Logging & Data Durability

Steve Rumble
Stanford University

April 1, 2010

Outline

- **Need for Durable, Non-volatile Storage**
- **Buffered Logging**
- **Log-structured Memory**
- **Locating Log Objects**
- **Data Replication** **(for durability)**
- **Spreading Writes** **(for performance)**
- **Performance and Buffering Expectations**

- ***Next talk will discuss recovery from durable storage***

Data Durability

- **We need durability**
  - Servers will fail
  - The power will go out
  - Failures will increase in frequency as we scale
    - Assume they're common, deal with them quickly
  - Murphy's Law is out to get us

- **We need to replicate main memory contents**
  - Can't use RAM
    - Assumes we can keep RAM powered
    - Too expensive: increase cost/decrease capacity by replication factor

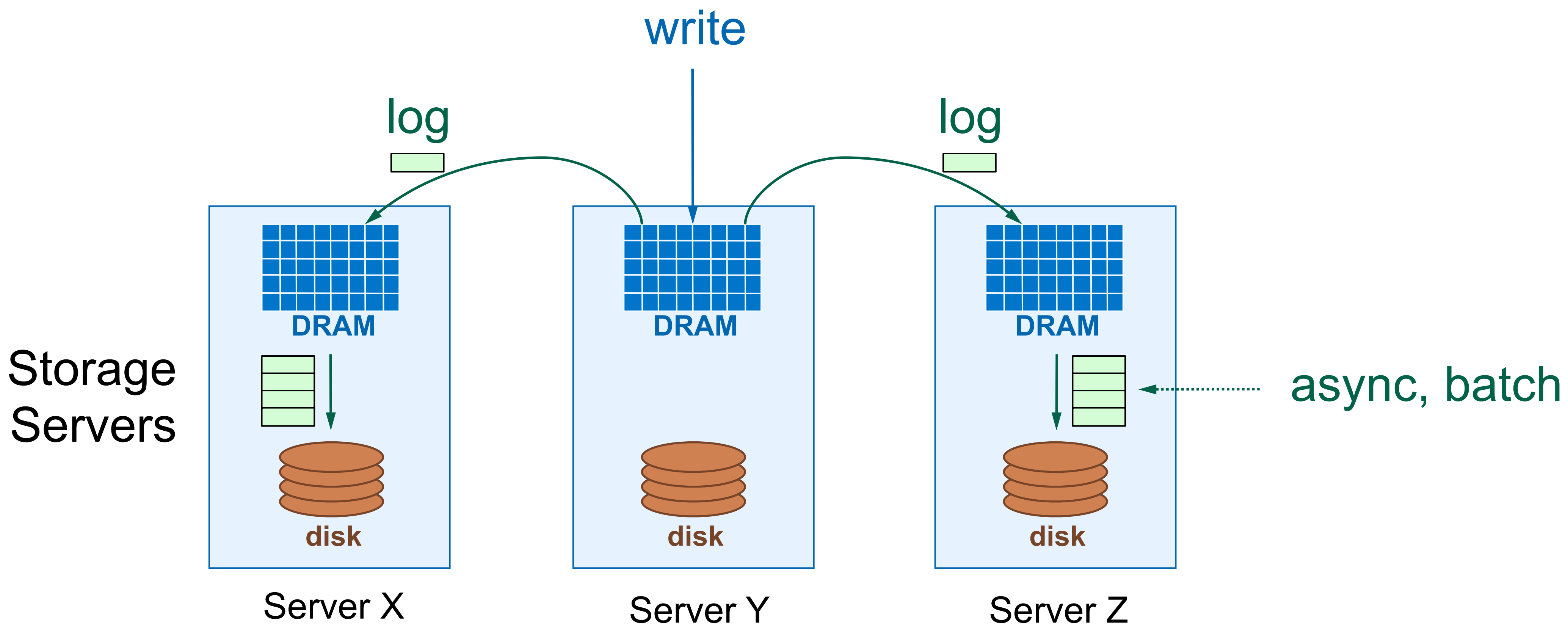
  - Can't use local disk
    - Too slow to recover
    - What if the box dies?

Cluster Approach to Durability

- **Problem: Make writes sufficiently durable while:**
  - Not horribly affecting latency
  - Not artificially limiting aggregate write bandwidth
- **Guiding Principles**
  - All backup devices favour sequential I/O
    - Buffer writes
  - All backup devices have significantly higher latency
    - Buffer and asynchronously commit
  - We are assuming lots (10s to 1,000s) of servers at our disposal
    - Buffer on other servers
- **Our Solution: Buffered Logging**
  - Each server logs updates in memory
  - RPCs return when log updates reflected in *k* backup memories
  - Backup servers asynchronously flush log updates to disk
  - Every master server is also a backup

Buffered Logging

- **All RAMCloud objects are logged**
  - Each server maintains one log (for now)
  - Log modifications synchronised with backups
    - Backups buffer fixed-sized pieces of the Log
    - One log/server, *k* replicas implies 2*k* buffers per backup
      - *k* replicas/master, double buffering for write & flush



Log-Structured Memory

- **Problem: Server must keep track of the Log**
  - E.g., need to do cleaning

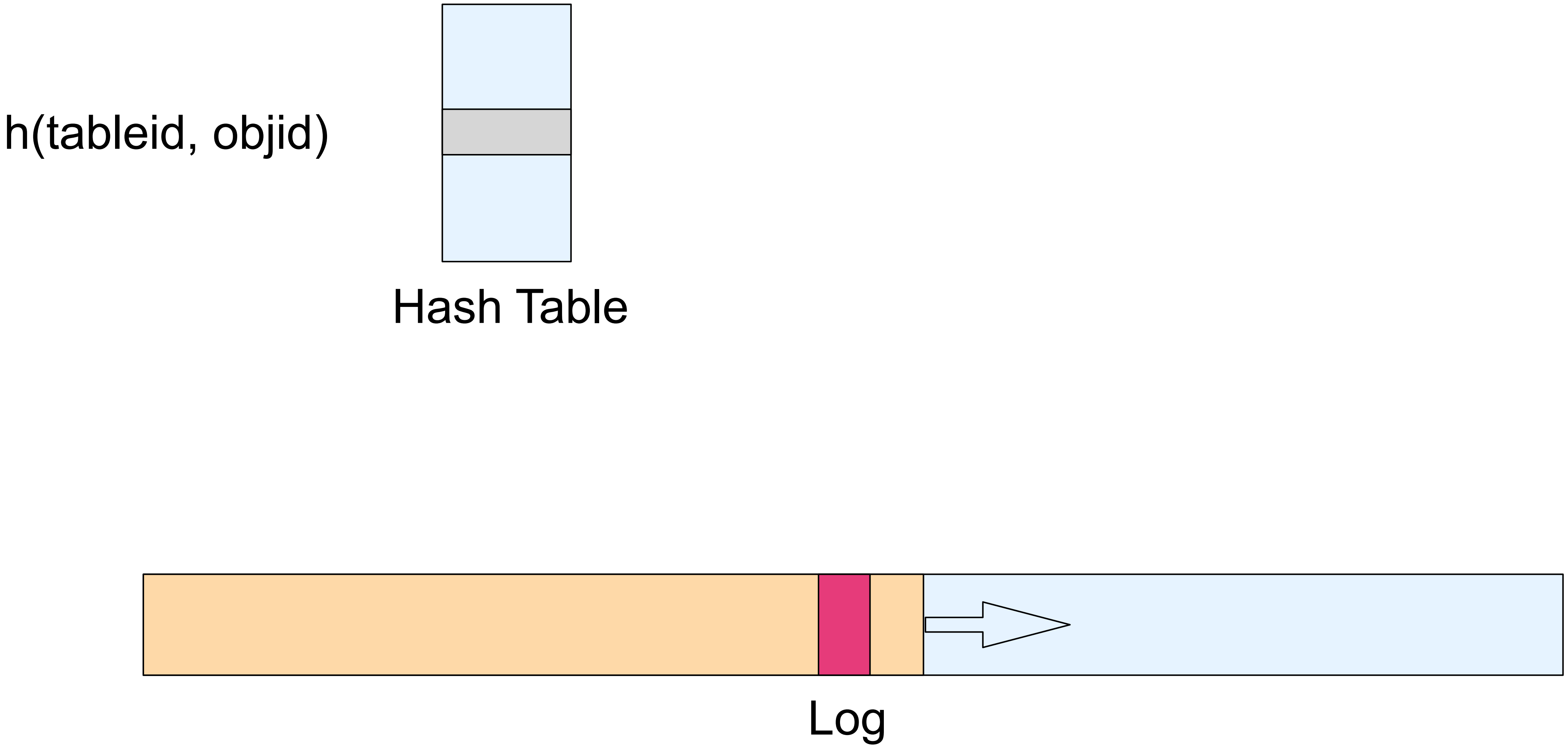
- **Solution: Make server memory log-structured**
  - Memory layout matches disk layout
  - Simplicity Benefit: Unify disk-based storage and memory allocation
    - Handle RAM fragmentation while boosting write rates

- **Drawbacks**
  - Couples disk utilisation with memory

- **Log-structured Memory is LFS to the extreme**
  - RAM “caches” everything
  - Disks are read only on failure
  - Cleaning requires no disk reads! (avoids 1/2 of cleaning overhead)

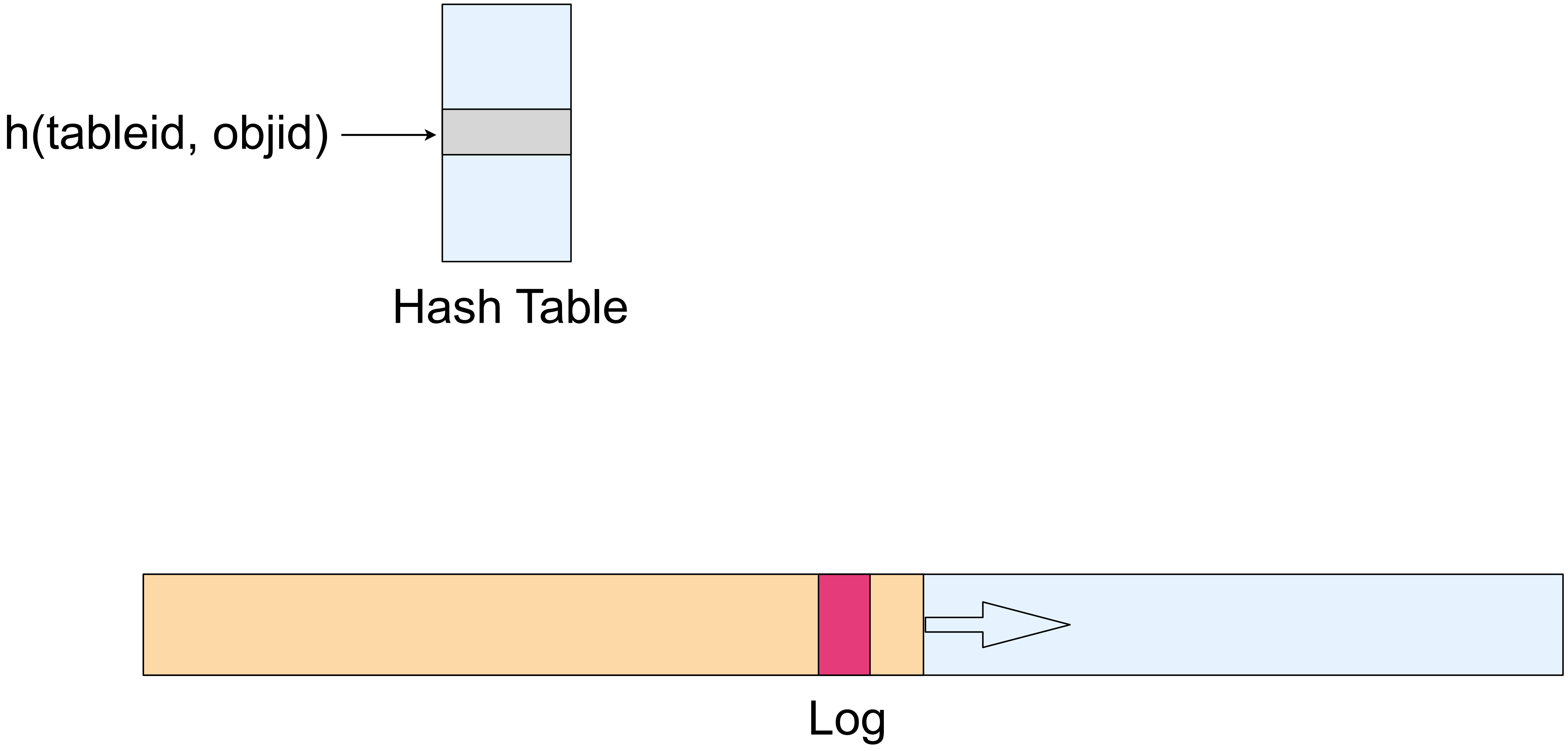
Locating Objects in the Log

- **How do we find objects in the main-memory Log?**
  - Hash table lookup
  - Two cache misses from (tableId, objectId) to object



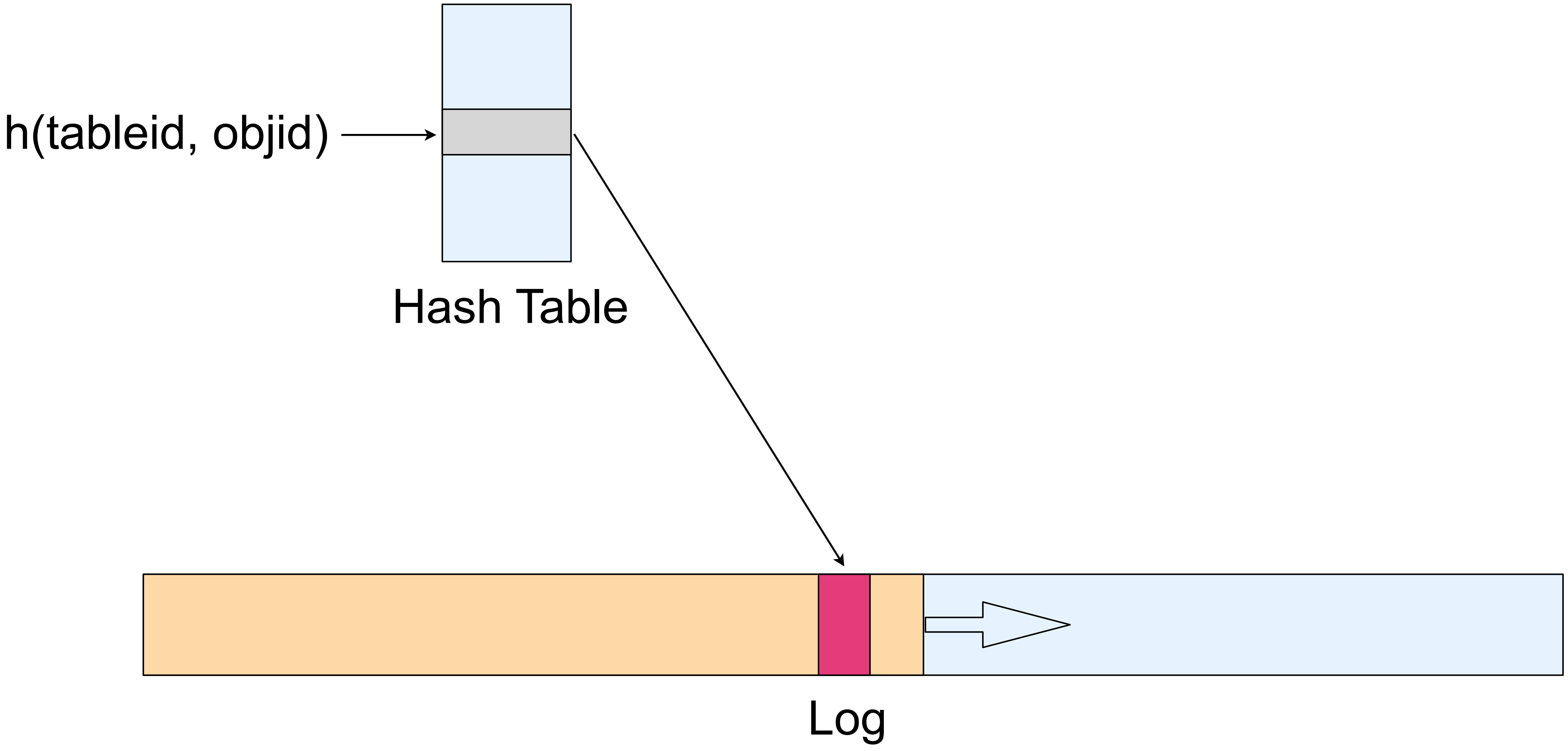
Locating Objects in the Log

- **How do we find objects in the main-memory Log?**
  - Hash table lookup
  - Two cache misses from (tableId, objectId) to object



Locating Objects in the Log

- **How do we find objects in the main-memory Log?**
  - Hash table lookup
  - Two cache misses from (tableId, objectId) to object



Scattering Writes

- **Problem: Need fast recovery**
  - Writing log updates to same *k* backups not good enough
  - k * 100MB/s I/O bandwidth insufficient for quick recovery

- **Solution: Scatter log across many spindles**
  - Don't fix the *k* backup hosts for each master
  - Fill buffers on *k* backups, then move on
  - < 1 second to read 64GB from 1,000 disks

- **For each new segment, choose a new set of *k***
  - Find additional backups with idle bandwidth
  - Can accommodate more writes immediately
  - Example mechanism:
    - Cache potential backup lists (obtained from cluster coordinator)
    - Choose 2*k* of them as potentials and query to find the best k

Scattering Writes, cont’d

- **Scattering writes is like statistical multiplexing**
  - Lets us supports large write bursts
  - Servers make use of idle disk bandwidth throughout cluster
    - Recall the full bisection bandwidth assumption
- **Consider 1,000 node cluster:**
  - One 100MB/s disk per node
  - Even with overheads, aggregate bandwidth in 10’s of GB/s range
  - Best case well above 10GigE rates & our single server goals
- **But what about worst case performance?**
  - Worst case: congestion at all backups
    - Every node bound by backup disk I/O

Expected Sustained Write Rate

- **Only 2.5% of the read rate!**
  - About 25,000 1KB objects/server/second

- **Why? Write overheads:**
  - Log cleaning
  - *k* replicas (every server is a backup)
    - At best 1/*k*'th as much throughput as reads

- **k = 2, log cleaning overhead 100%**
  - 100MBs / 2k = 25MB/sec for writes
  - 25,000 1KB writes/sec
  - Recall per-server read estimate: 1M 1KB reads/sec
    - 1,000,000/25,000 = 2.5%

Boosting Writes

- **2.5% is conservative**

- **We can:**
  - Compress log buffers before flushing
  - Add disks
  - Do pre-flush cleaning (only flush live data)

- **3 disks/server, 2x compression, less cleaner overhead**
  - > 15% of read rate seems reasonable
  - Flash SSDs would add another 2-3x today

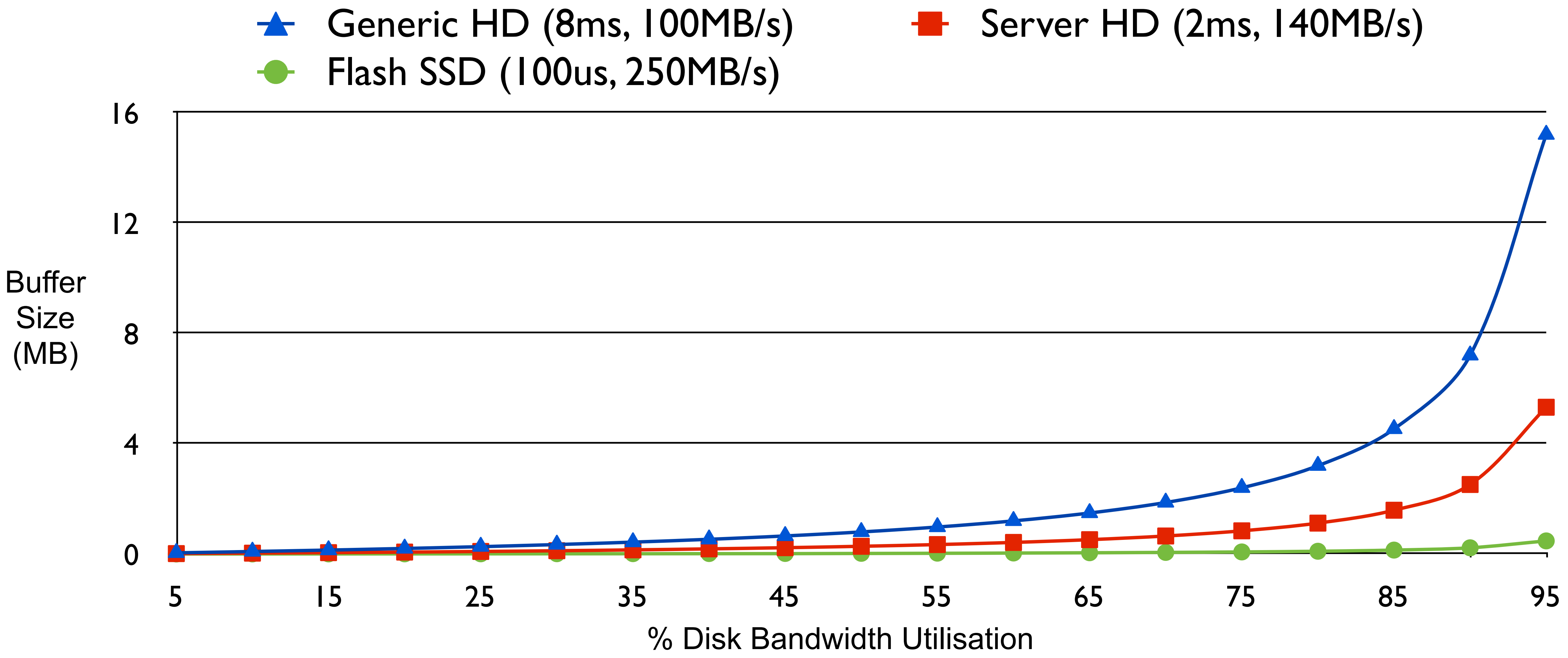
- **Need modest capacity devices with high bandwidth**
  - Prefer cheap bandwidth over cheap capacity
  - Latency less important

How Big are the Buffers?

- **Amortising overhead means sizing buffers properly**
  - Example: Want 90% utilisation
    - Assume average overhead of 8ms per operation (seek + 1/2 rotation)
    - 90% bandwidth => 72ms of data access for every op
      - 72ms at 100MB/s => 7.2MB per op
  - Generalised:
    - $\text{Buffering\_Needed} = \text{Latency} \times \dfrac{\text{Utilisation}}{1 - \text{Utilisation}} \times \text{Disk\_Bandwidth}$

How Big are the Buffers? (cont'd)

- **Buffering needed as function of desired utilisation**



  - ~7-8MB buffers means:
    - 90% utilisation with 8ms, 100MB/s HD
    - 96% utilisation with 2ms, 140MB/s 15K rpm server HD
    - 99.7% utilisation with 100us, 250MB/s flash

What About Flash?

- **Latency too high for primary store, but what about backup?**

- **Could be promising**
  - Flash is currently modest-sized and high bandwidth

- **However**
  - SSDs are still very expensive
  - Performance expectations are complex

- **Our techniques should work well with flash**
  - Locality is still important, so buffered approach fits
  - And may obviate complex FTLs

Conclusion & Discussion

- **Conclusion**
  - RAMCloud uses Log-structured memory
    - Each server has a *Log*
    - Each log is backed up to *k* other servers' disks
    - All RAMCloud objects live in the Log
    - Buffering is crucial due to non-volatile storage properties:
      - Sequential I/O bias
      - Access latency
    - Memory structure matches disk structure
  - Logs are distributed across the cluster, enabling:
    - Fast recovery
    - High burst write rates
- **Possible Discussion Topics**
  - What disks are used in data centers today? Cheap IDE, SCSI?
  - How does flash perform now? What can we expect in the future?
  - Alternatives to logging? Will flash shortly obviate buffering?

End of the Line

- **Do not pass Go.**

Storing to Disk

- **How do backups store data on disk?**
  - I.e. how are main memory write buffers drained?

- **Considerations**
  - Disks become the system’s write rate bottleneck
  - Locality is crucial for performance
    - Need very fast writes to drain backup write buffers
    - Need efficient reads to achieve fast recovery
  - Need to play standard filesystem games
    - Amortise seeks and rotational latency => batched writes

LFS, Revisited

- **Make the main memory object store log-structured**
  - Objects live in the log at all times
- **What is a Log?**
  - A set of (1 + M + N) segments:
    - One “head”
      - appends go to it
    - M segments are free
      - future heads; no live data
    - N segments are in use
      - may contain live data
- **The Log simply:**
  - appends to the head segment (and synchronises with backups)
  - Tries to maintain free segments for future appends

Log Cleaning

- **Why cleaning?**
  - Modifications or deletions to objects supersede past log entries
  - Need to defrag non-live data to free contiguous space
  - 2TB disk fills in < 6 hours at 100MB/s
  - Want to bound recovery time

- **Cleaning isn’t free**
  - In traditional LFS:
    - Read in segments, pack live data, write out again
    - Goal: Efficiently reclaim contiguous regions of disk
      - Cost/benefit balance - carefully choose segments to clean
  - RAMCloud affords us a twist on the LFS story

Log Cleaning, cont’d

- **RAMCloud master maintains all data in RAM**
  - Segments can be cleaned without first reading from disk
  - Removes >= 1/2 of the cleaning cost

- **RAM-based Log**
  - Objects stored in log format in main memory
  - Precisely reflects log stored on disks
  - Backups simply synchronise segment writes

Finding a Balance

- **We need data durability, but *don't* want:**
  - to spend a lot of money
  - to sacrifice (too much) performance
  - recovery to take too long

- **So, what *do* we want?**
  - RPCs that modify data complete in near RAM speed
  - Reasonable sustained write rate for busy cluster
  - Very high burst write rates for loaded servers

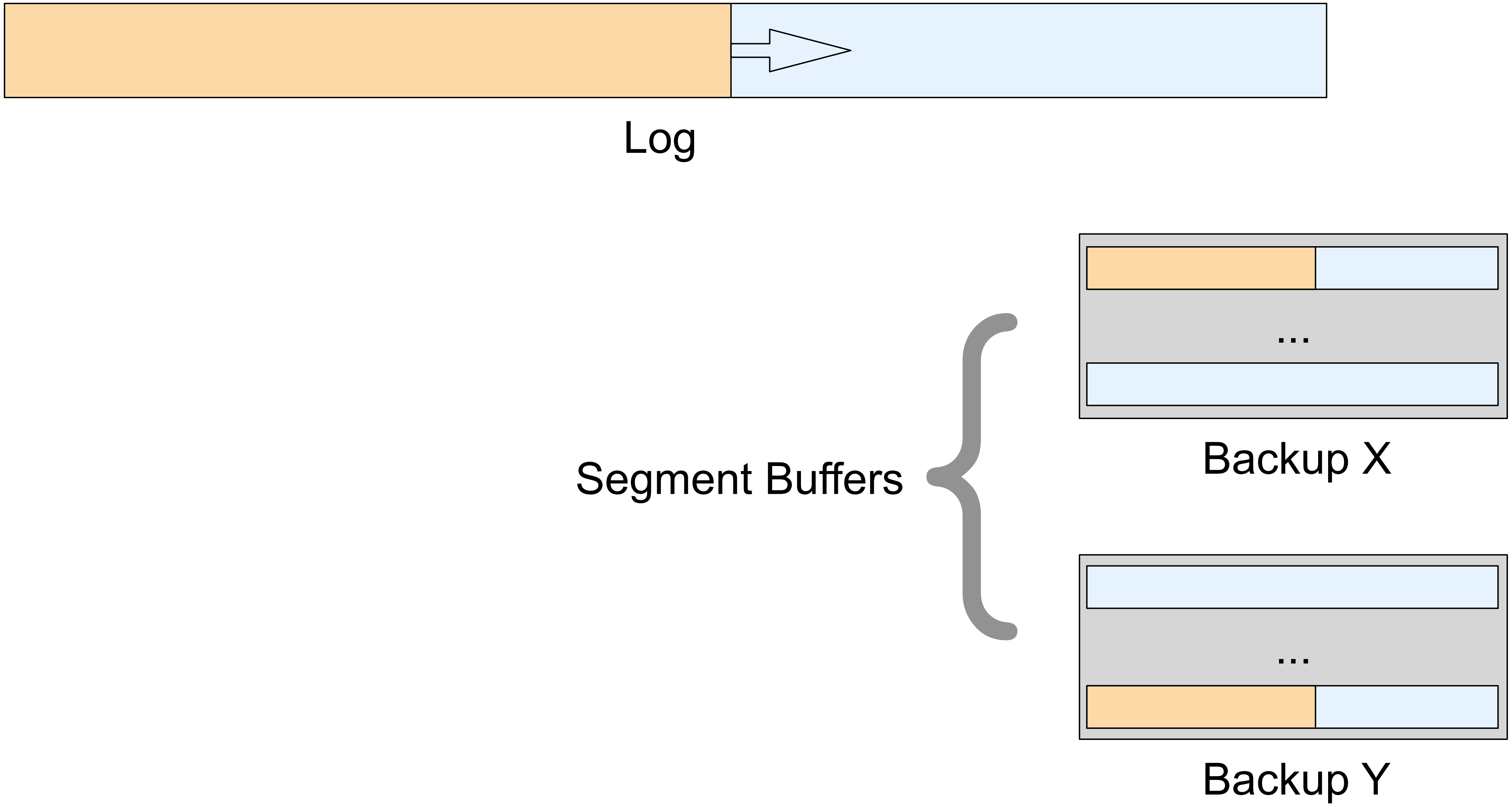
- **Logging falls out naturally**
  - All stable storage prefers sequential I/O

Statistical Multiplexing

- **Aggregate Disk Bandwidth**
  - N servers, K backups per object, 100MB/s writes (ignore cleaning)
  - 100N/K MB/s total bandwidth for object writes
  - N = 1,000, K = 3 => ~33GB/s
- **Worst case Performance**
  - 33MB/s/server (before log cleaning overheads)
    - 33,000 1K objects/second/server at 2 network RTTs
- **Best case well above 10GigE rates**
  - Servers should make use of idle bandwidth for fast write bursts
    - Spread log segments across all backups (benefit: helps recovery)

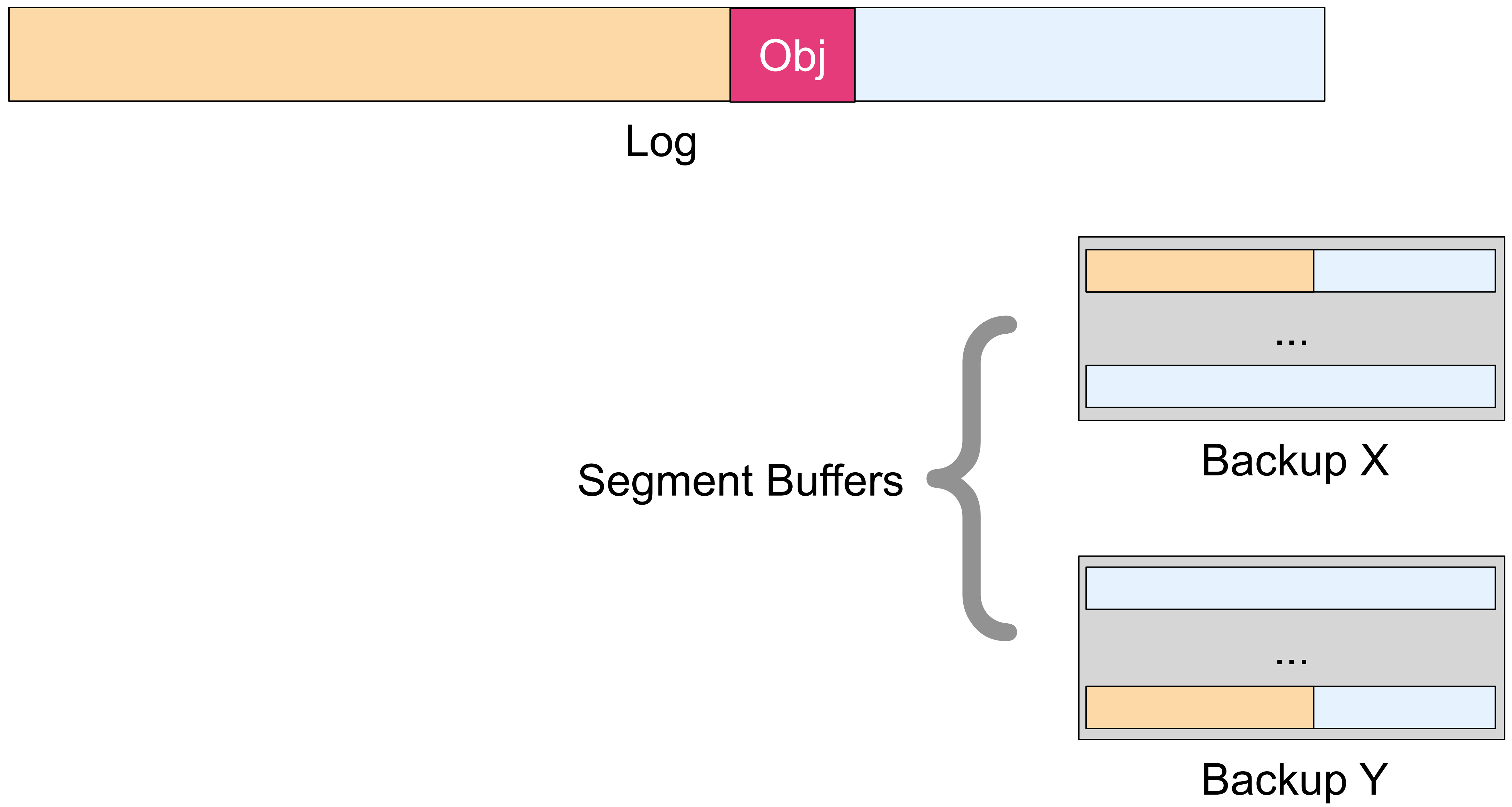
Servicing Requests, cont'd

- **Write** Obj
  - Append to log, distribute to backups, update hash table



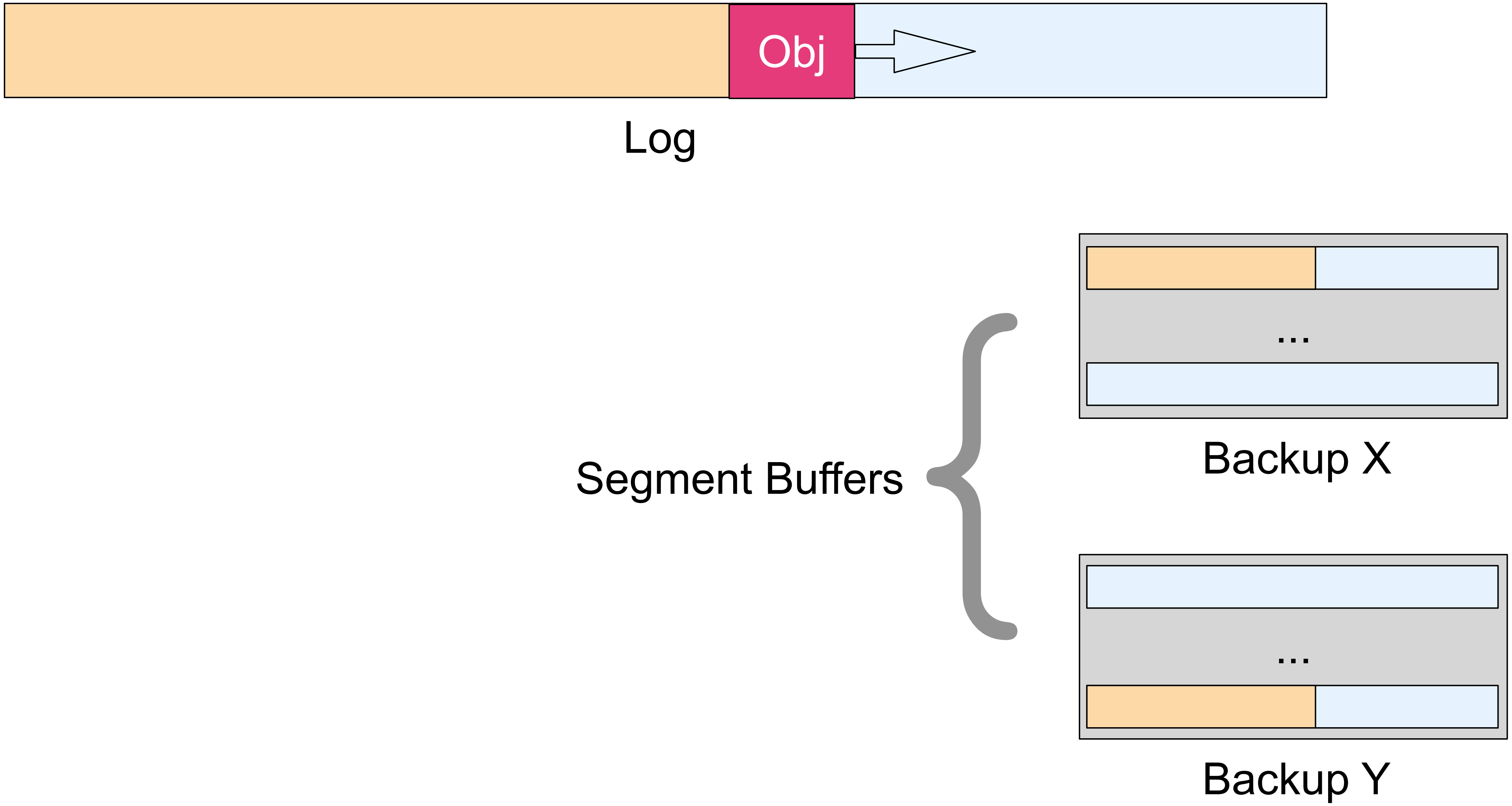
Servicing Requests, cont'd

- **Write**
  - Append to log, distribute to backups, update hash table



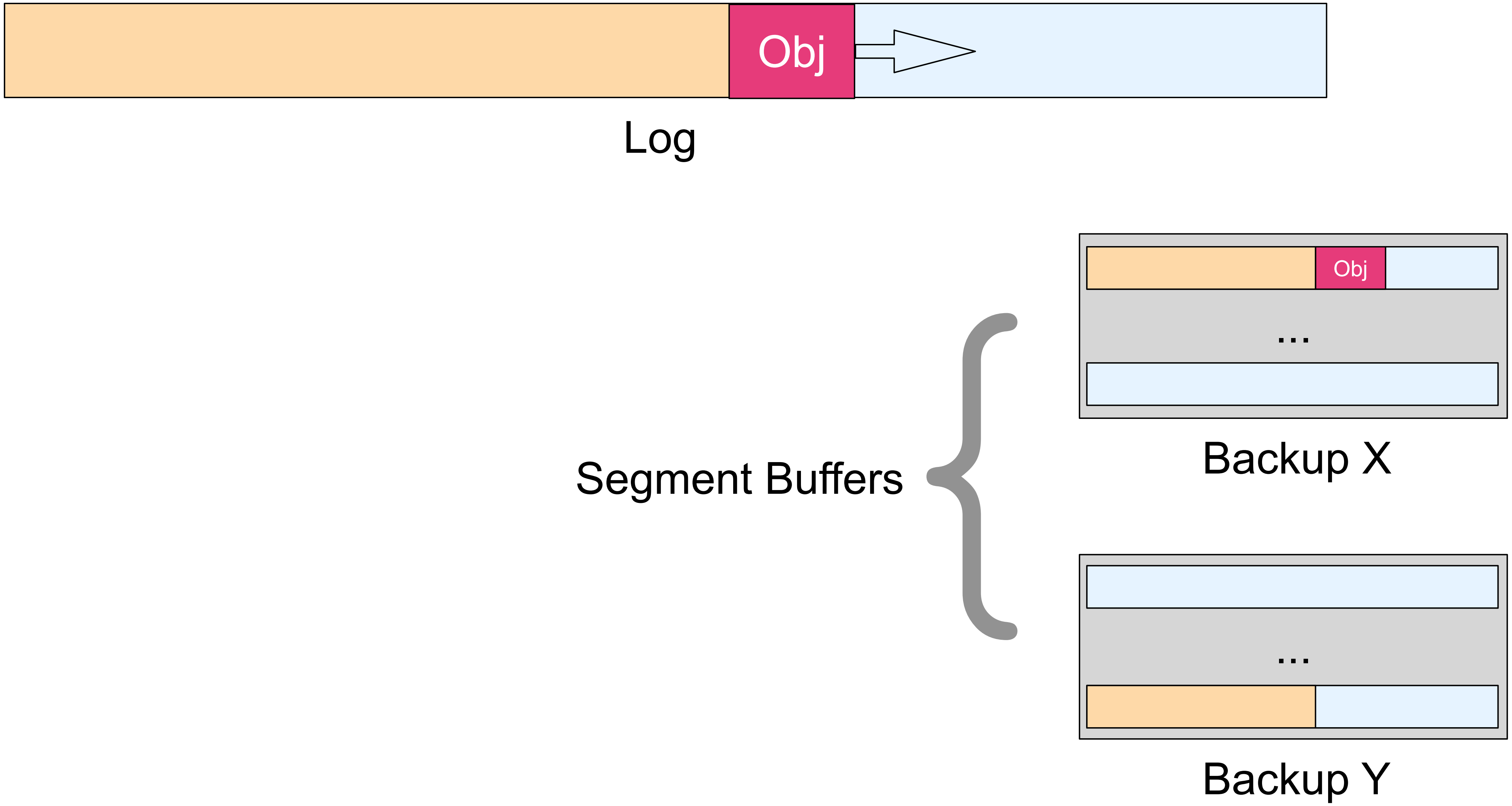
Servicing Requests, cont'd

- **Write**
  - Append to log, distribute to backups, update hash table



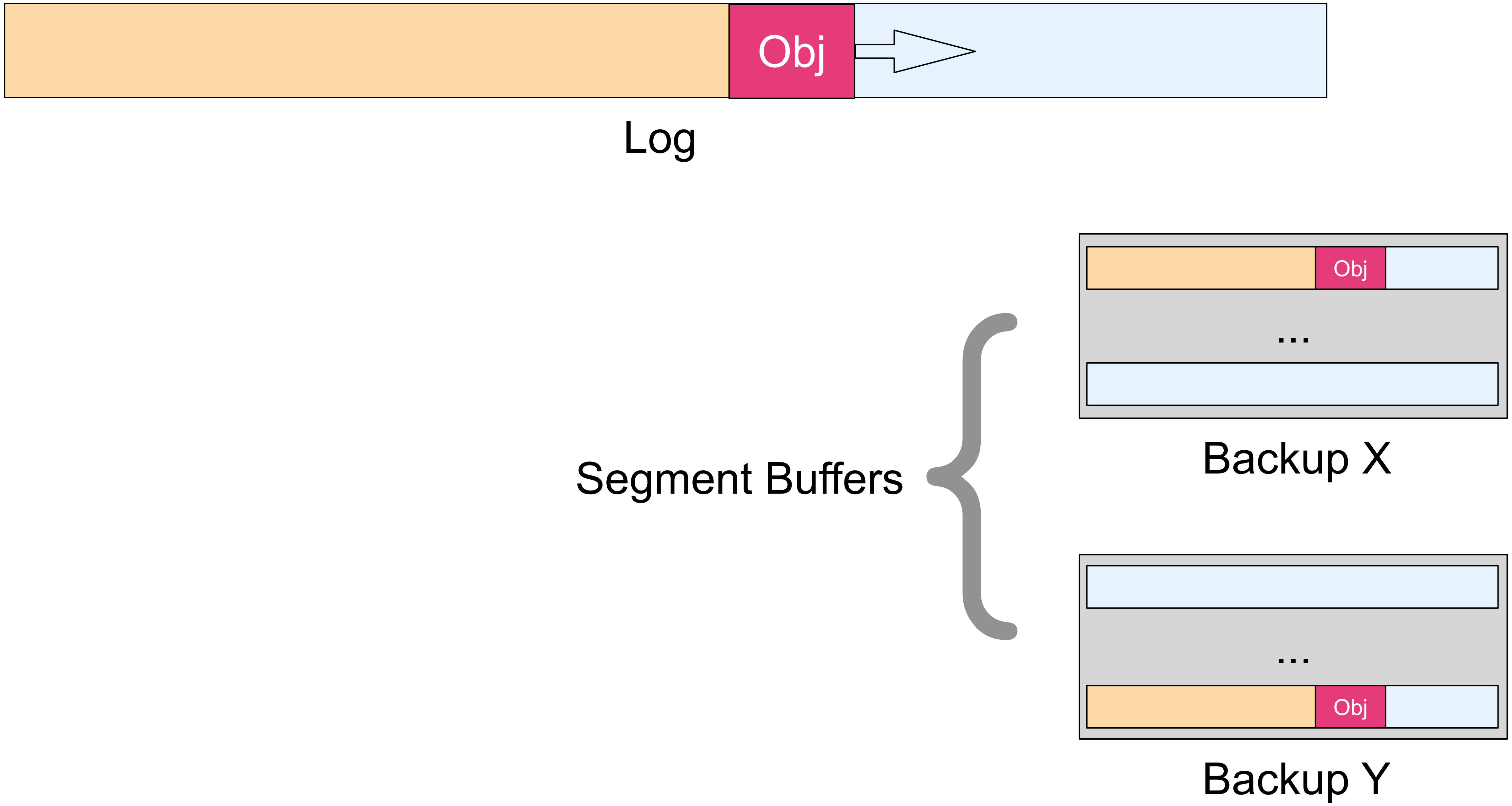
Servicing Requests, cont’d

- **Write**
  - Append to log, distribute to backups, update hash table



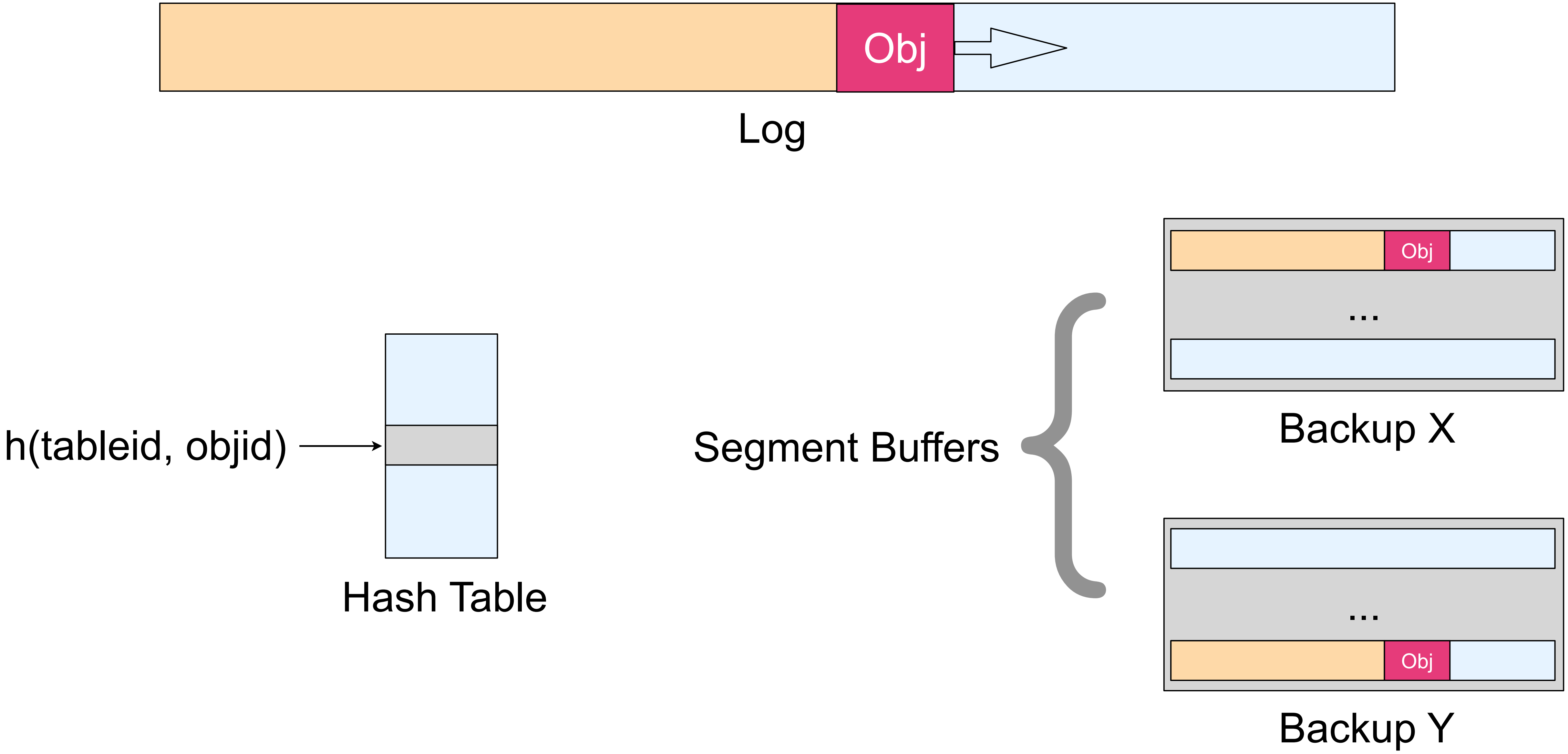
Servicing Requests, cont'd

- **Write**
  - Append to log, distribute to backups, update hash table



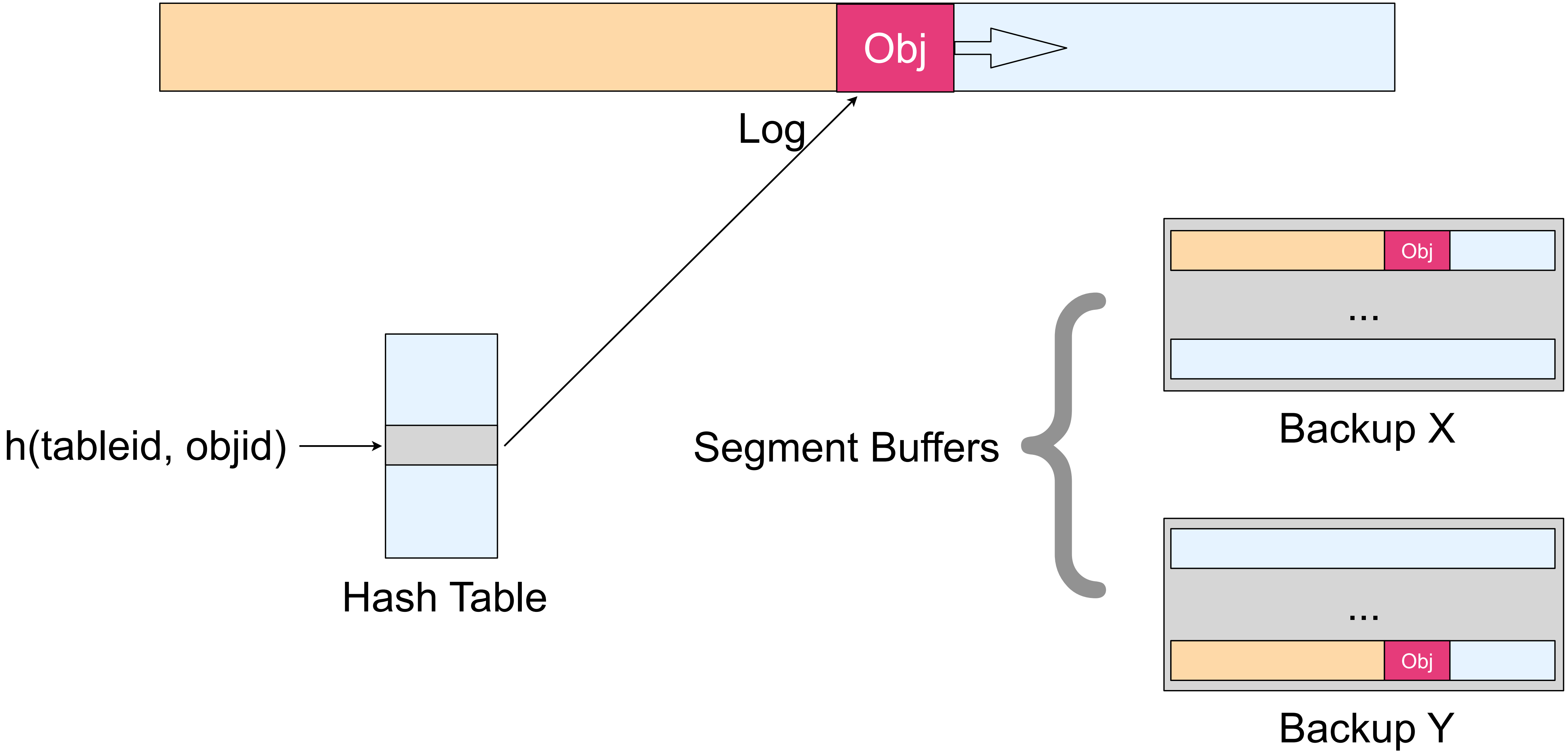
Servicing Requests, cont'd

- **Write**
  - Append to log, distribute to backups, update hash table



Servicing Requests, cont'd

- **Write**
  - Append to log, distribute to backups, update hash table

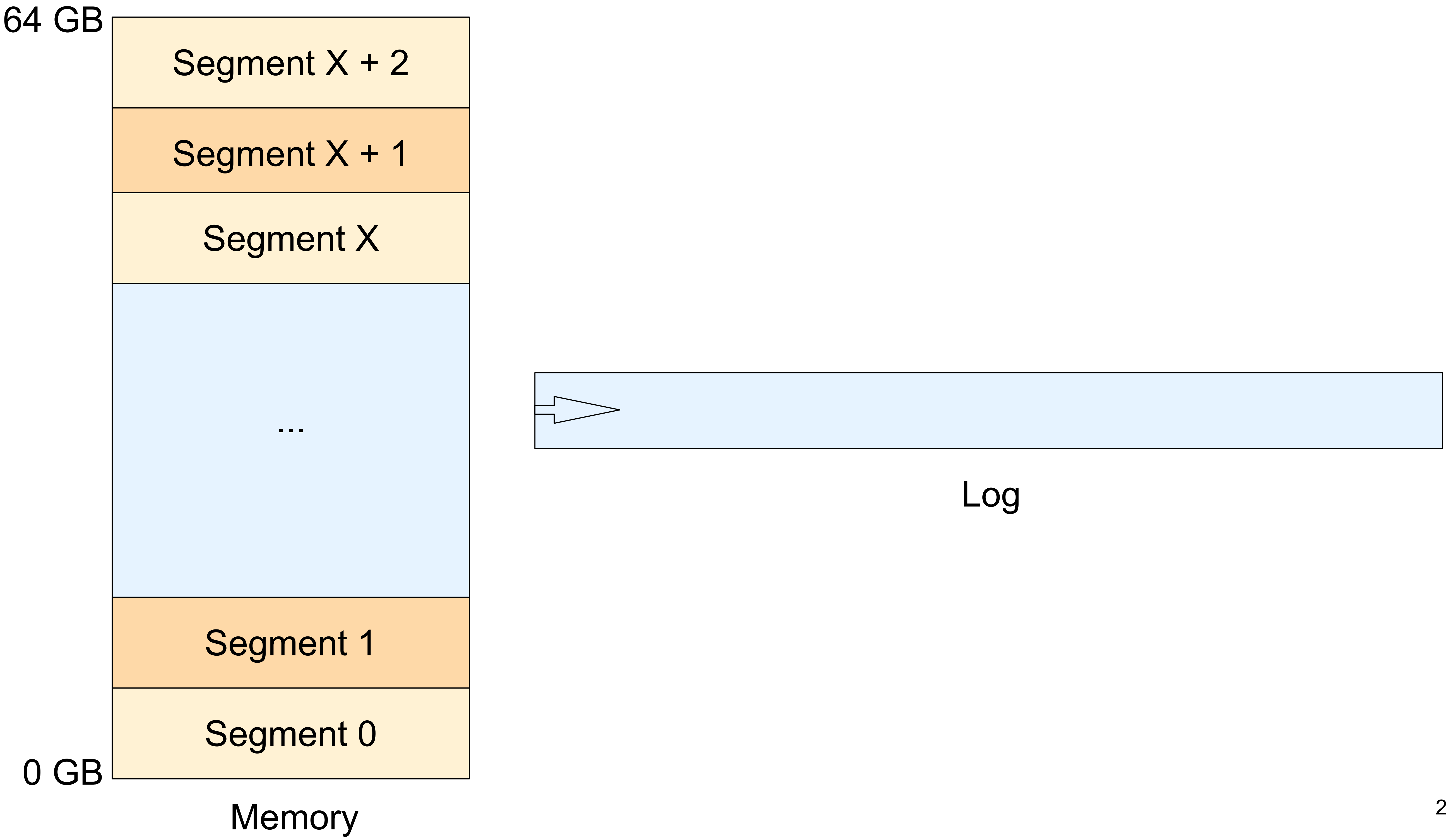


LFS: The Next Generation

- **LFS Premise:**
  - RAM = cheap read cache, so worry about writes
  - Make writes sequential for maximum write I/O

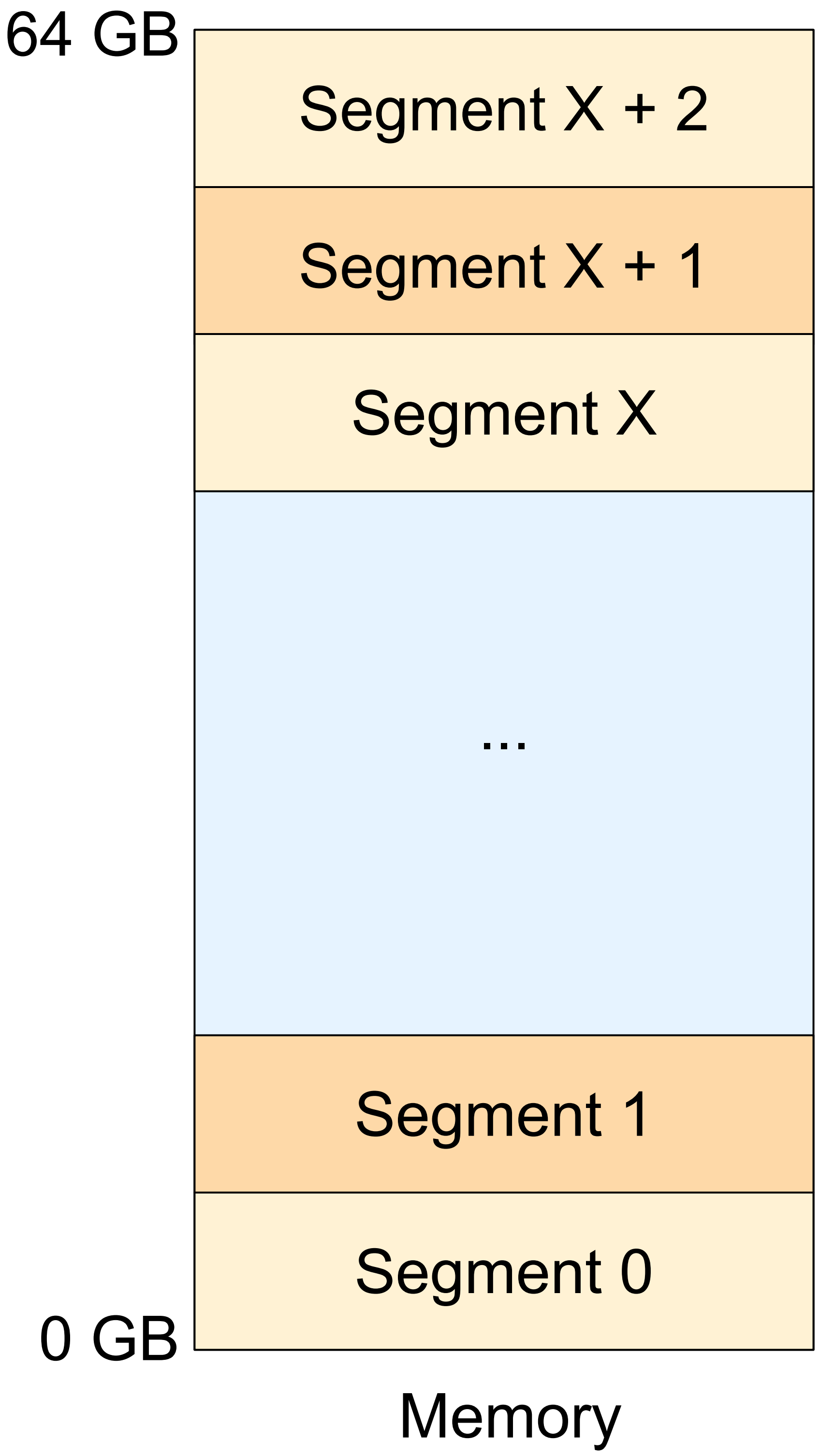
Segments & Cleaning

- **Logical *log* formed with fixed-sized *segments***



Segments & Cleaning

- **Logical *log* formed with fixed-sized *segments***



64 GB

Segment X + 2

Segment X + 1

Segment X

...

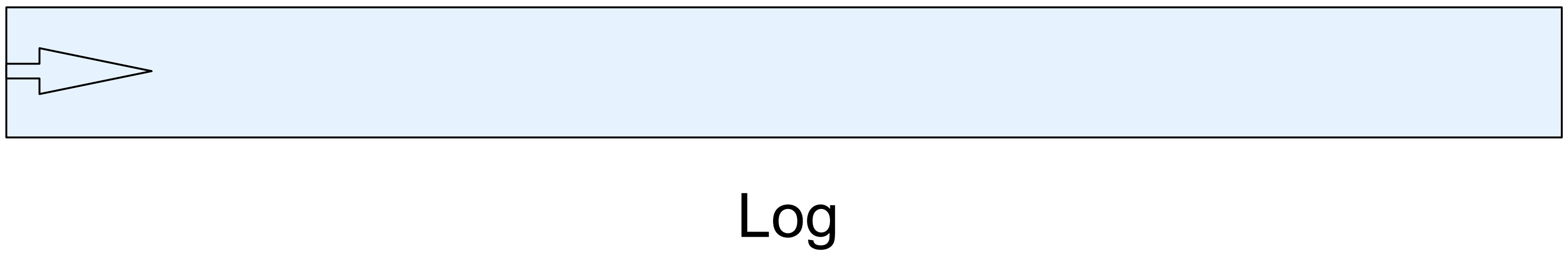
Segment 1

Segment 0

0 GB

Memory

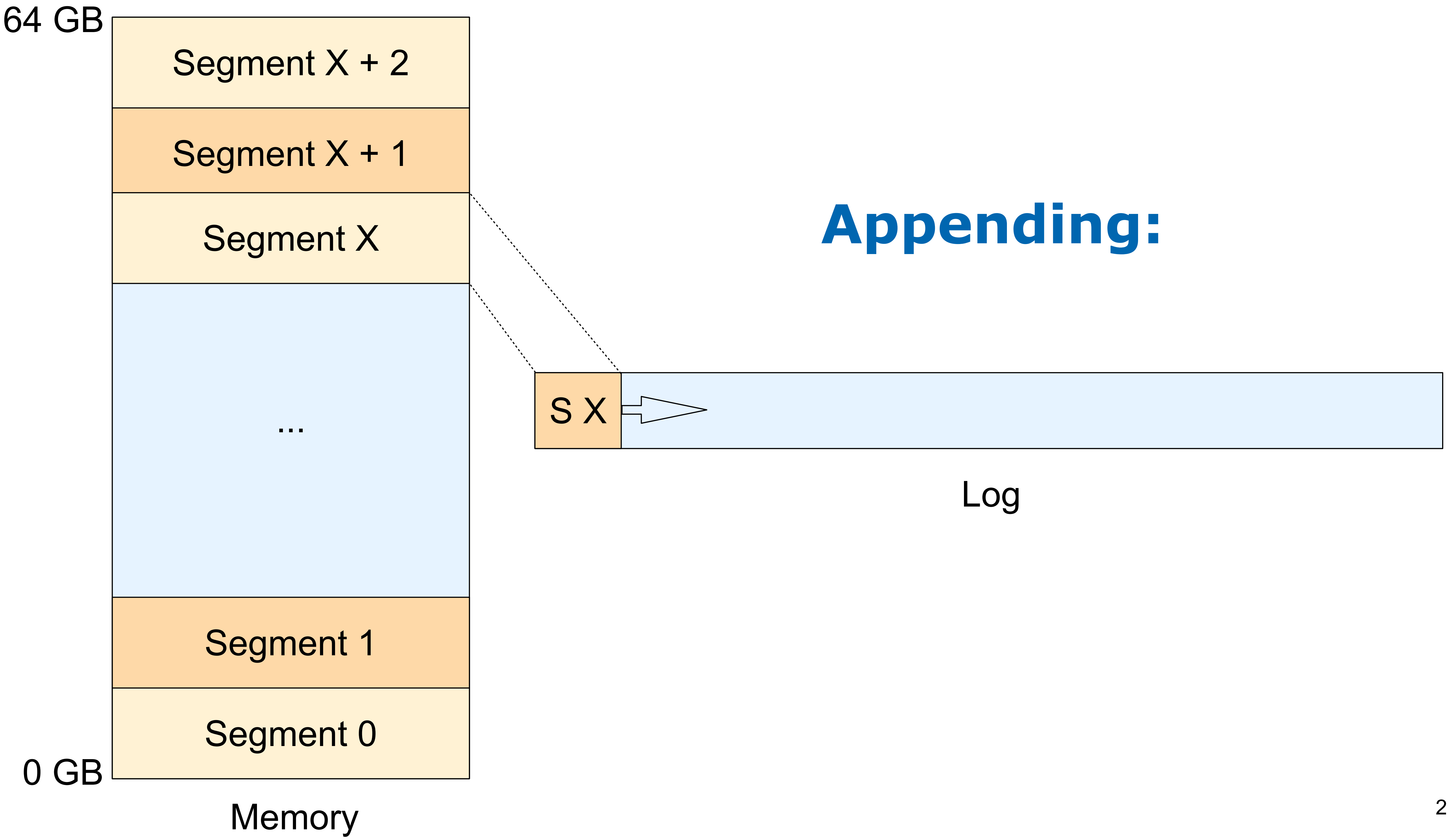
Appending:



Log

Segments & Cleaning

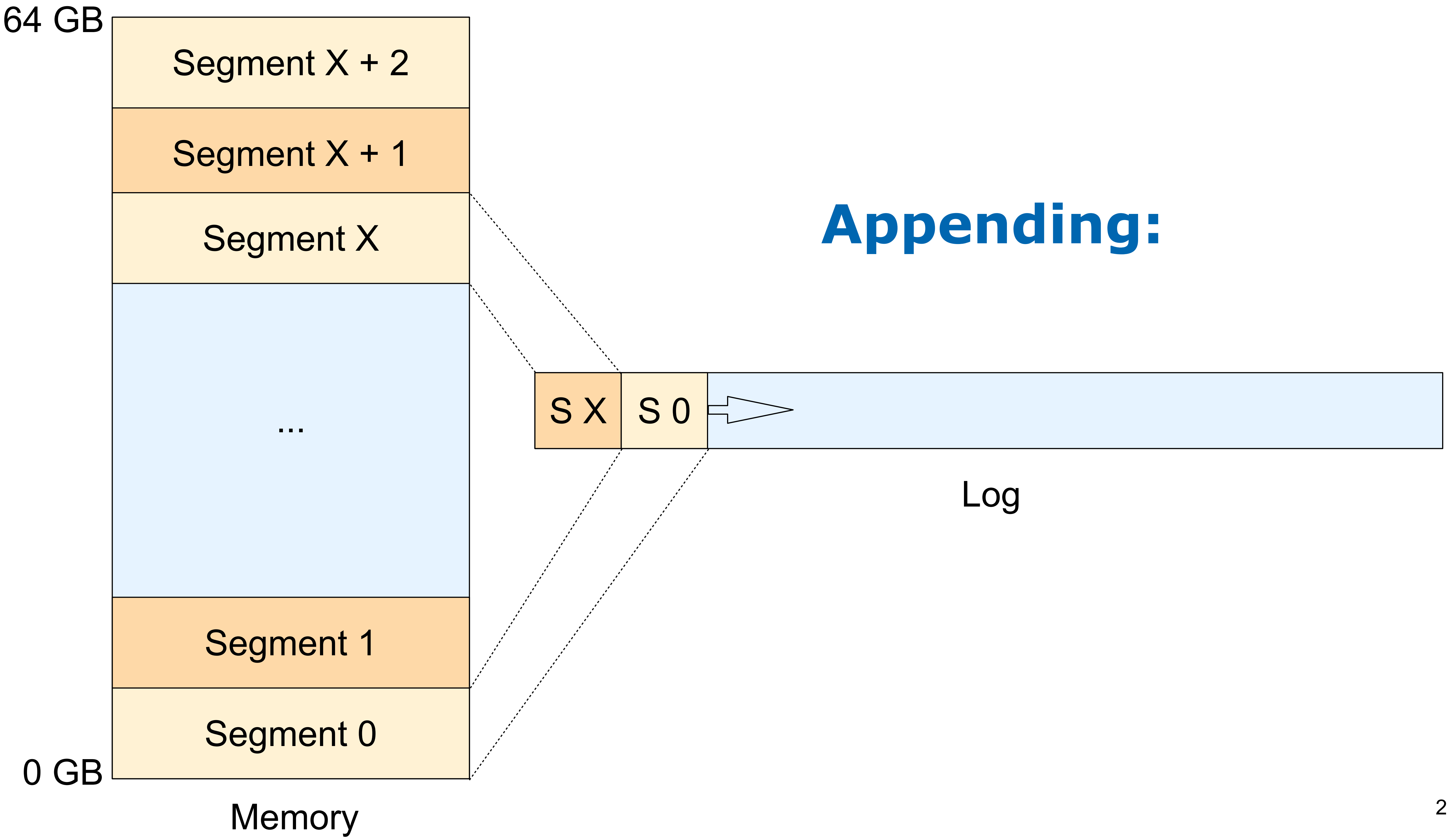
- **Logical *log* formed with fixed-sized *segments***



Appending:

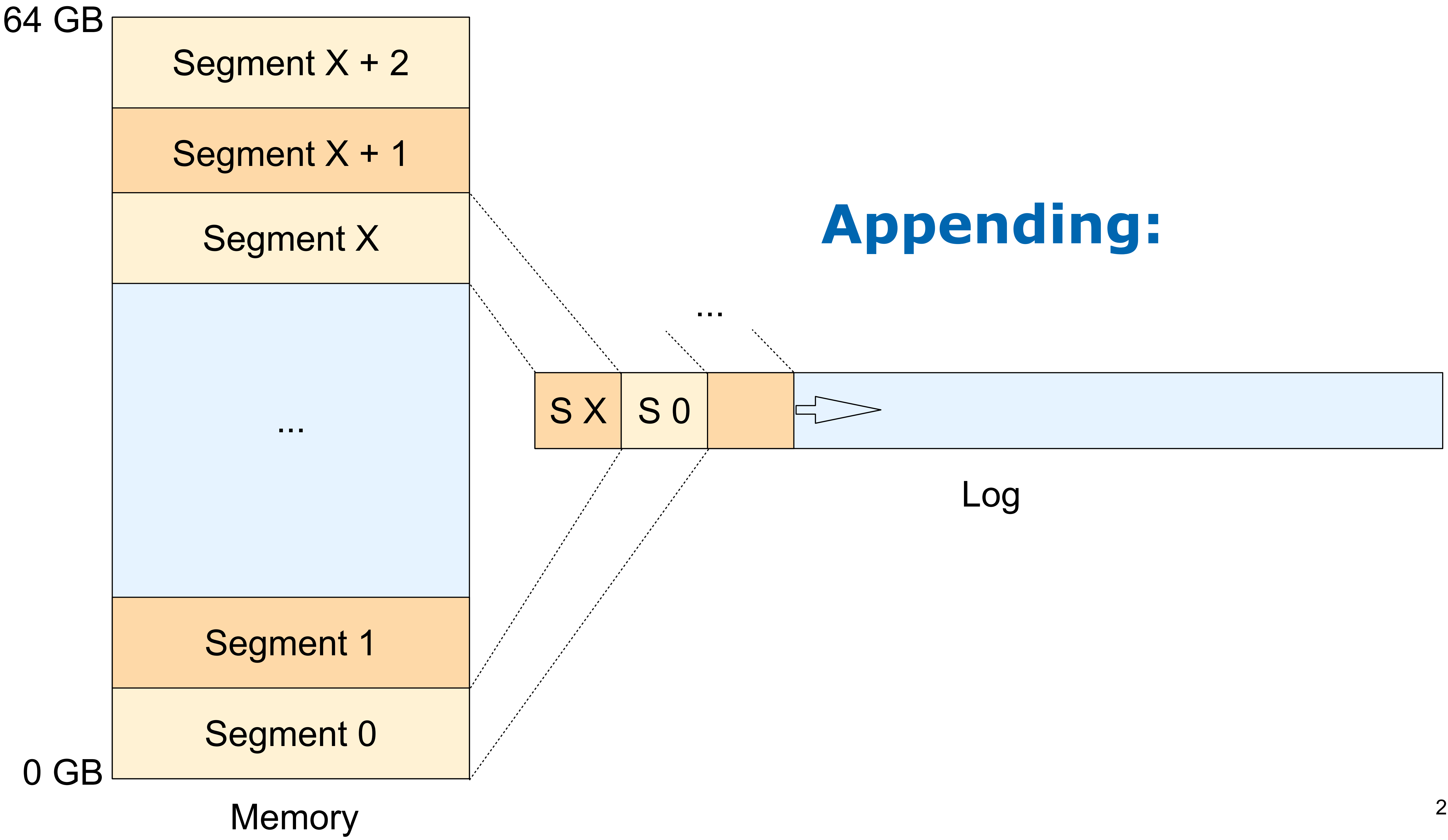
Segments & Cleaning

- **Logical *log* formed with fixed-sized *segments***



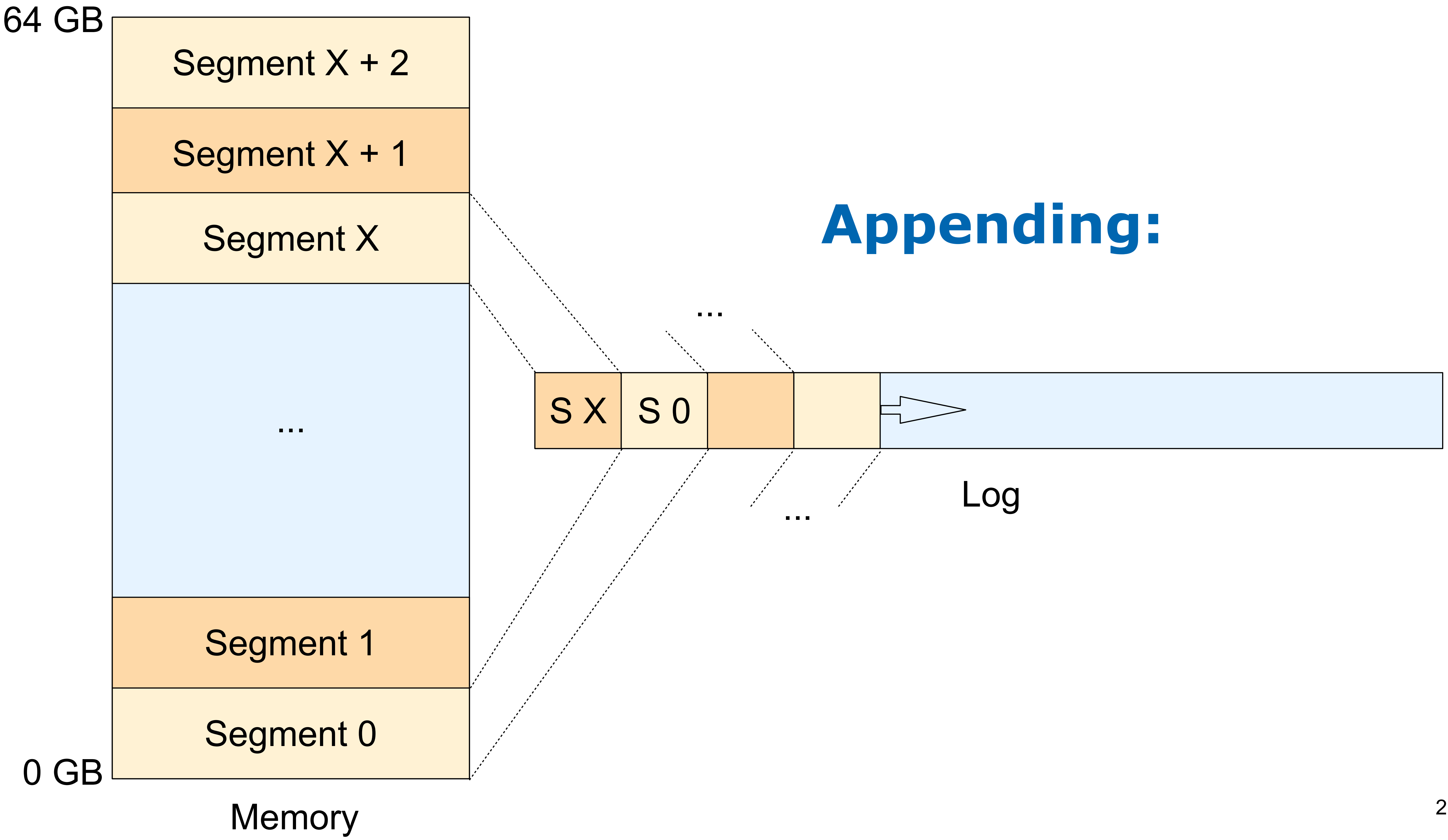
Segments & Cleaning

- **Logical *log* formed with fixed-sized *segments***



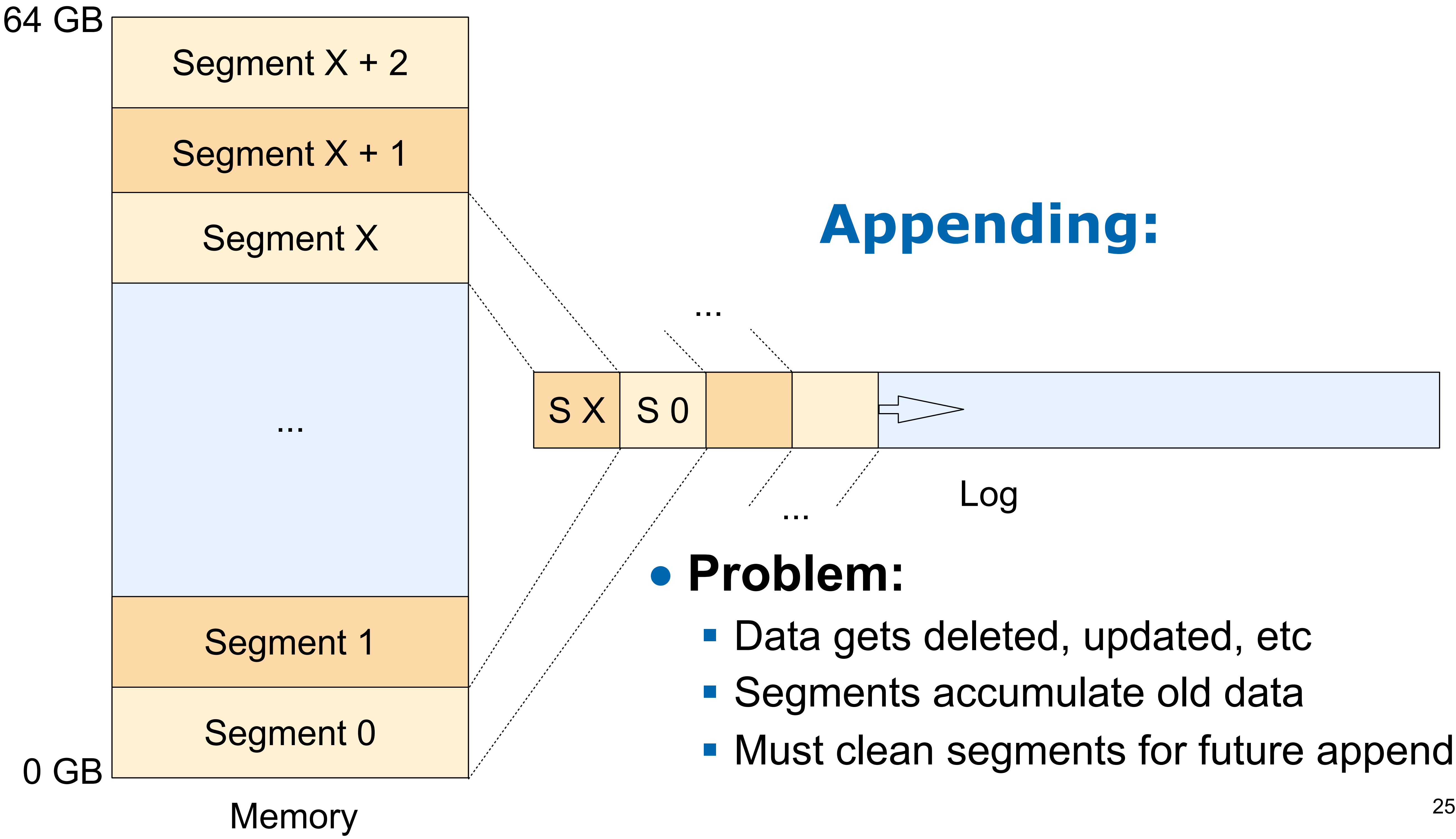
Segments & Cleaning

- **Logical *log* formed with fixed-sized *segments***



Segments & Cleaning

- **Logical *log* formed with fixed-sized *segments***

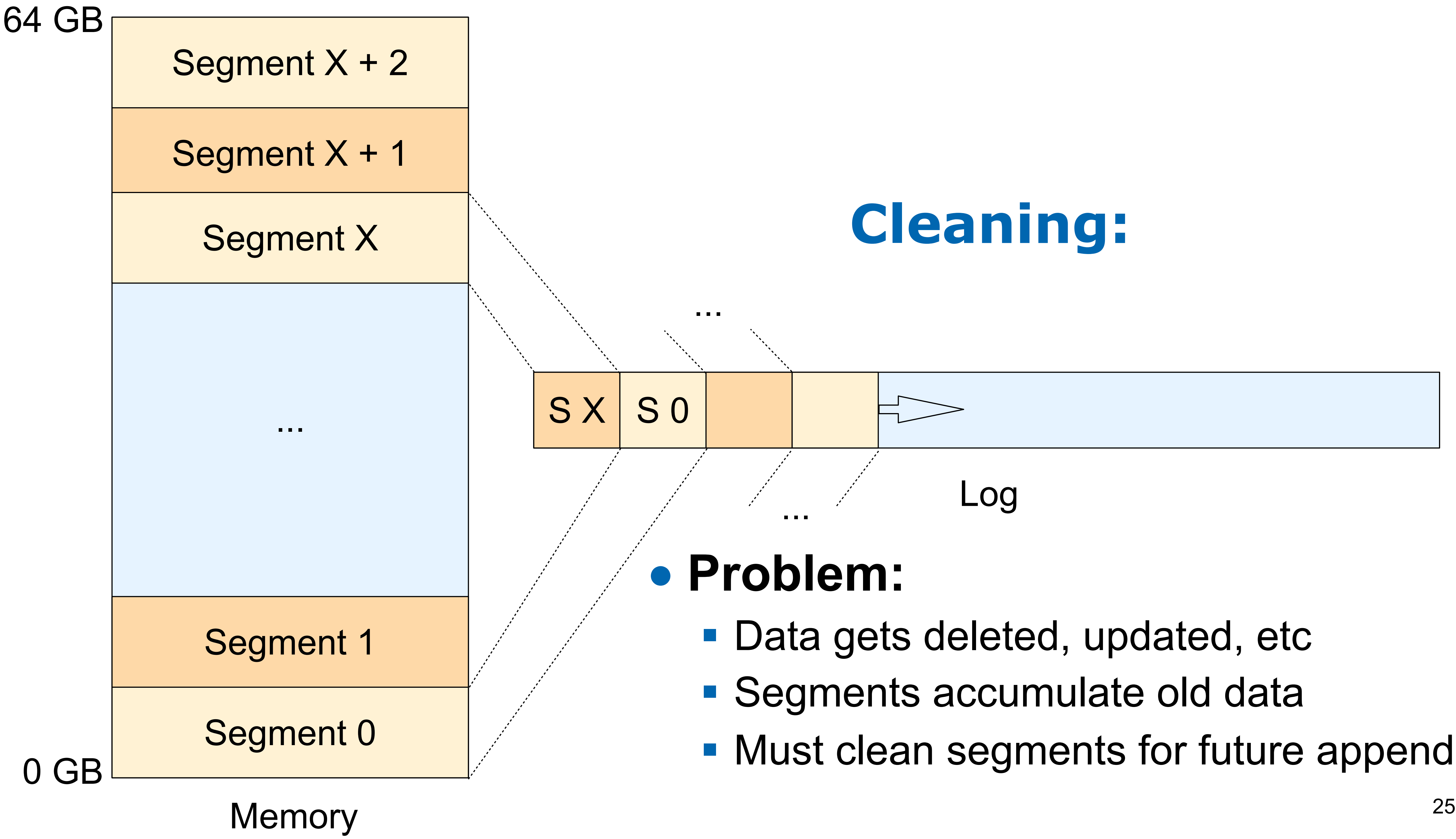


Appending:

- **Problem:**
  - Data gets deleted, updated, etc
  - Segments accumulate old data
  - Must clean segments for future appends

Segments & Cleaning

- **Logical *log* formed with fixed-sized *segments***

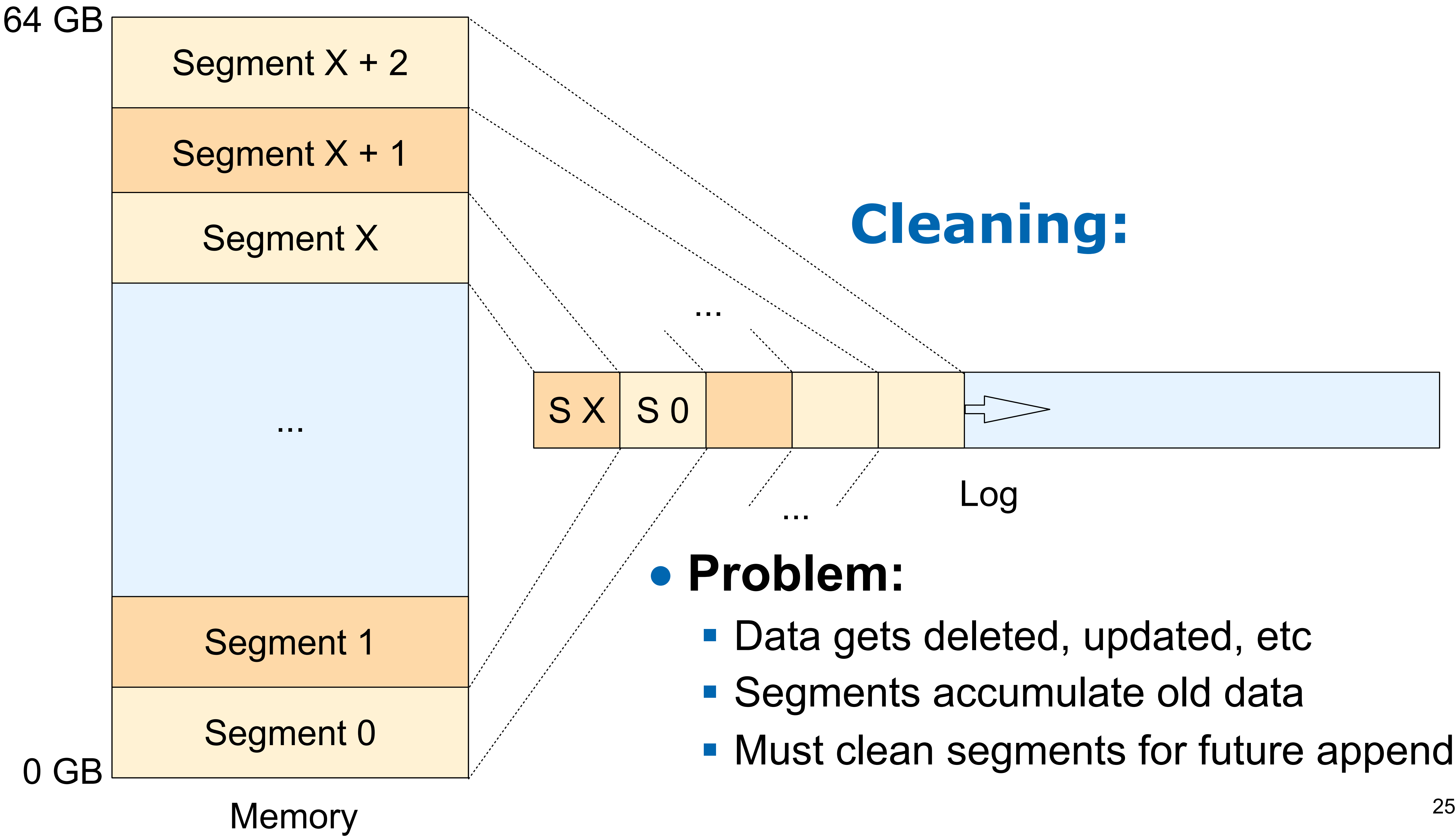


Cleaning:

- **Problem:**
  - Data gets deleted, updated, etc
  - Segments accumulate old data
  - Must clean segments for future appends

Segments & Cleaning

- **Logical *log* formed with fixed-sized *segments***

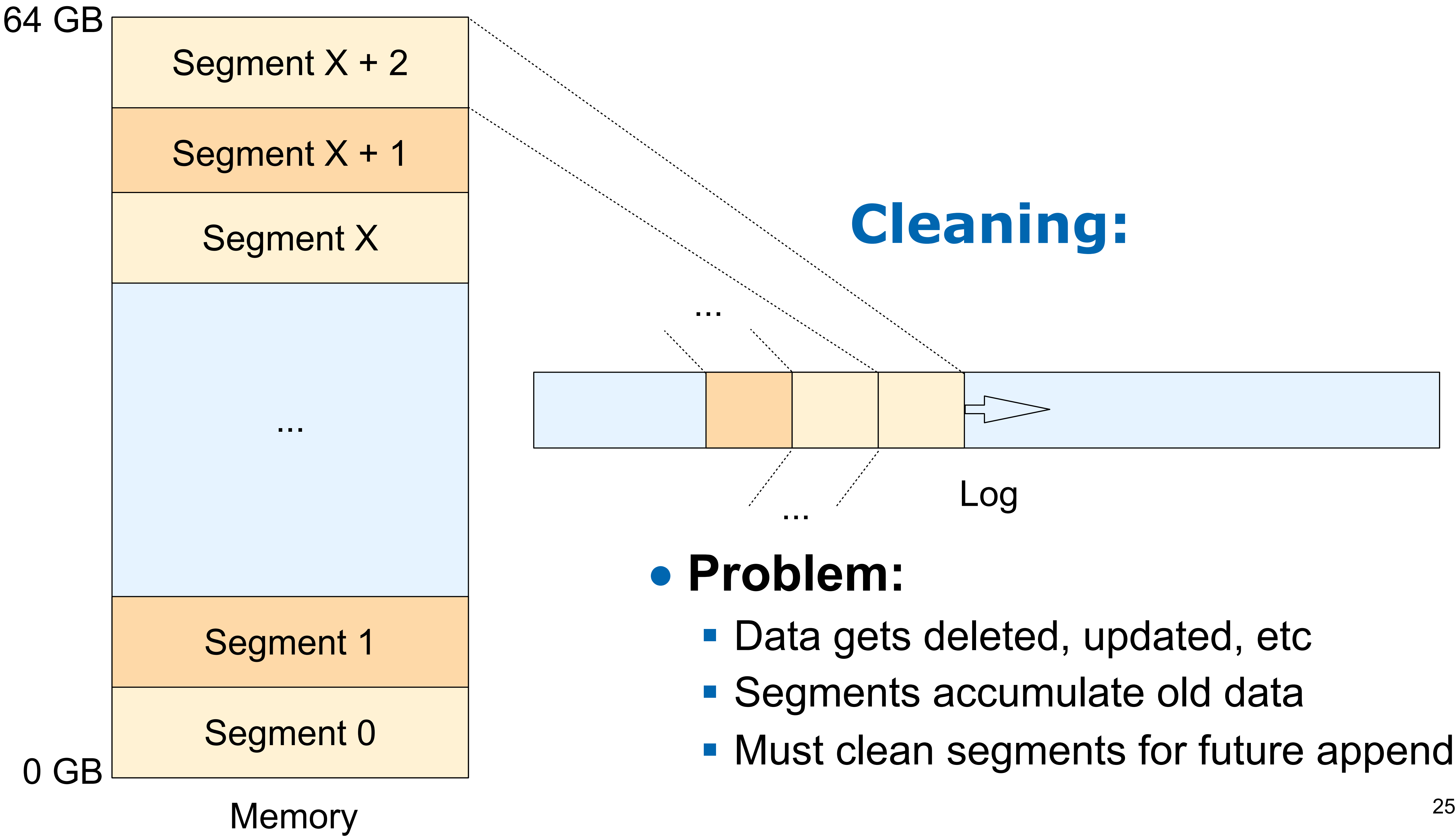


Cleaning:

- **Problem:**
  - Data gets deleted, updated, etc
  - Segments accumulate old data
  - Must clean segments for future appends

Segments & Cleaning

- **Logical *log* formed with fixed-sized *segments***



Cleaning:

- **Problem:**
  - Data gets deleted, updated, etc
  - Segments accumulate old data
  - Must clean segments for future appends