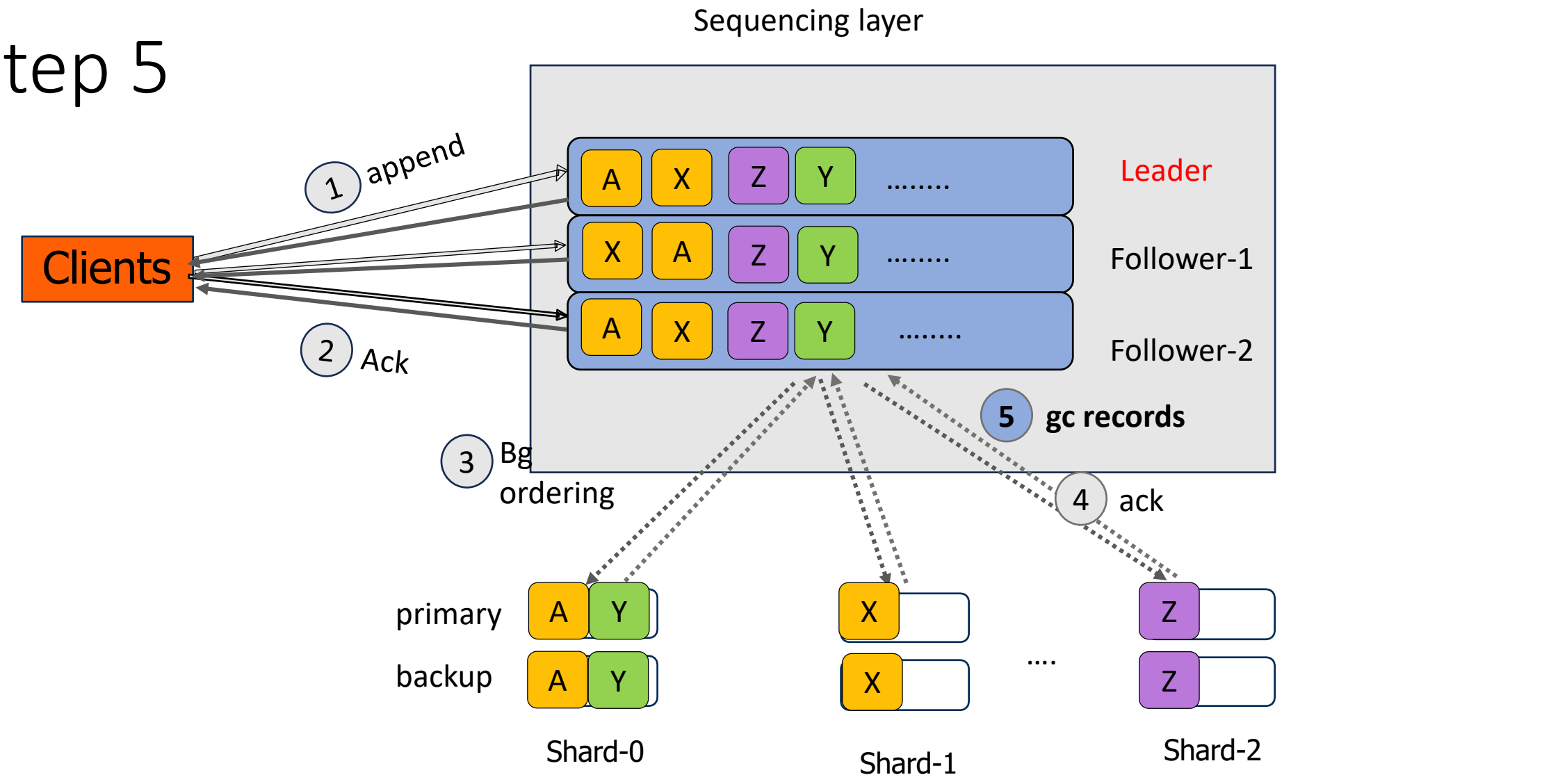
Step 3.2


Record
Sequencing layer
1 append
Clients
2 Ack
A X Z Y .....
Leader
X A Z Y .....
Follower-1
A X Z Y .....
Follower-2
3 Bg ordering
primary
backup
A Y
A Y
X
X
....
Z
Z
Shard-0
Shard-1
Shard-2

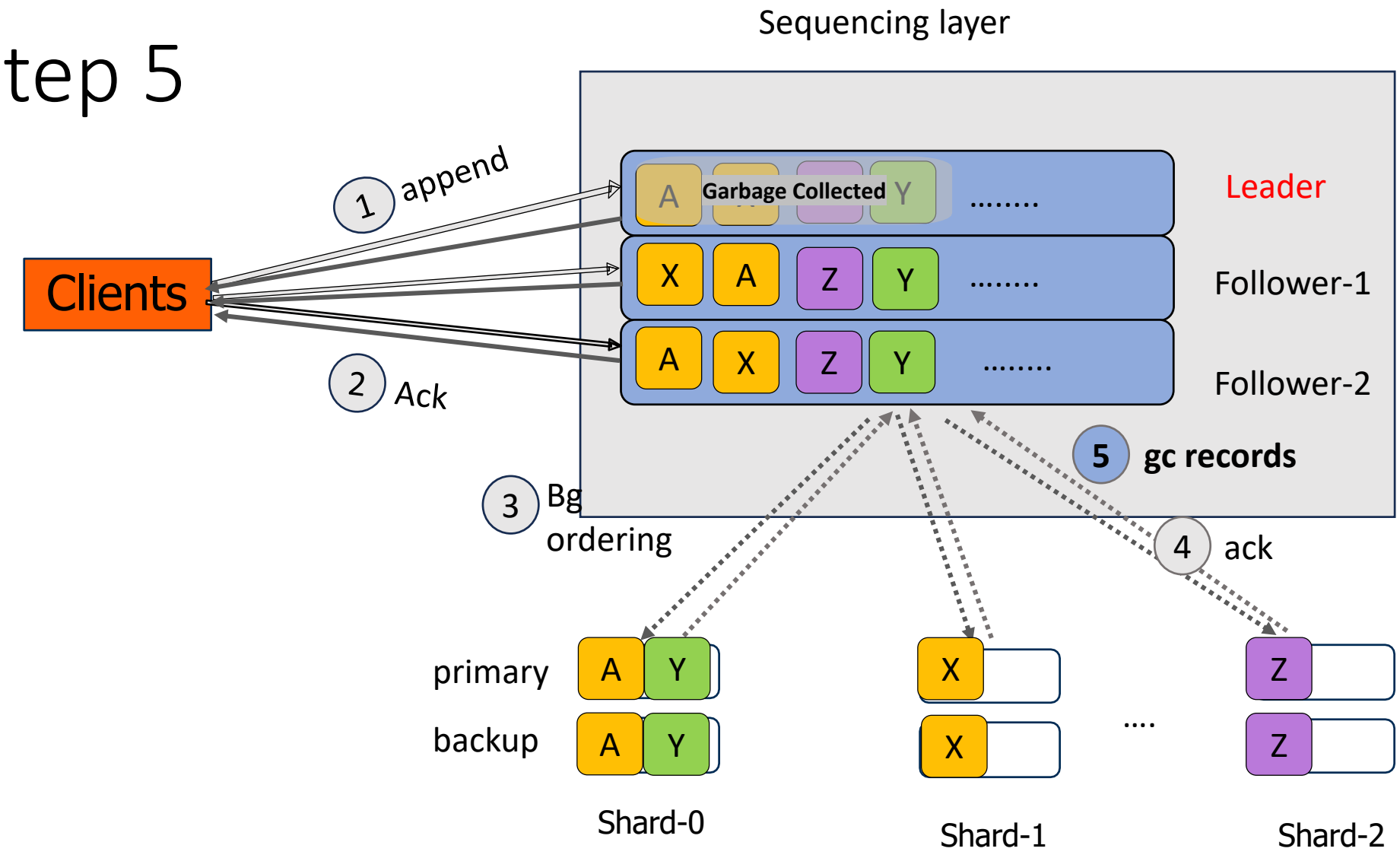
Step 4


Record
Sequencing layer
Clients
1 append
2 Ack
A X Z Y .....
Leader
X A Z Y .....
Follower-1
A X Z Y .....
Follower-2
3 Bg ordering
4 ack
primary
backup
A Y
A Y
X
X
....
Z
Z
Shard-0
Shard-1
Shard-2

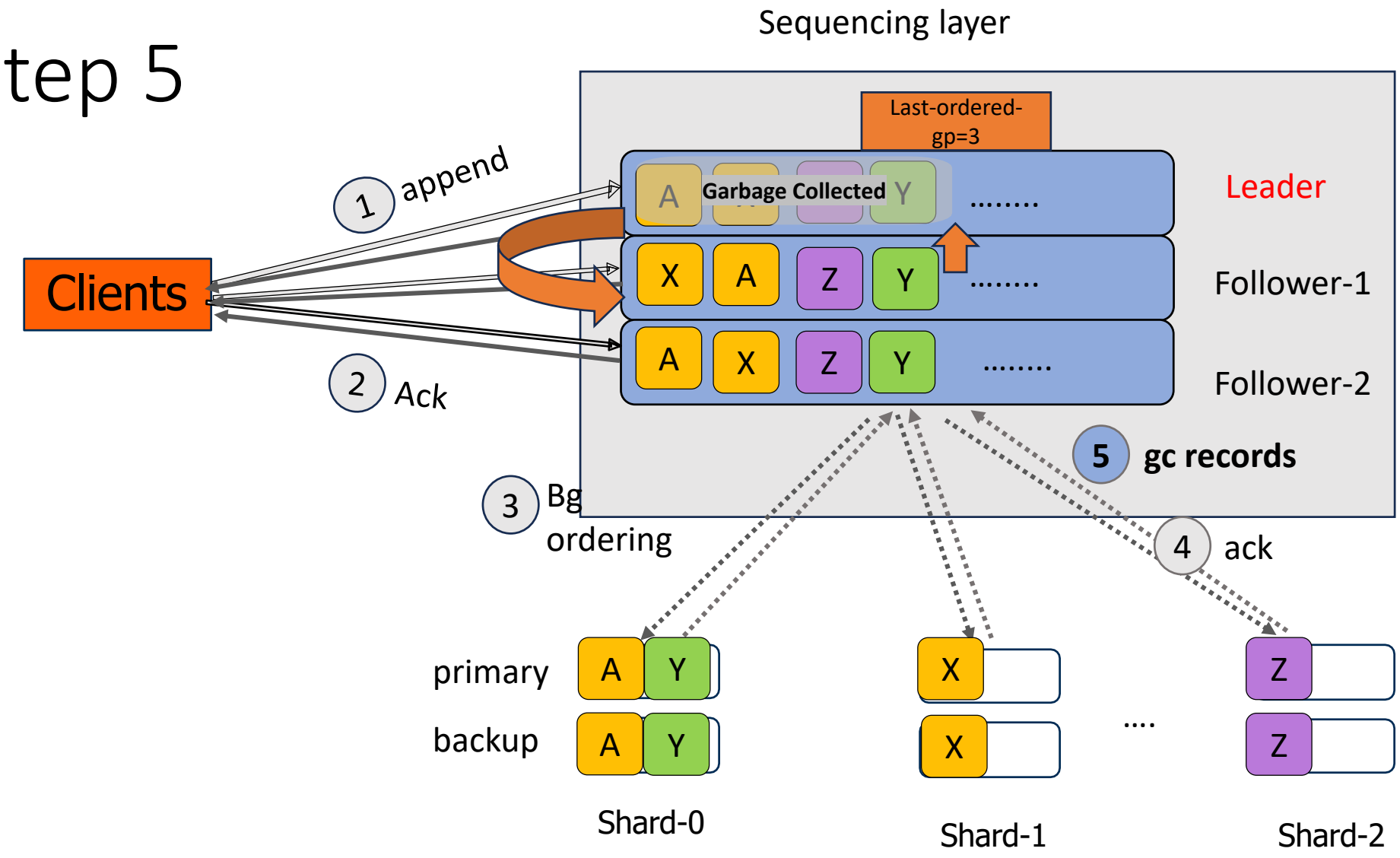
Step 5


Record
Sequencing layer
Clients
1 append
2 Ack
A X Z Y ……..
Leader
X A Z Y ……..
Follower-1
A X Z Y ……..
Follower-2
5 gc records
3 Bg ordering
4 ack
primary
backup
A Y
A Y
X
X
Z
Z
….
Shard-0
Shard-1
Shard-2

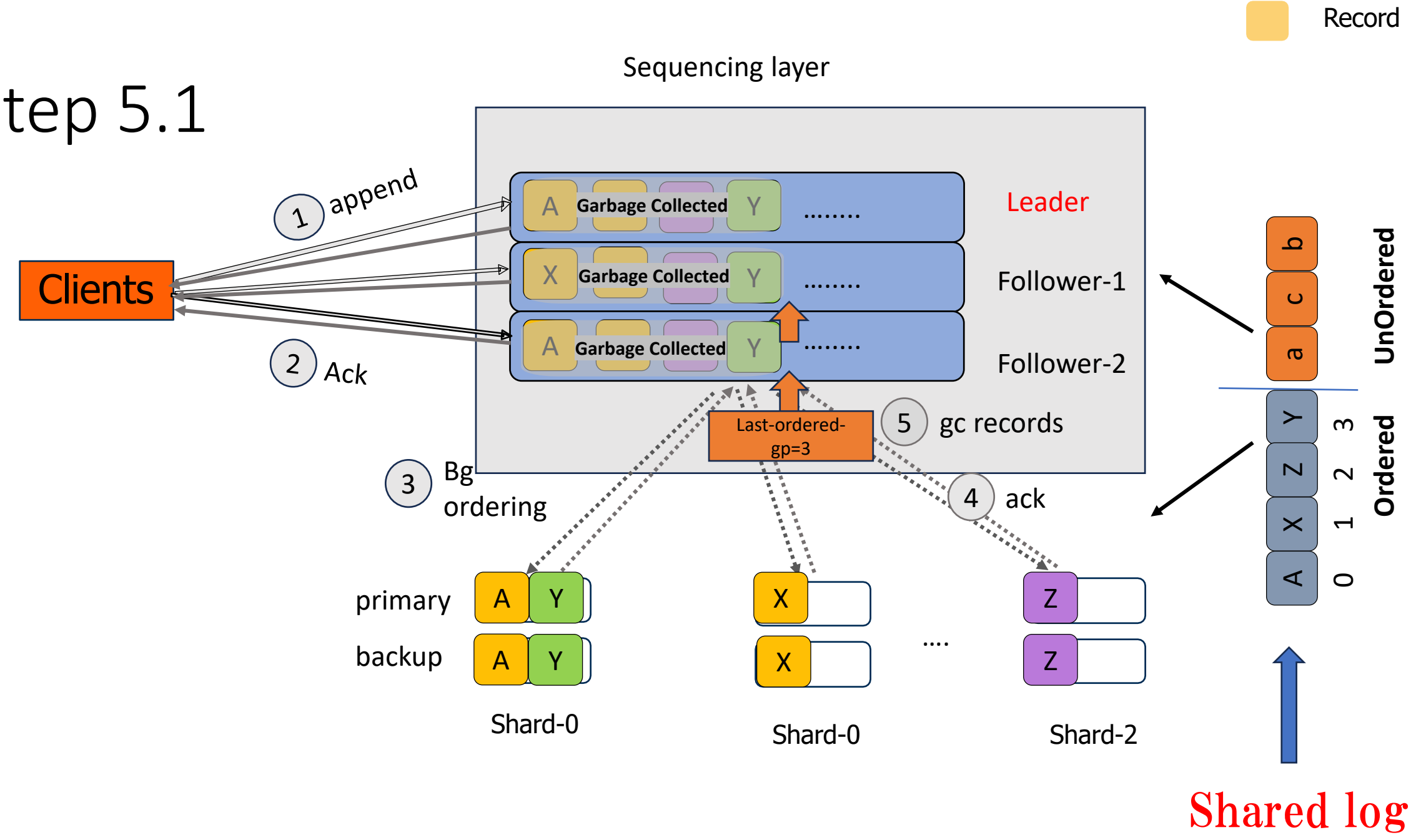
Step 5


Record
Sequencing layer
Clients
1 append
2 Ack
A
Garbage Collected
Y
.....
Leader
X
A
Z
Y
.....
Follower-1
A
X
Z
Y
.....
Follower-2
5 gc records
3 Bg ordering
4 ack
primary
backup
A
Y
X
Z
....
Shard-0
Shard-1
Shard-2

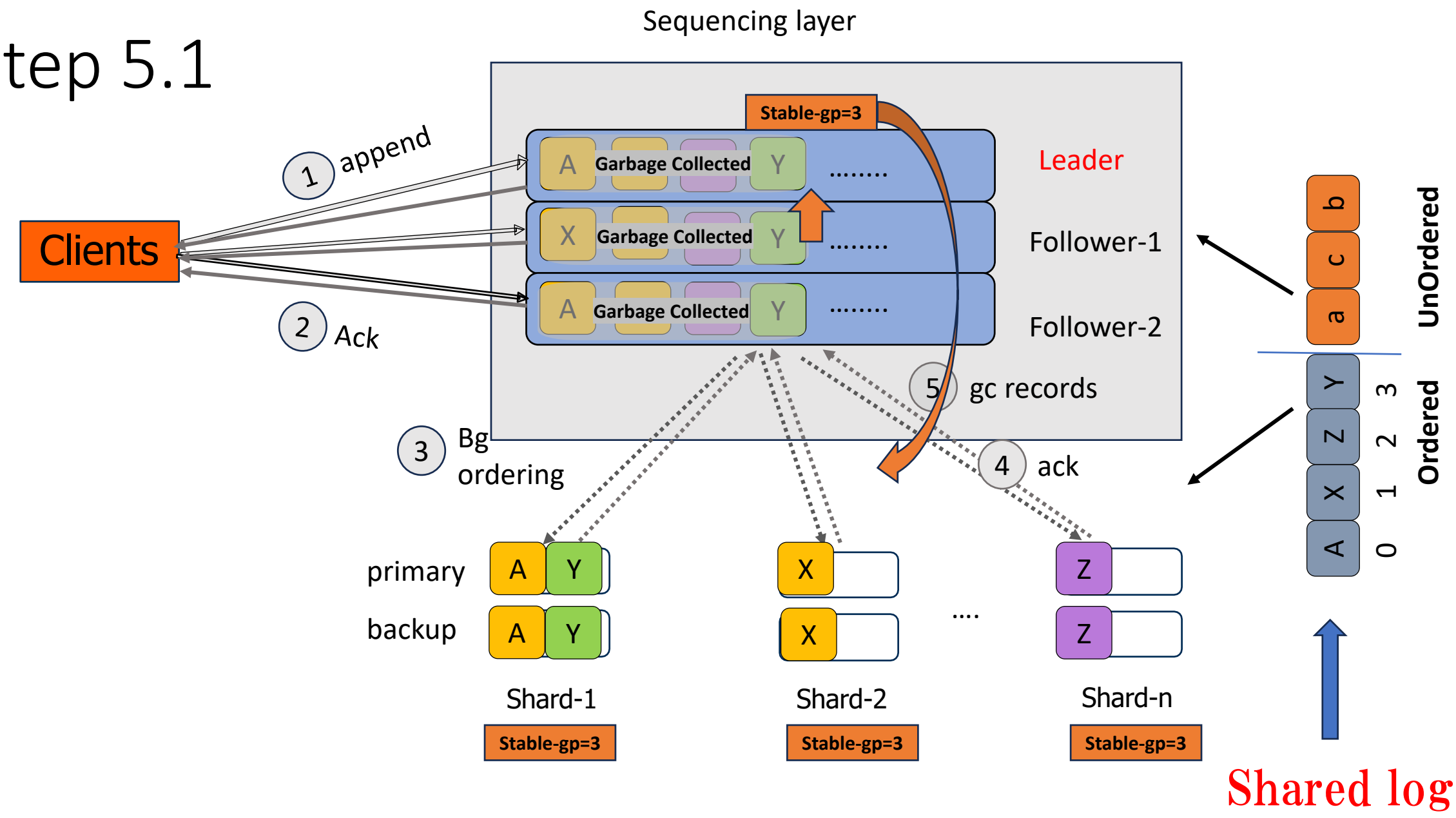
Step 5


Record
Sequencing layer
Last-ordered-gp=3
Garbage Collected
Leader
Follower-1
Follower-2
Clients
1 append
2 Ack
3 Bg ordering
4 ack
5 gc records
primary
backup
Shard-0
Shard-1
Shard-2

Step 5.1


Record
Sequencing layer
Clients
1 append
2 Ack
A Garbage Collected Y .....
X Garbage Collected Y .....
A Garbage Collected Y .....
Leader
Follower-1
Follower-2
Last-ordered-gp=3
5 gc records
3 Bg ordering
4 ack
primary
backup
A Y
A Y
X
X
Z
Z
....
Shard-0
Shard-0
Shard-2
a c b
A X Z Y
0 1 2 3
UnOrdered
Ordered
Shared log

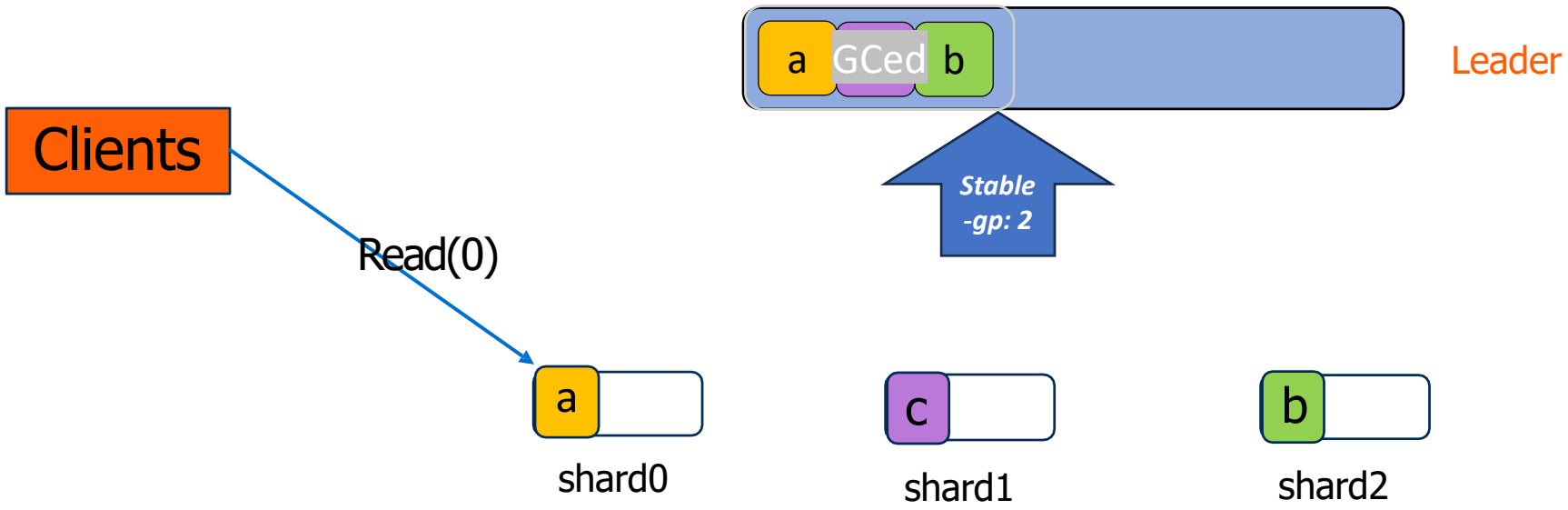
Step 5.1


Record
Sequencing layer
Stable-gp=3
A
Garbage Collected
Y
X
Garbage Collected
Y
A
Garbage Collected
Y
Leader
Follower-1
Follower-2
Clients
1 append
2 Ack
3 Bg ordering
4 ack
5 gc records
primary
backup
A
Y
X
Z
Shard-1
Shard-2
Shard-n
Stable-gp=3
UnOrdered
Ordered
0 1 2 3
Shared log

How does Erwin Read work?

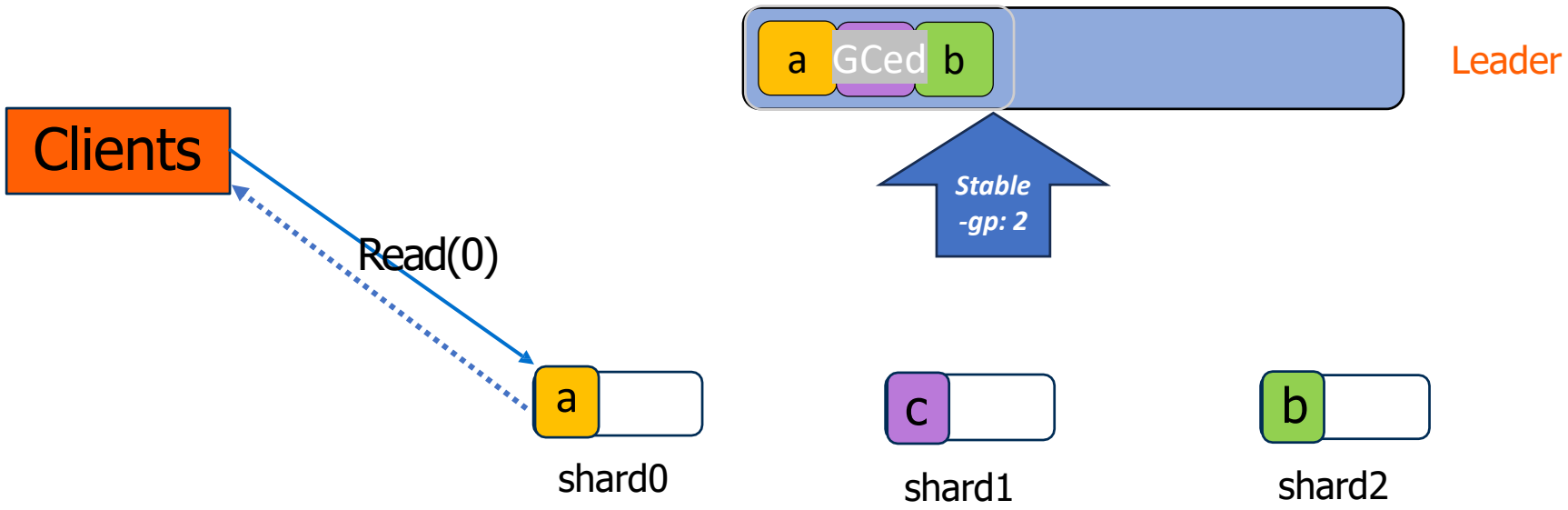
Step by step

Erwin: Read



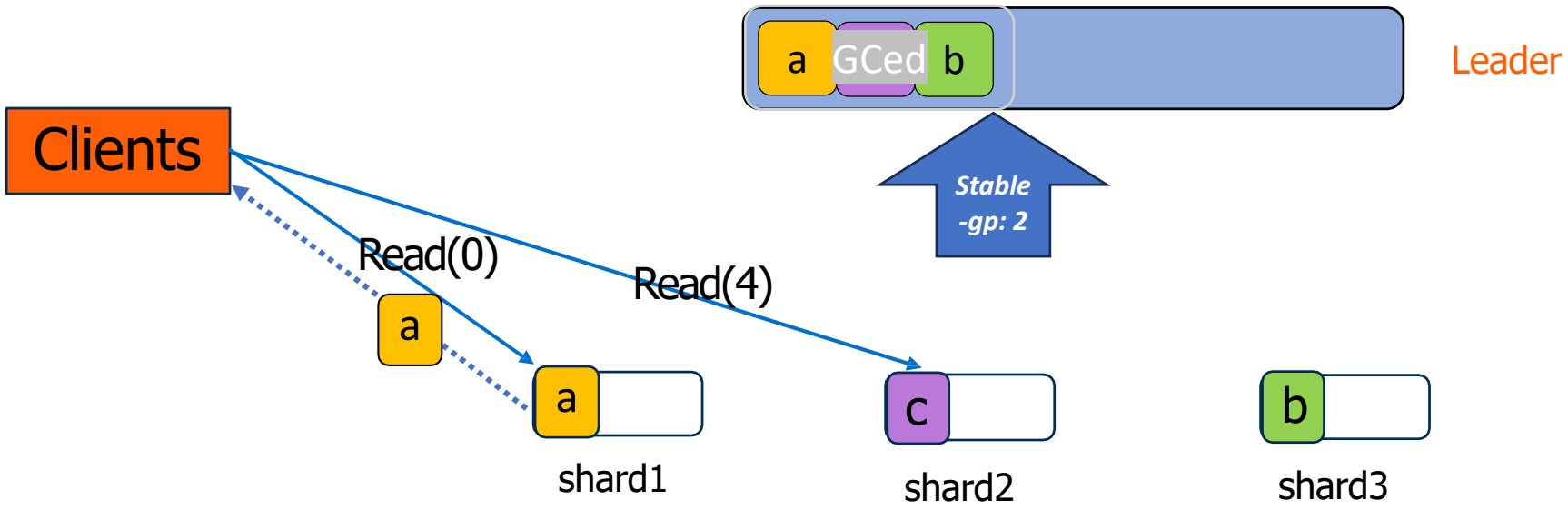
- Reading ordered position (fast read): entry returned directly

Erwin: Read



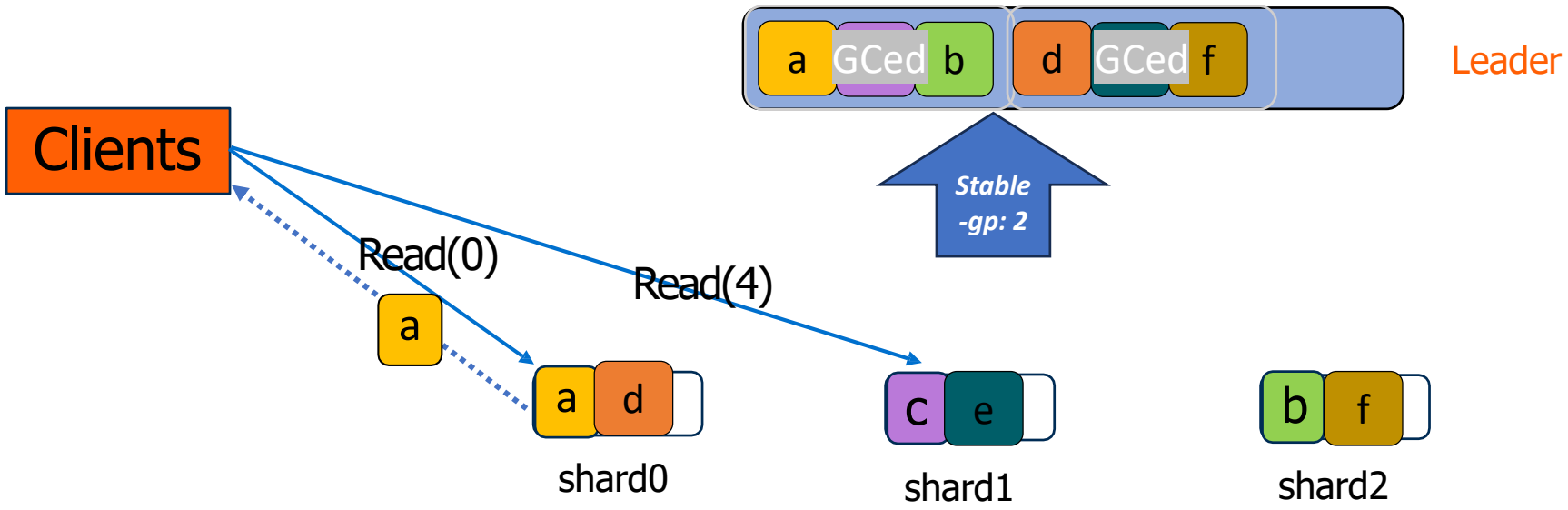
- Reading ordered position (fast read): entry returned directly

Erwin: Read



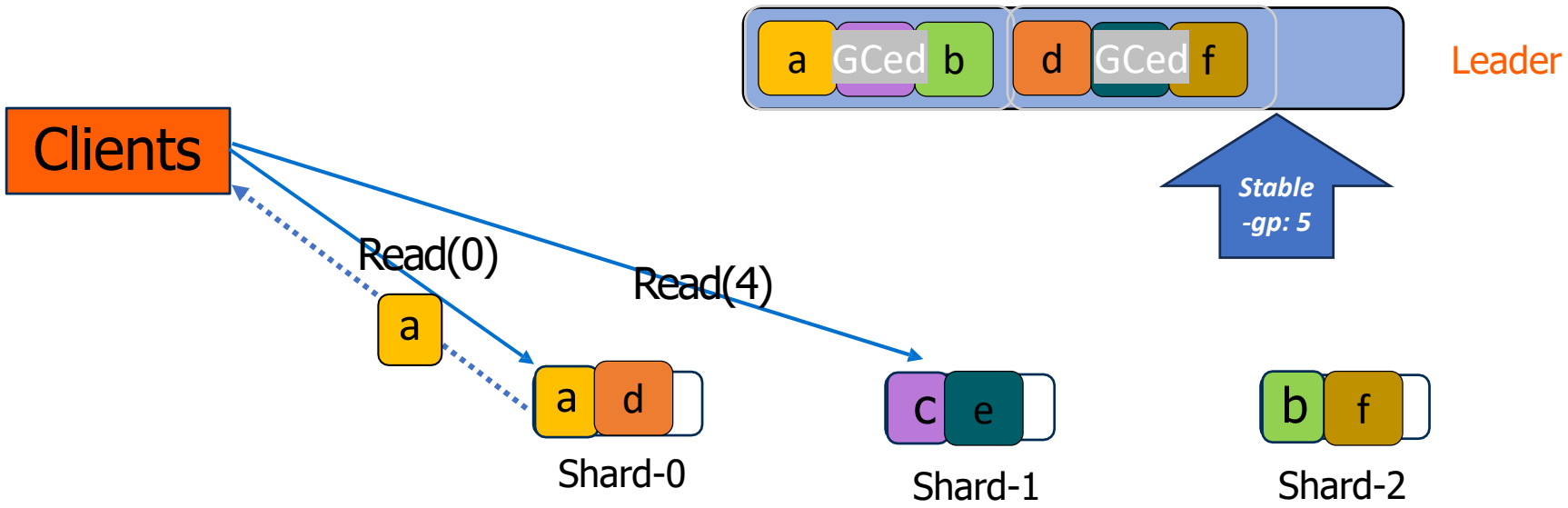
- Reading ordered position (fast read): entry returned directly
- Reading unordered position (slow read): must wait until ***stable-gp*** is advanced to the read position

Erwin: Read



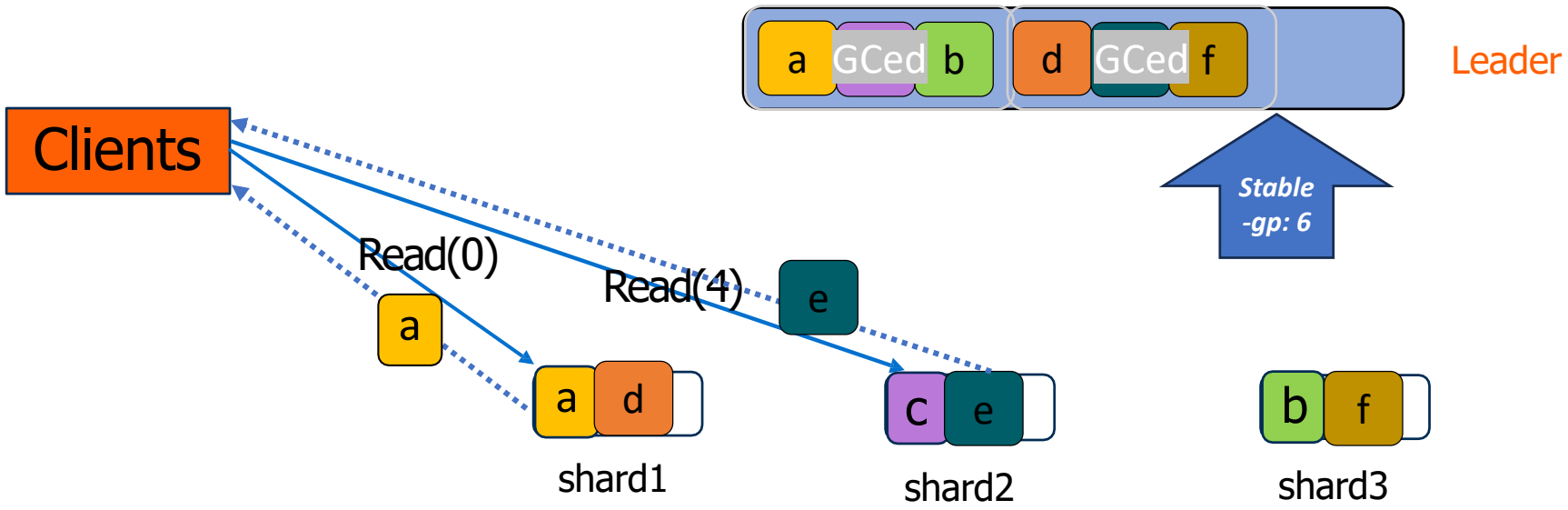
- Reading ordered position (fast read): entry returned directly
- Reading unordered position (slow read): must wait until ***stable-gp*** is advanced to the read position

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



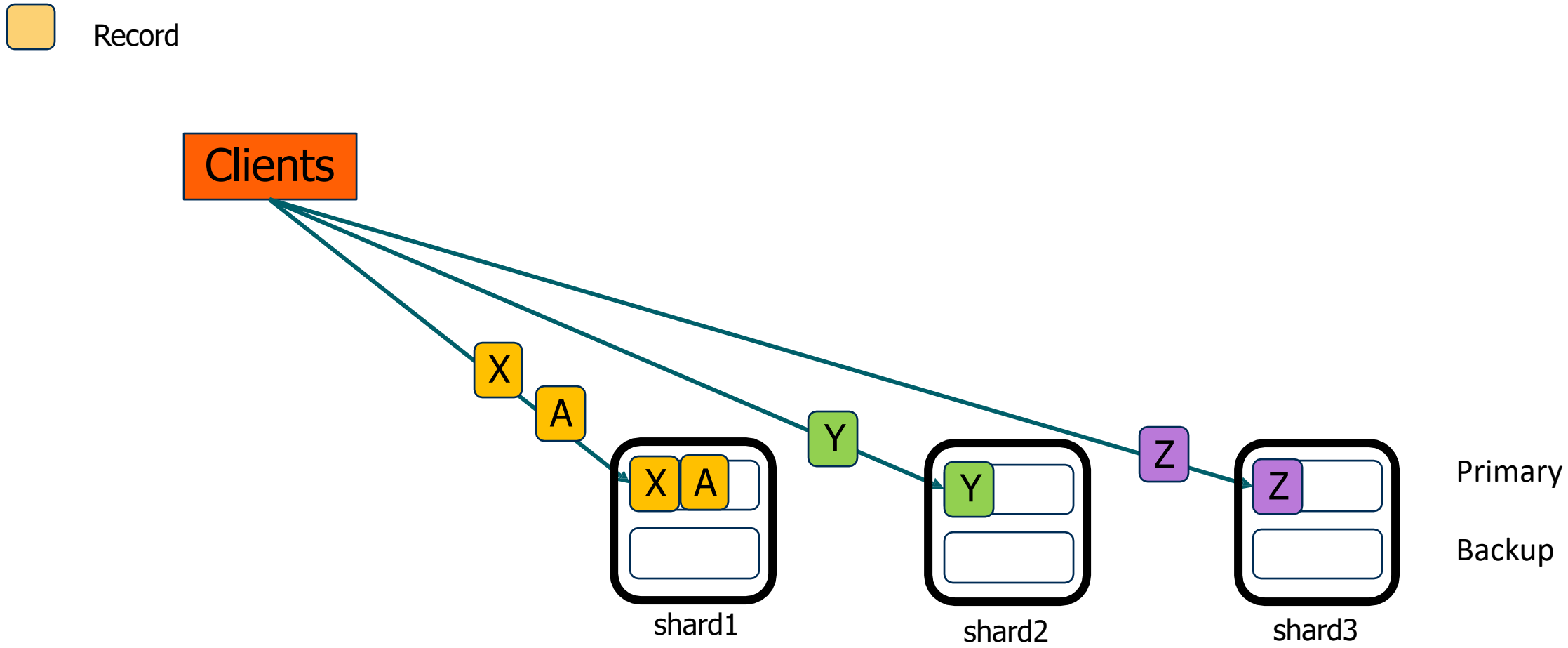
- Reading ordered position (fast read): entry returned directly
- Reading unordered position (slow read): must wait until ***stable-gp*** is advanced to the read position

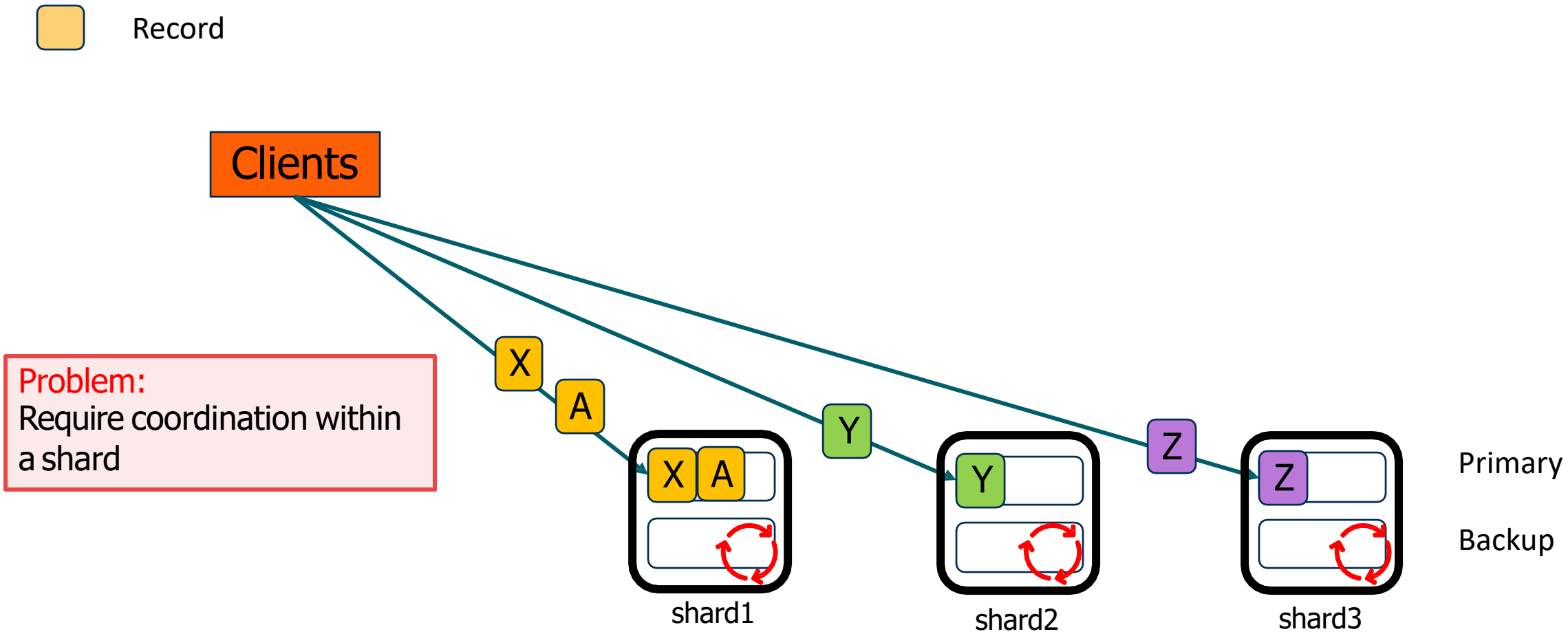
Problem with Erwin Blackbox

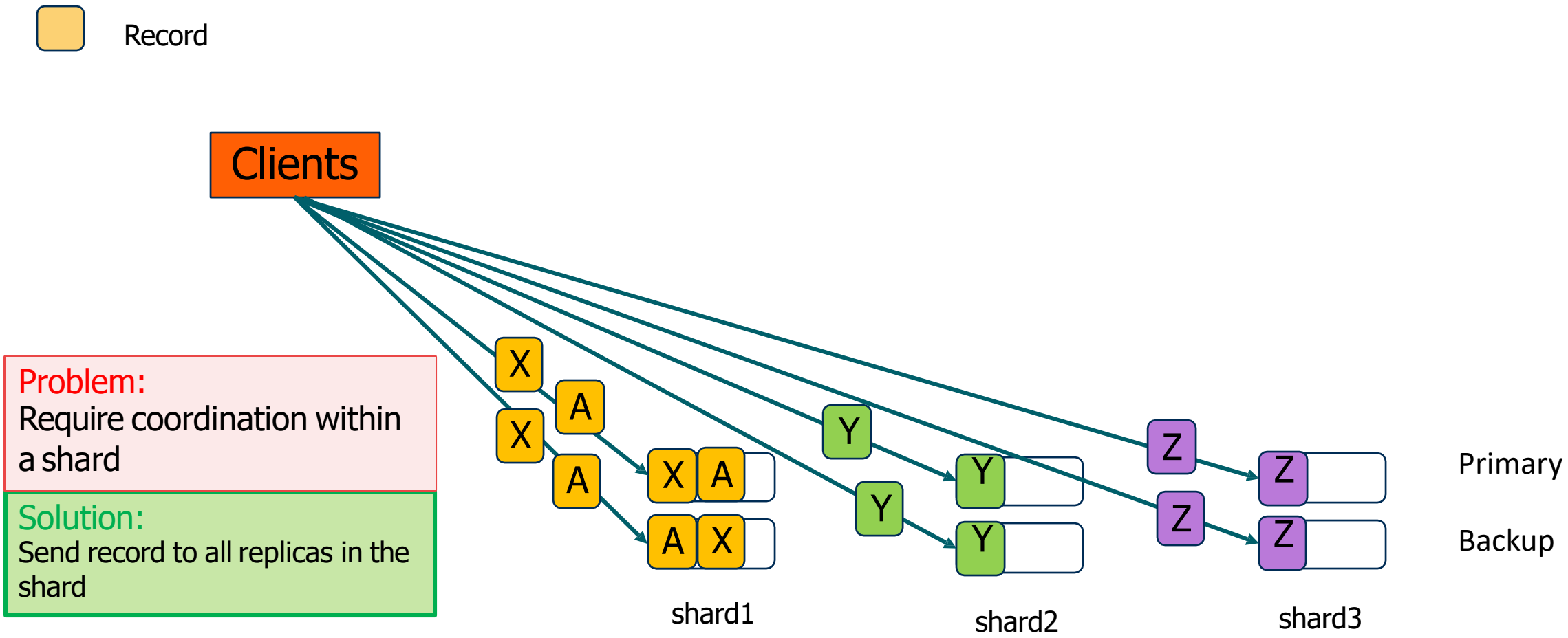
Large Record --> Sequencing layer will be saturated

Solution

Split Record --> into Data and Metadata (Record_Id, Shard_Id)


Record
Clients
X
A
Y
Z
X
A
Y
Z
shard1
shard2
shard3
Primary
Backup

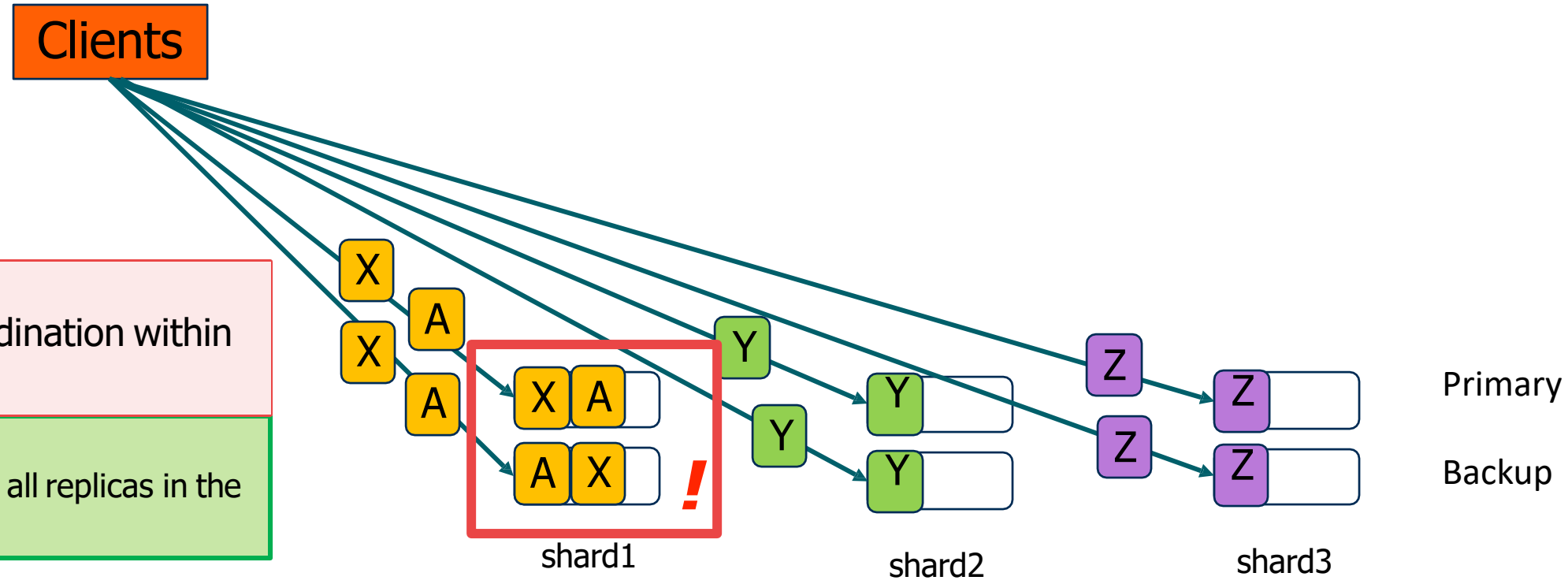

Record
Clients
Problem:
Require coordination within a shard
X
A
Y
Z
X
A
Y
Z
shard1
shard2
shard3
Primary
Backup


Record
Clients
X
X
A
A
X
A
A
X
Y
Y
Y
Y
Z
Z
Z
Z
shard1
shard2
shard3
Primary
Backup
Problem:
Require coordination within a shard
Solution:
Send record to all replicas in the shard

Motivation

Record

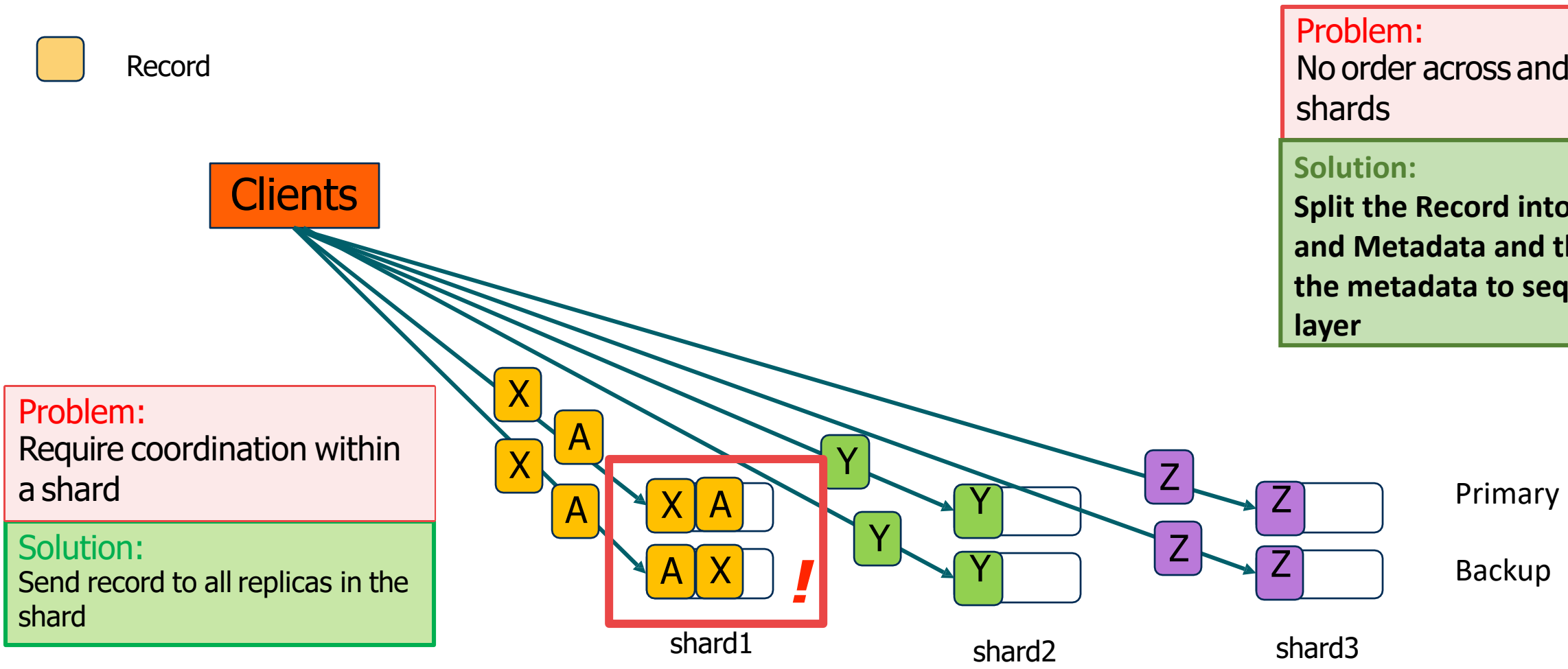
Problem:
No order across and within shards



Problem:
Require coordination within a shard

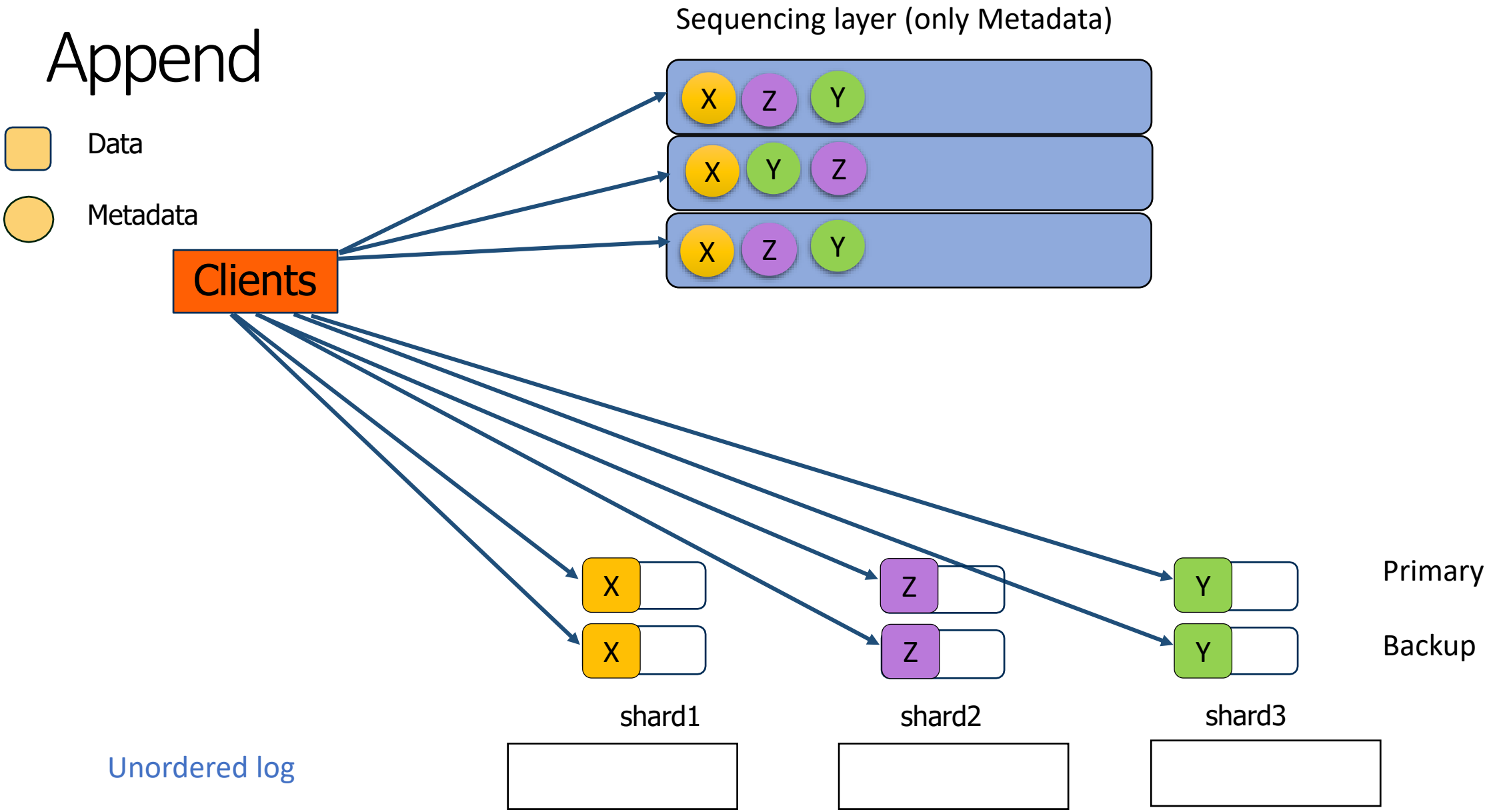
Solution:
Send record to all replicas in the shard

Motivation

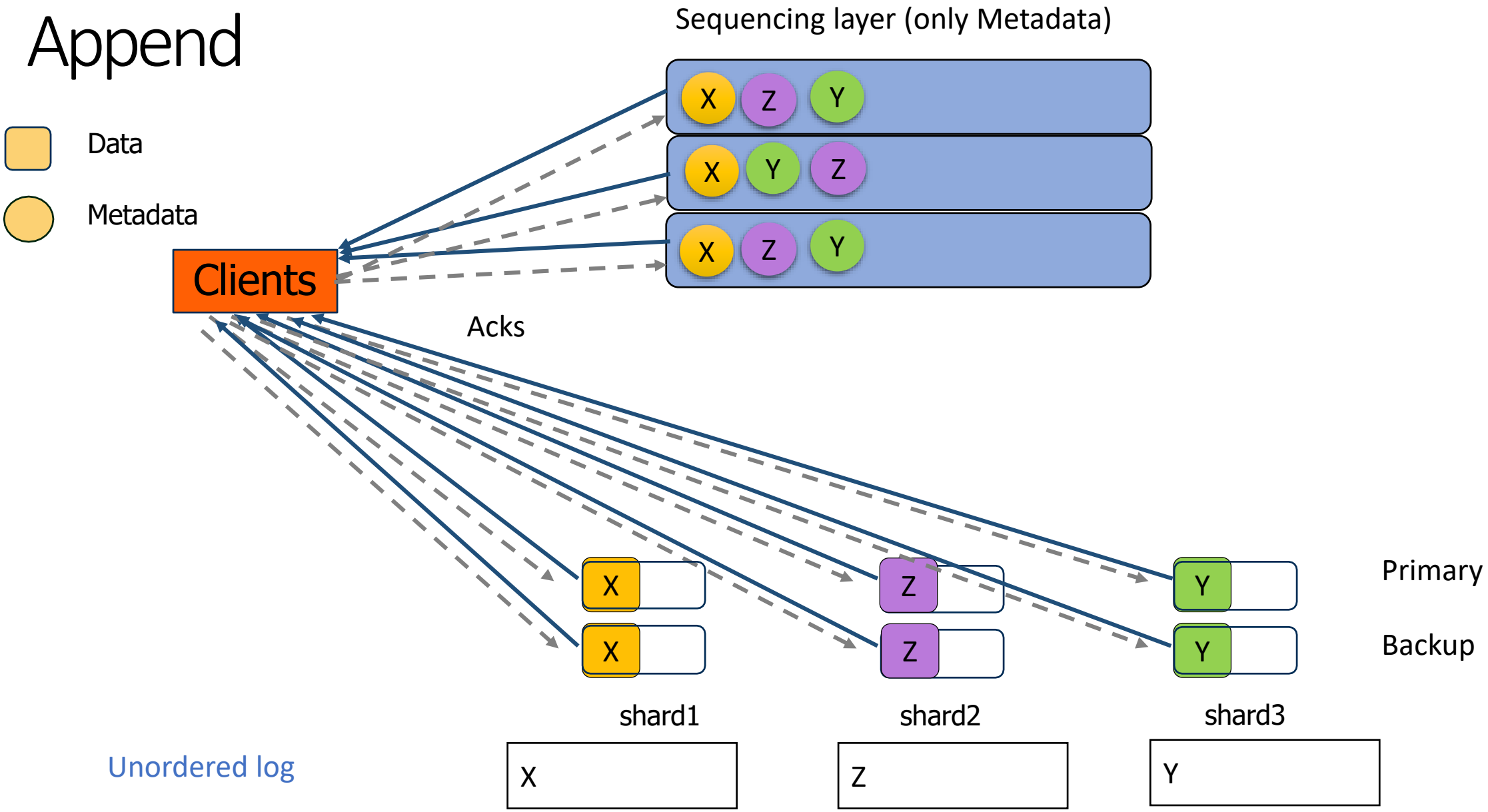

Record
Clients
Problem:
No order across and within shards
Solution:
Split the Record into Data and Metadata and then send the metadata to sequencing layer
Problem:
Require coordination within a shard
Solution:
Send record to all replicas in the shard
X
X
A
A
X
A
A
X
!
Y
Y
Y
Y
Z
Z
Z
Z
shard1
shard2
shard3
Primary
Backup

Erwin-st(**s**calable **t**hroughput)

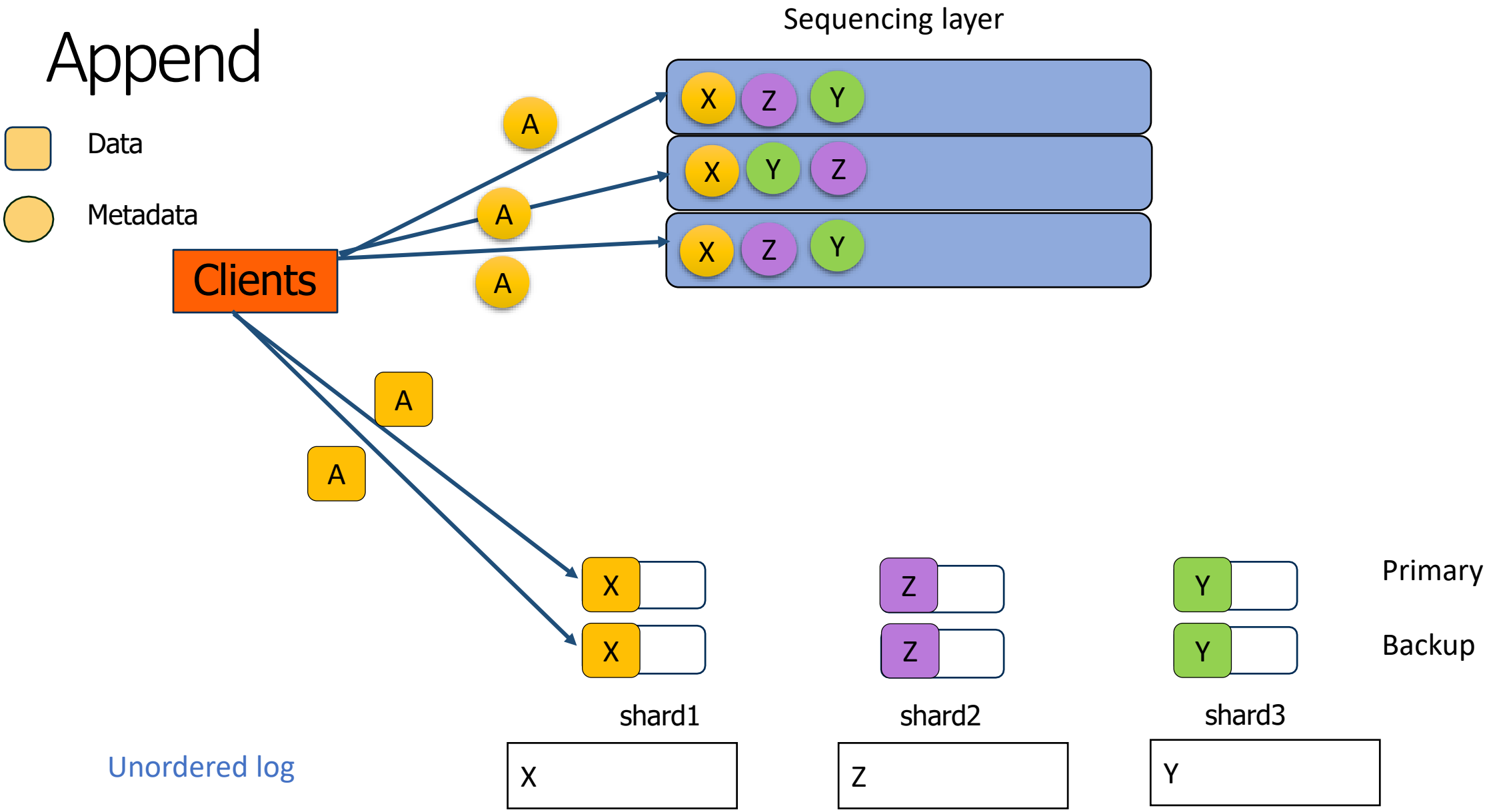
Erwin's Goal: 1-RTT Append



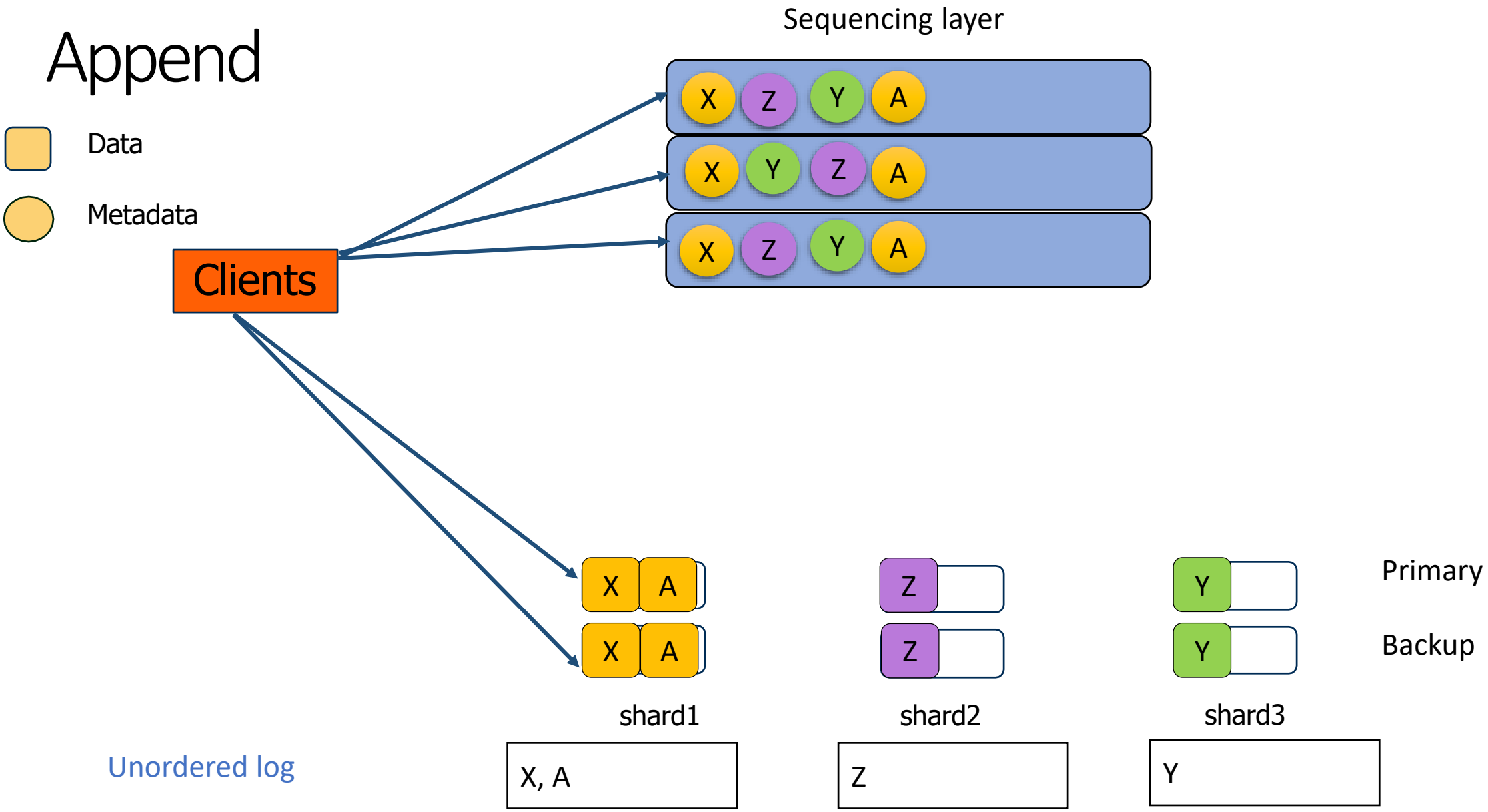
Erwin's Goal: 1-RTT Append



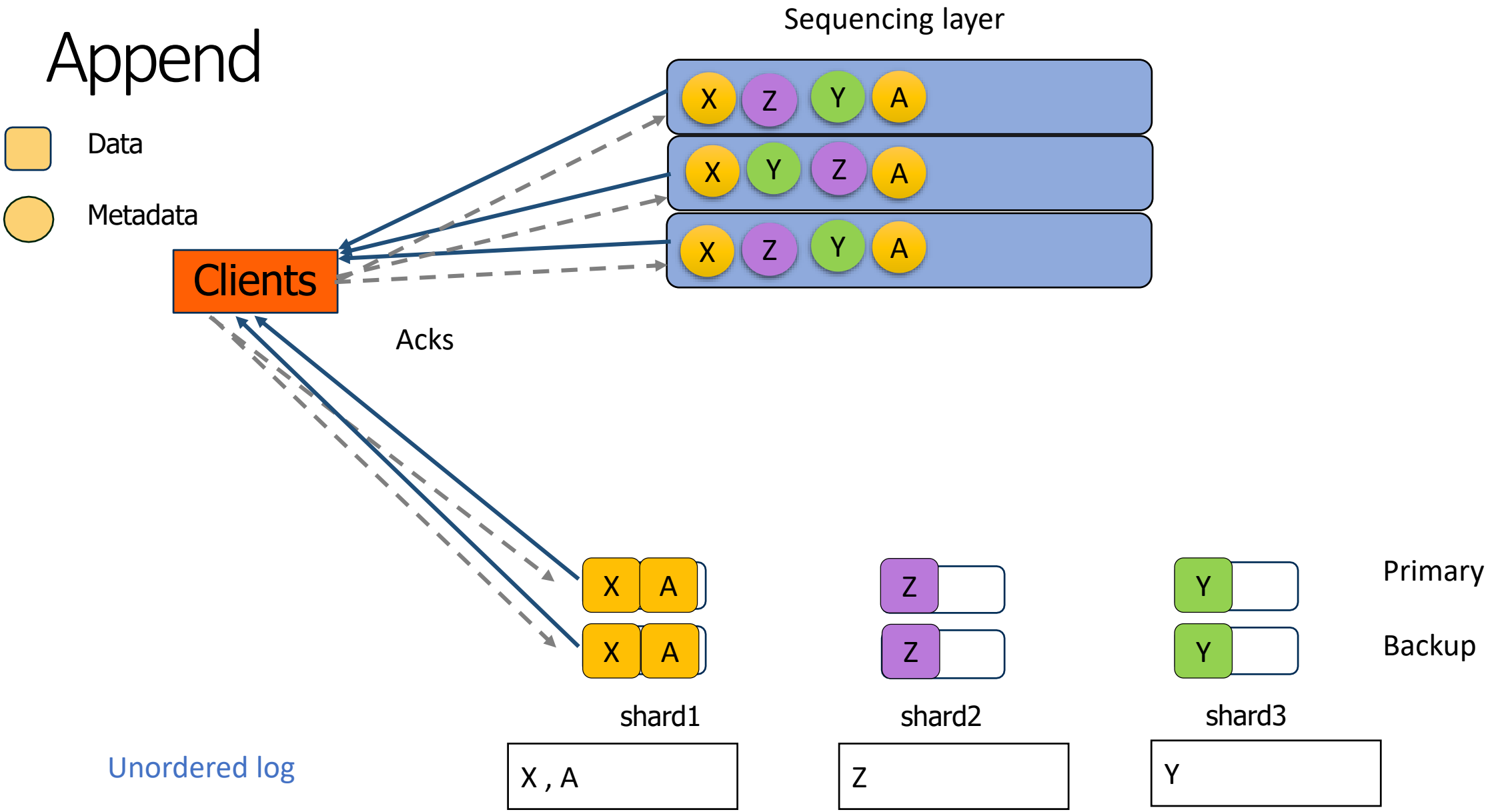
Erwin's Goal: 1-RTT Append



Erwin's Goal: 1-RTT Append



Erwin's Goal: 1-RTT Append

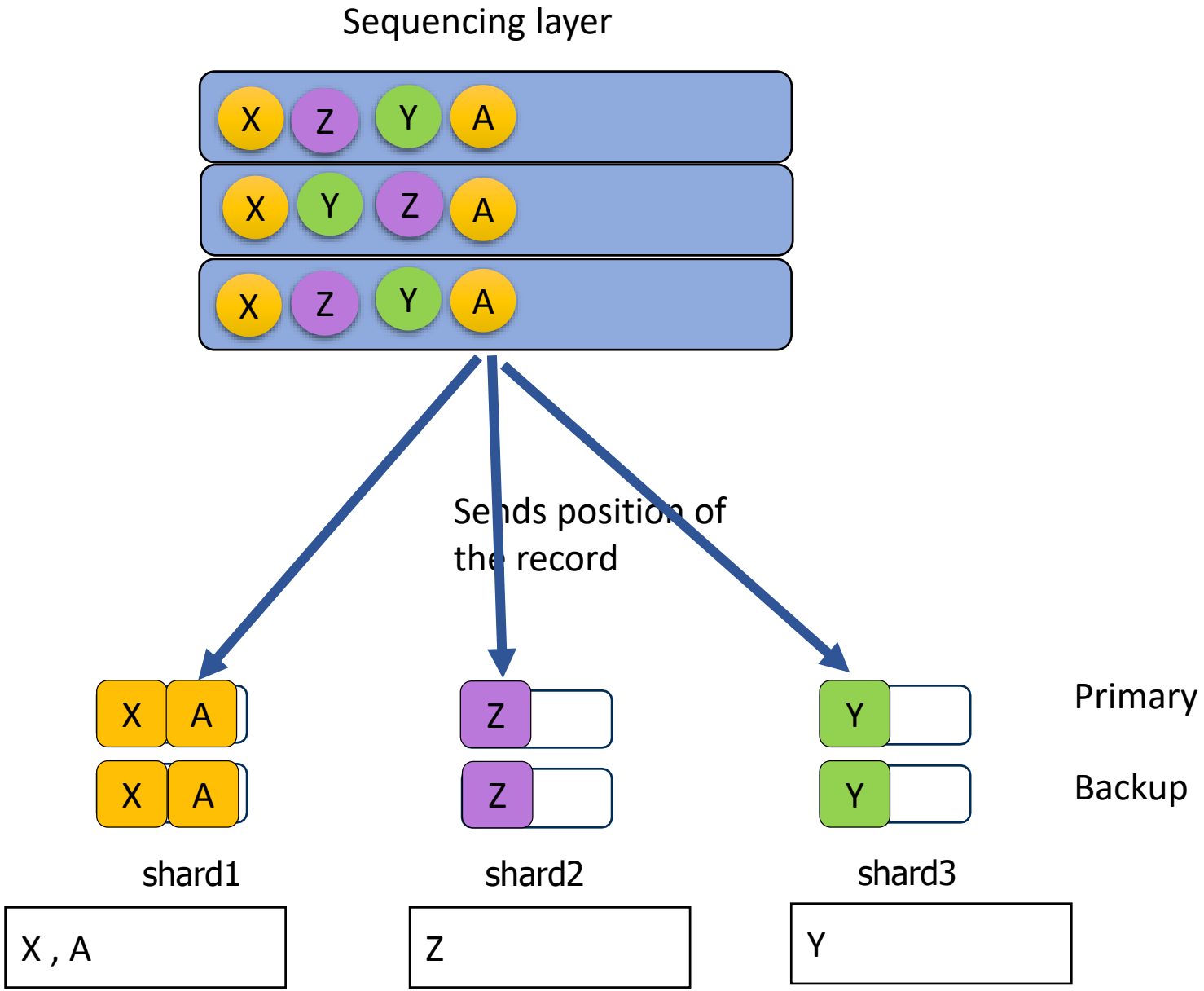


Erwin's Goal: 1-RTT Append

Data

Metadata

Clients



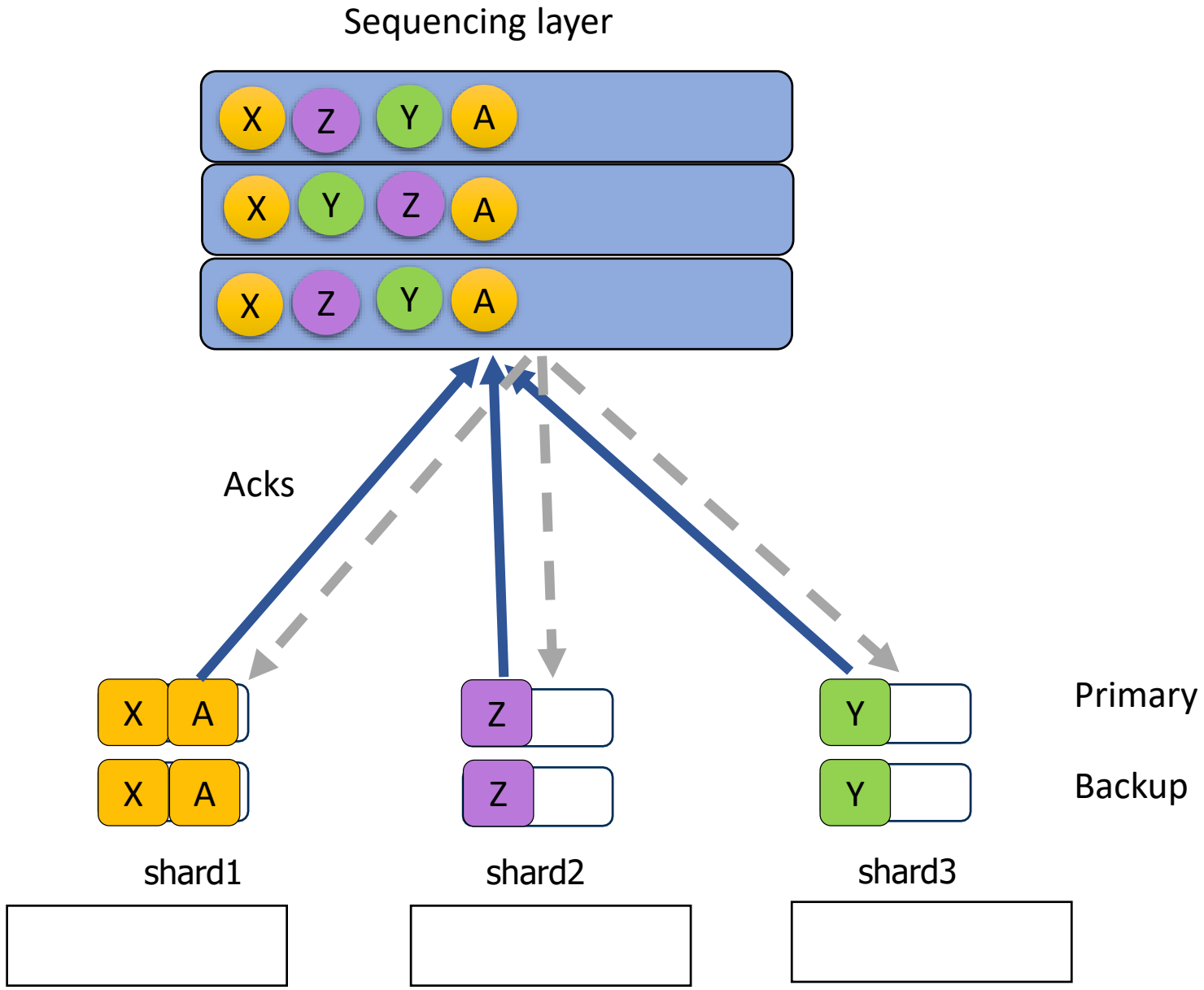
Unordered log

Erwin's Goal: 1-RTT Append

Data

Metadata

Clients



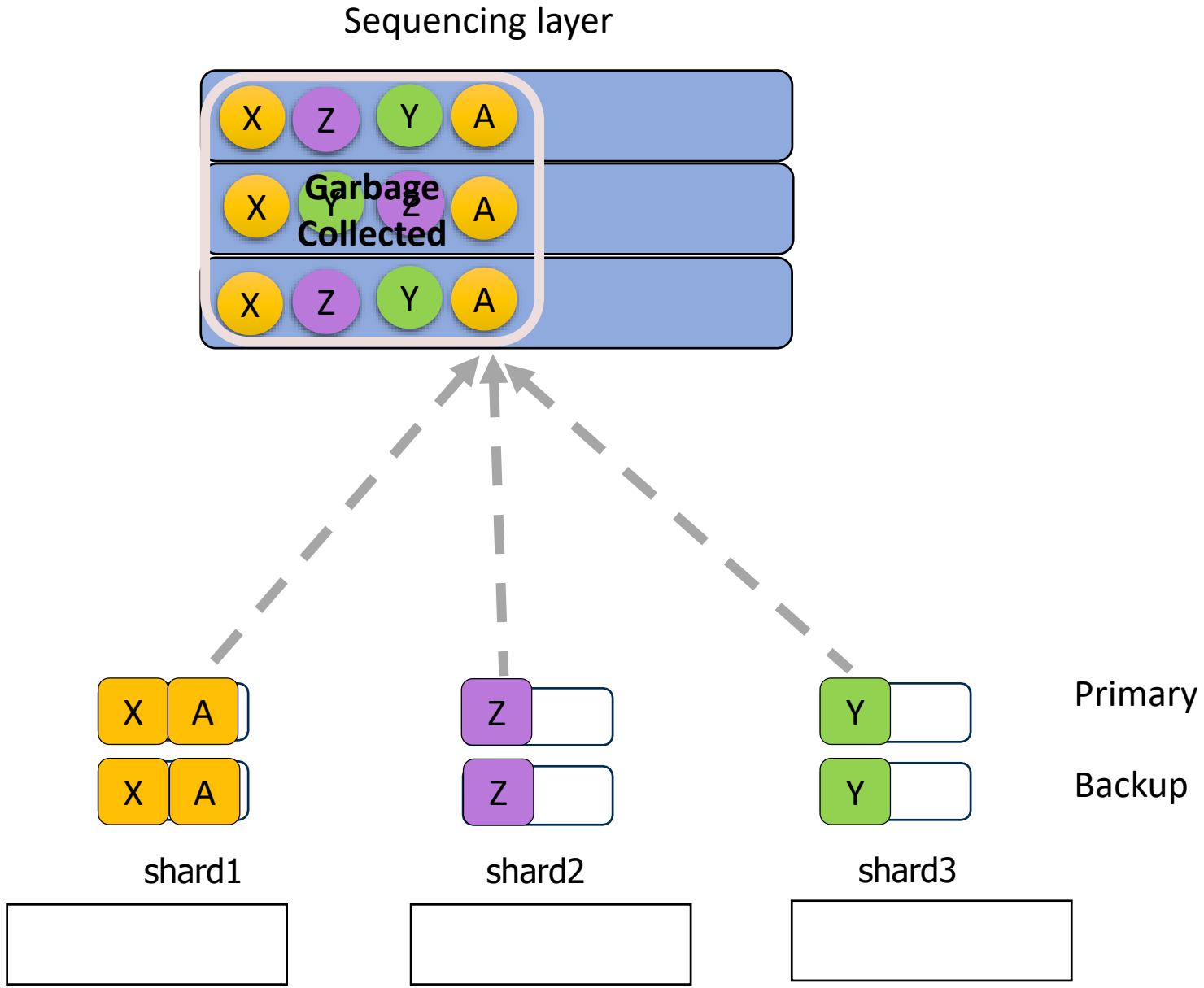
Unordered log

Erwin's Goal: 1-RTT Append

Data

Metadata

Clients



Failure Handling

Failure Handling

- Shard failure is handled within the shard- Paxos/Raft
- Network blips – clients can retry their append requests
- Sequencing layer replica failure- handled using **view and reconfiguration**

Step 1: Failure detection

- Using a control plane which has a zookeeper and a controller
- Every replica in the sequencing layer maintains heartbeat with the zookeeper and if it breaks then it informs the controller about the failed replica

Step 2: Start a new view

- Controller now seals the old **view**
- Any record sent with old view won't be appended
- Any of the other replica is assigned as recovery replica
- Flush records to shards starting from last-ordered-gp+1

Failure Handling Erwin St

- Sequencing replica failure
- Shard Failure

Same as Erwin Blackbox

Failure Handling

- Sequencing replica failure
- Shard Failure

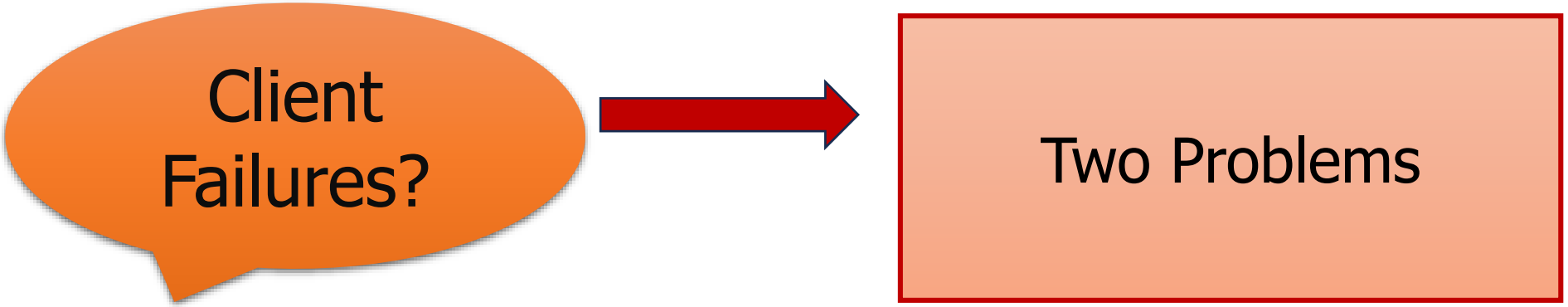
Same as Erwin Blackbox

Client Failures?

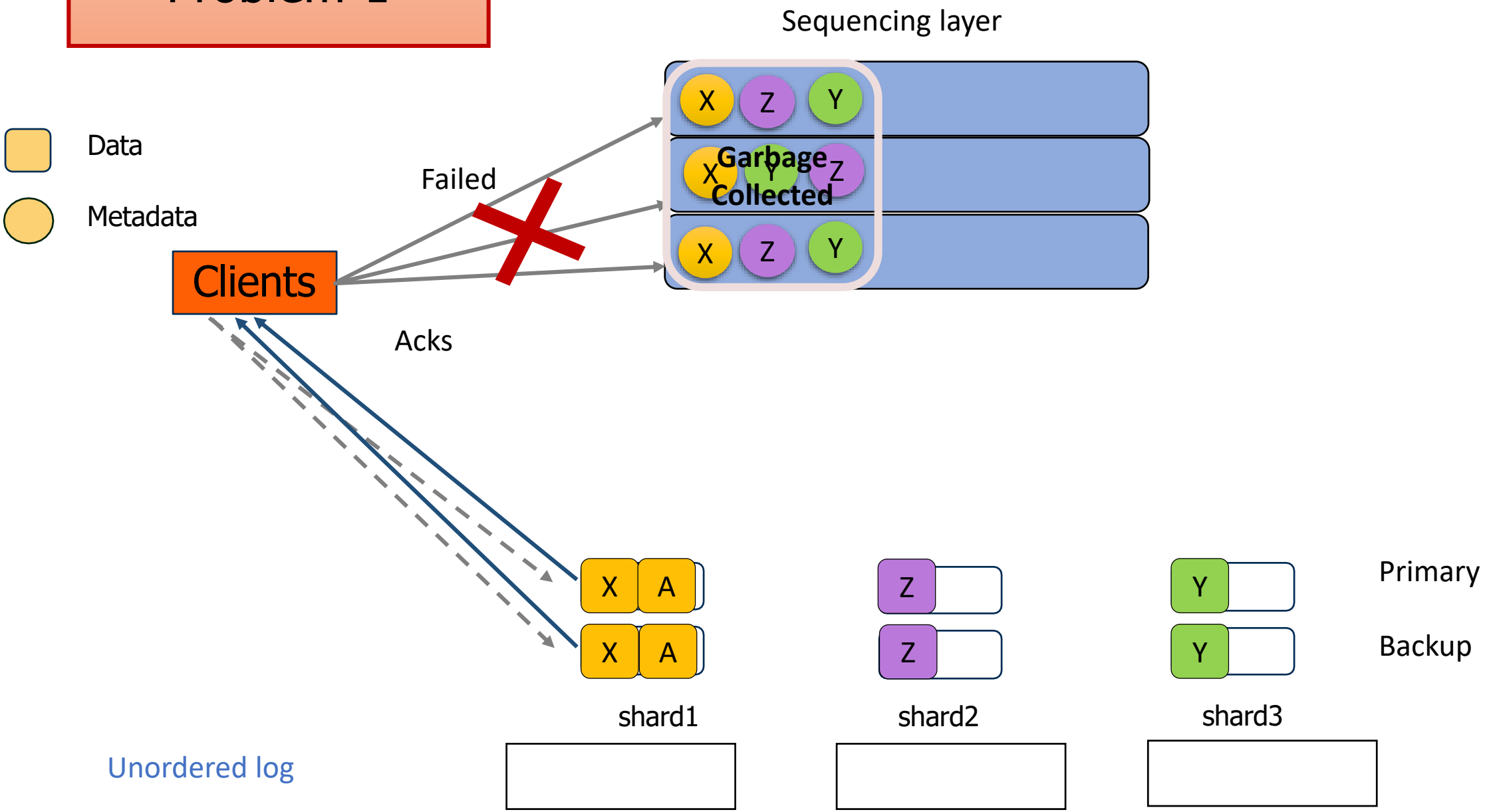
Failure Handling

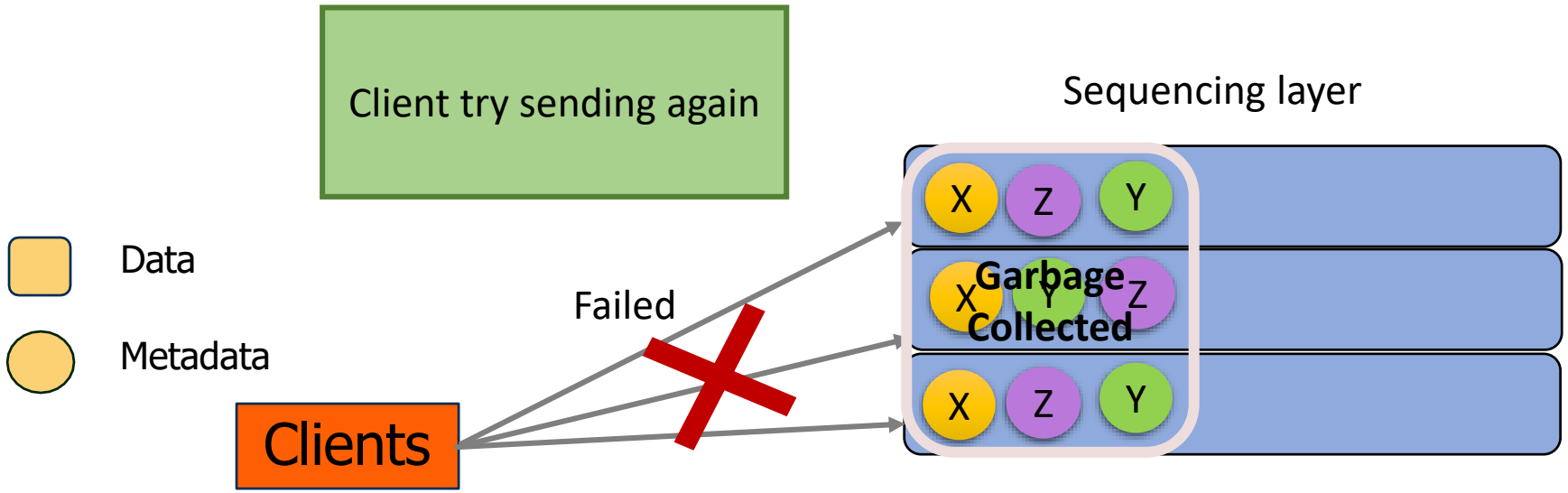
- Sequencing replica failure
- Shard Failure

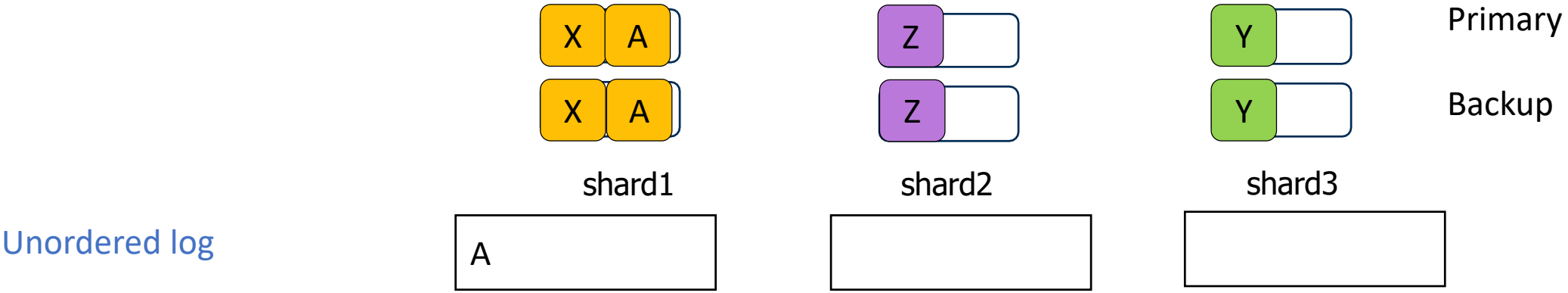
Same as Erwin Blackbox



Problem-1


Sequencing layer
Data
Metadata
Failed
Clients
X
Z
Y
Garbage Collected
Acks
X
A
Primary
Backup
shard1
shard2
shard3
Unordered log


Client try sending again
Sequencing layer
Data
Metadata
Failed
Clients
X
Z
Y
X
Y
Z
Garbage Collected
X
Z
Y

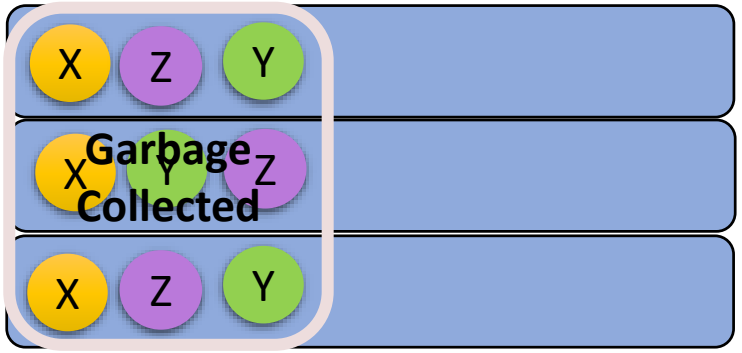

X
A
X
A
Z
Z
Y
Y
Primary
Backup
shard1
shard2
shard3
Unordered log
A

Data

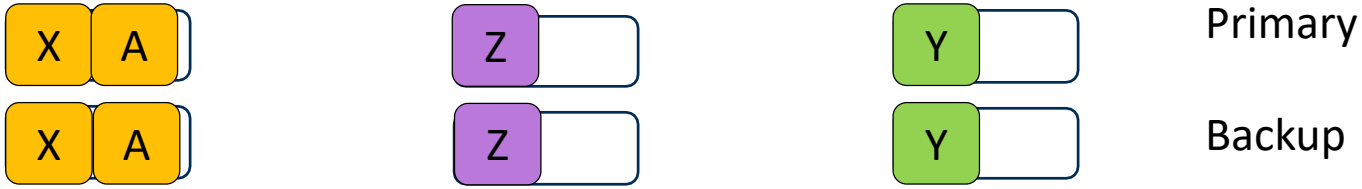
Metadata

Clients

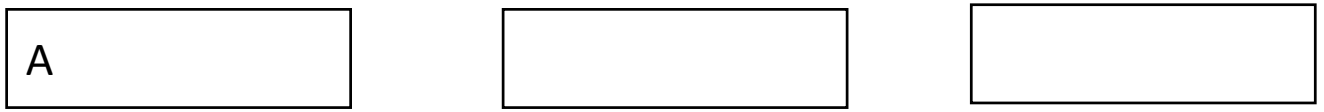
Sequencing layer



Data is Orphaned



Unordered log

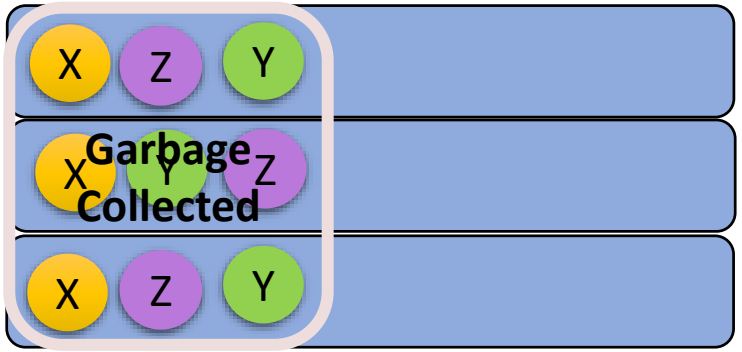


Data

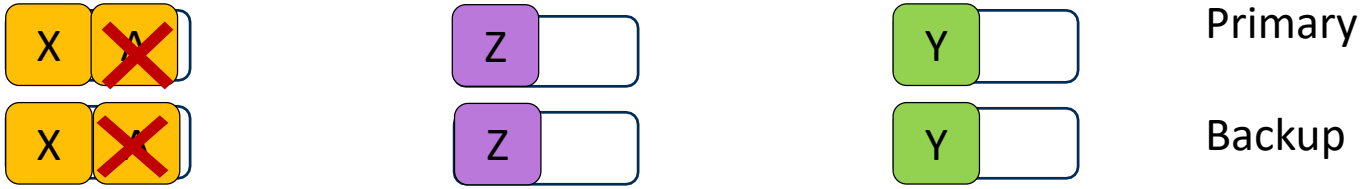
Metadata

Clients

Sequencing layer

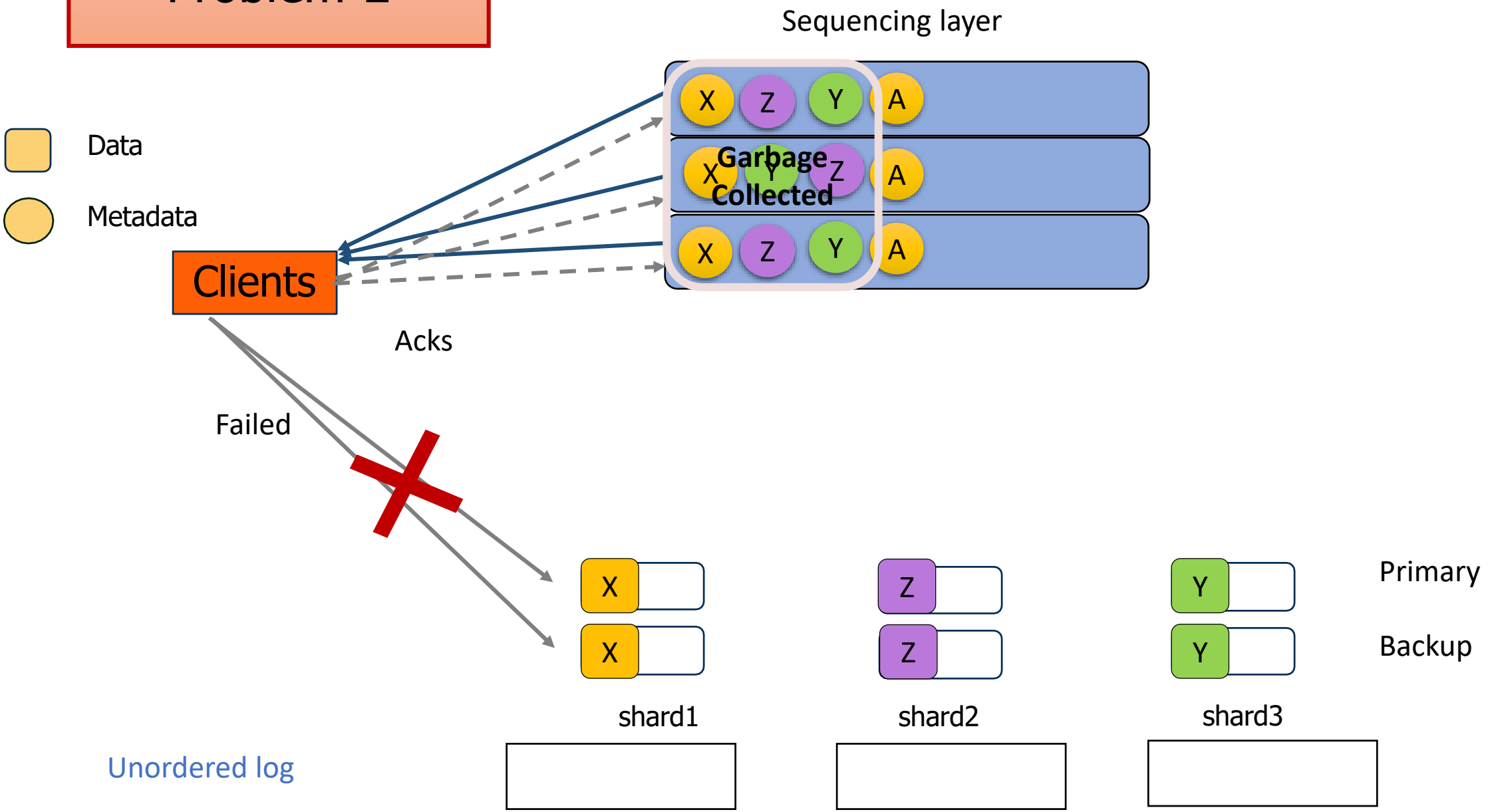


Data is Orphaned



Unordered log

Problem-2

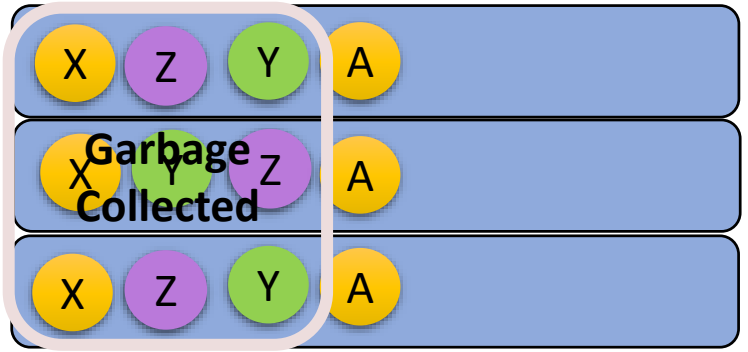

Sequencing layer
Data
Metadata
Clients
X
Z
Y
A
Garbage Collected
Acks
Failed
Primary
Backup
shard1
shard2
shard3
Unordered log

Problem-2


Data


Metadata

Sequencing layer

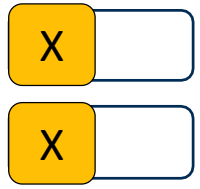


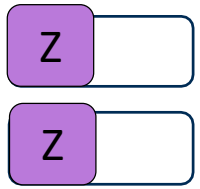
Clients

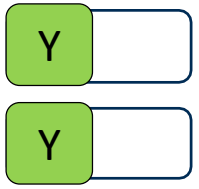
Failed

Client try sending again

Unordered log







Primary

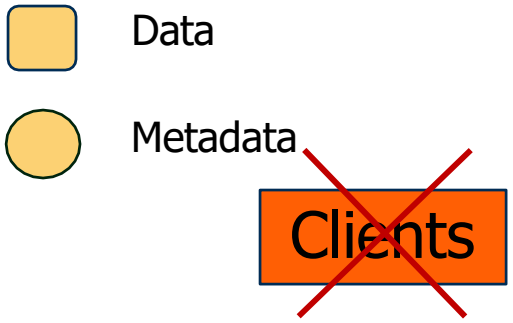
Backup

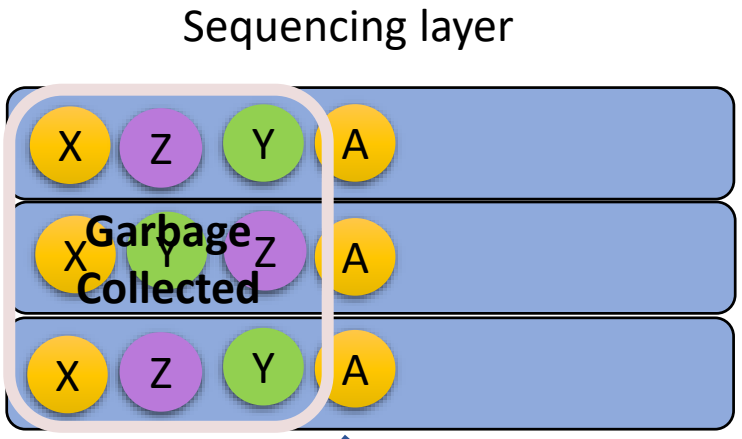
shard1

shard2

shard3

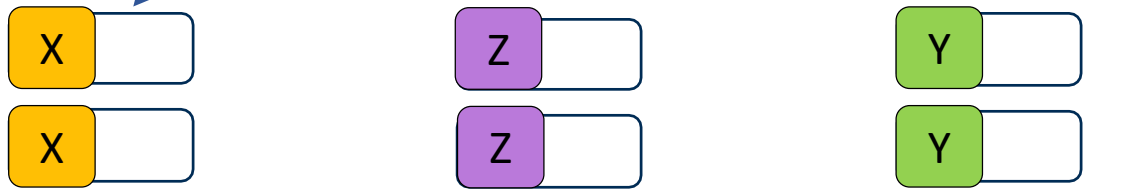
Problem-2





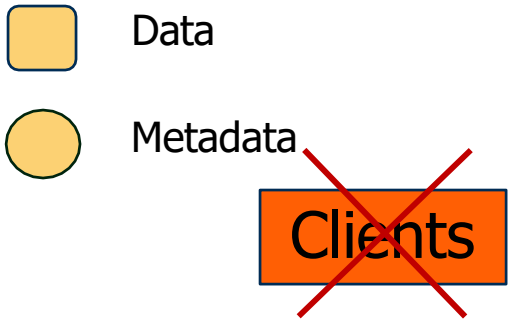
Sends position of the record

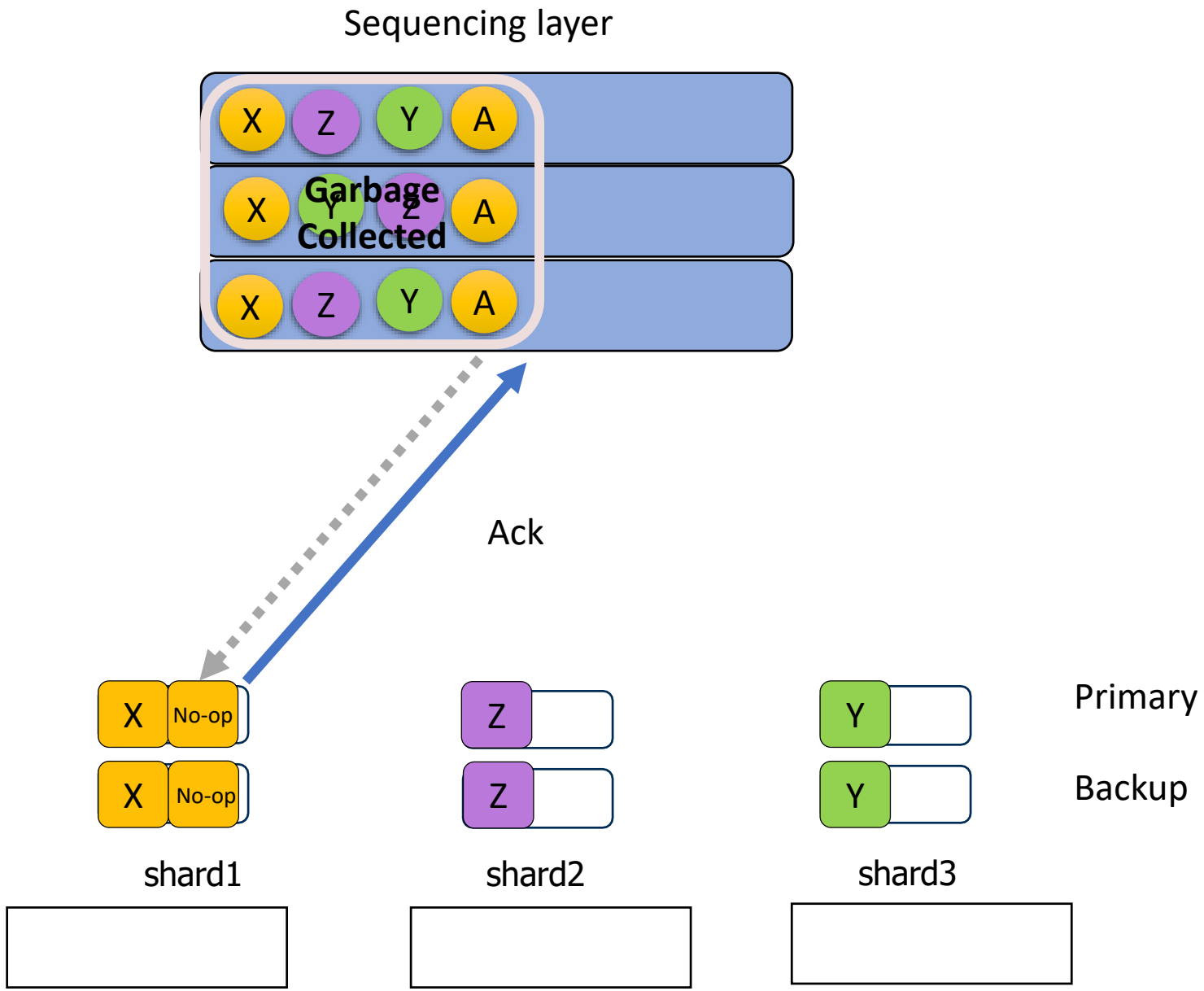
No Data with record_id



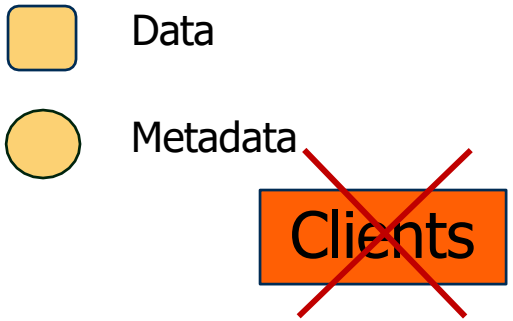
Unordered log

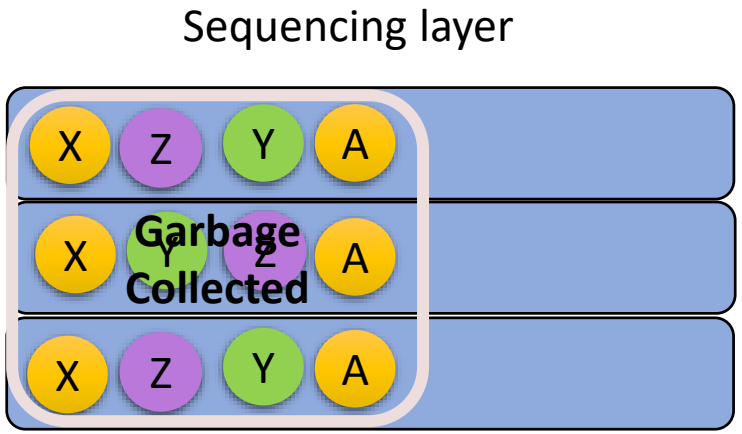
Problem-2

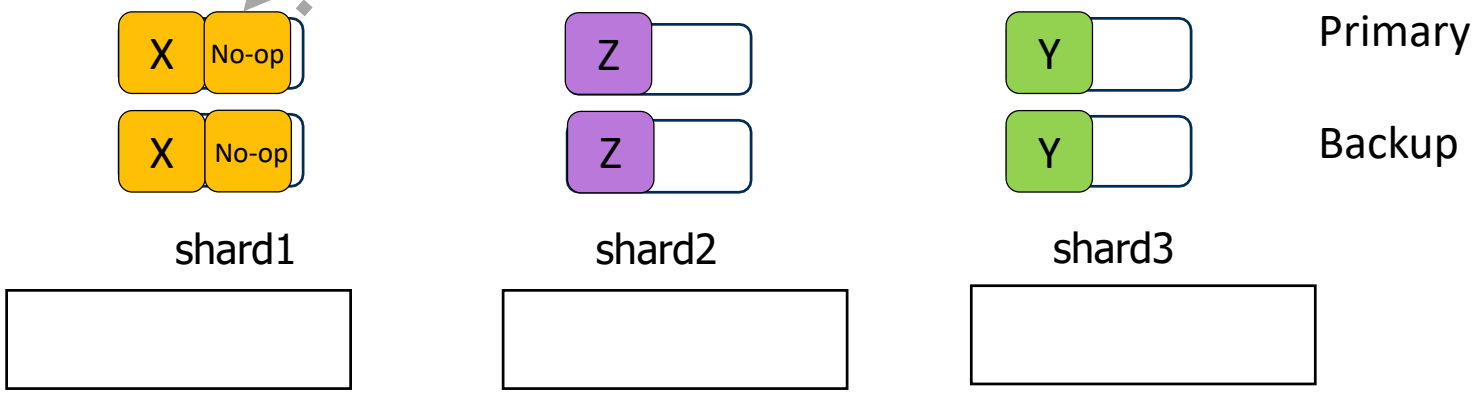

Data
Metadata
Clients


Sequencing layer
X
Z
Y
A
X
Garbage Collected
A
X
Z
Y
A
Ack
X
No-op
X
No-op
Z
Z
Y
Y
Primary
Backup
shard1
shard2
shard3
Unordered log

Problem-2


Data
Metadata
Clients


Sequencing layer
X
Z
Y
A
X
Garbage Collected
A
X
Z
Y
A


X
No-op
X
No-op
Z
Z
Y
Y
Primary
Backup
shard1
shard2
shard3
Unordered log

Performance Evaluation

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

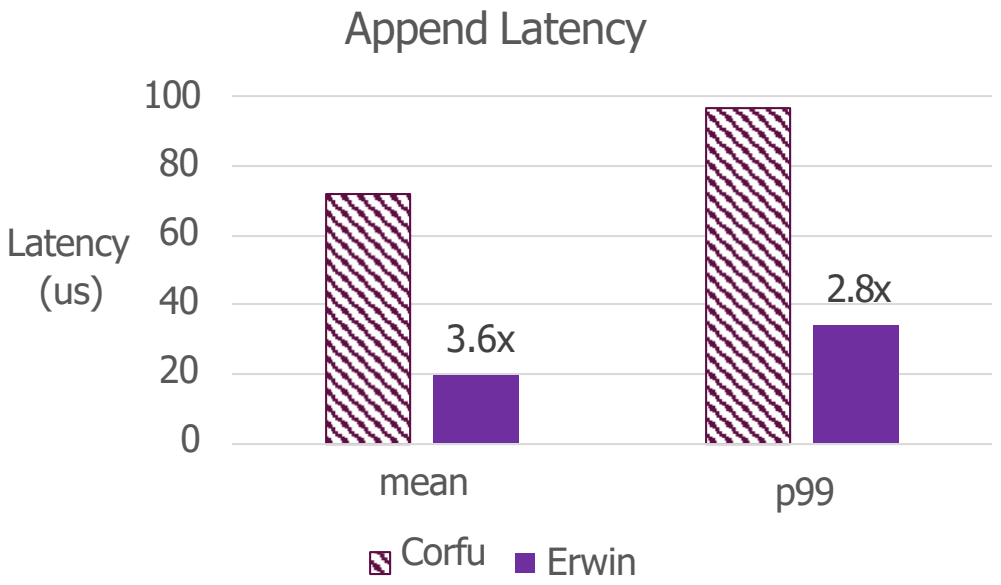
What's the Latency Benefit of Lazy Ordering?

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?

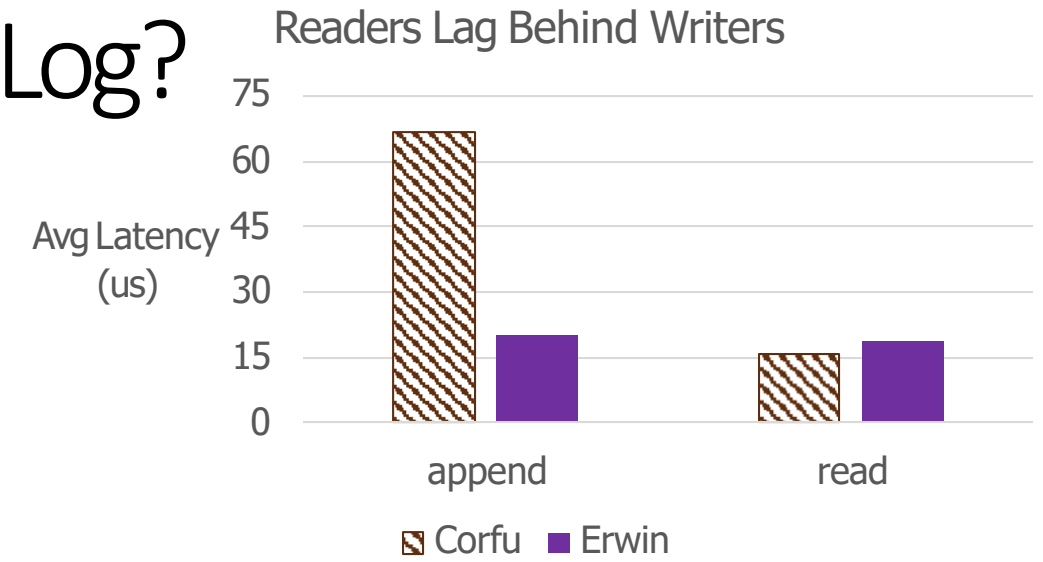
4KB record read after append

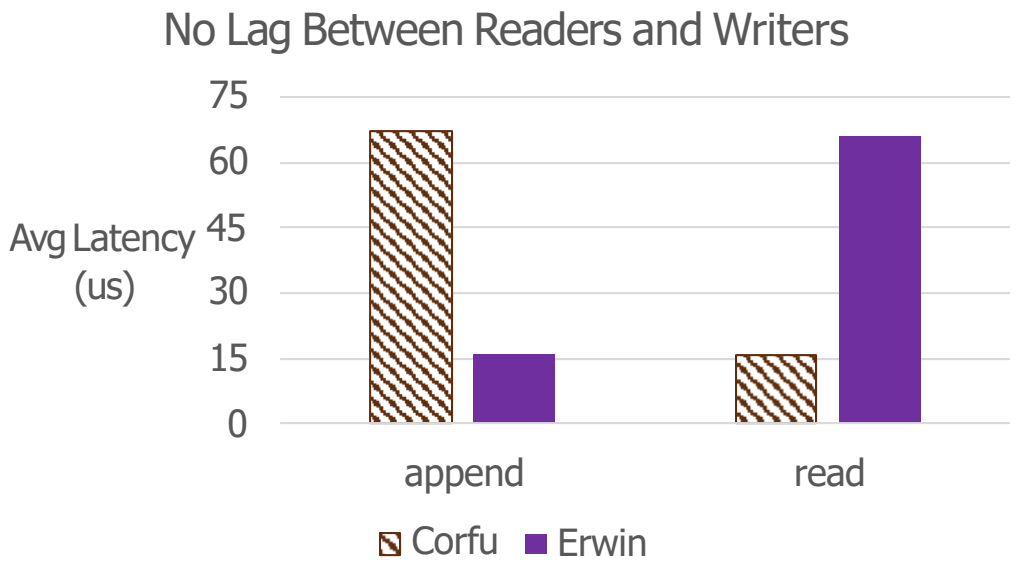
For many applications in which reads lag behind writes:

- 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?

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

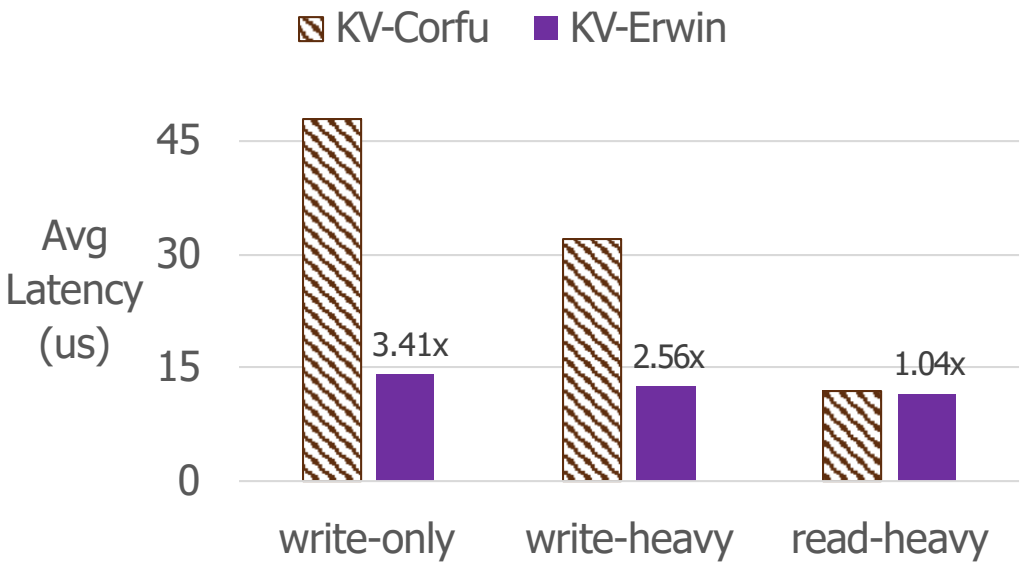
KV Store (decoupled WR-er and RD-er):

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



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