Log-structured Memory For DRAM-based Storage

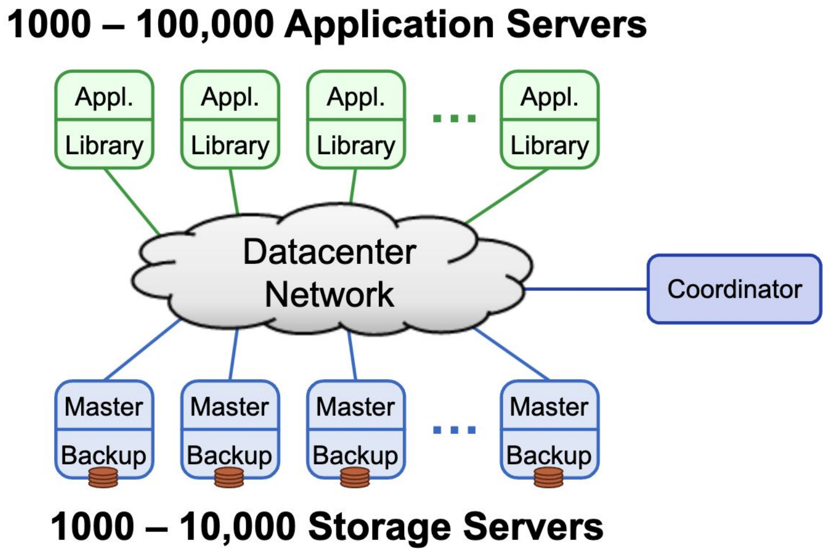
Paper by Stephen M. Rumble et.al.
Pics credit to the original paper and slides

Presenter: **Qing Feng**
MIT 6.5810

Nov 14, 2022

Context: RAMCloud

- A **key-value** in-memory storage
- **Low-latency network** for small objects (5 us read, 16 us write)
- Each storage server runs **master and backup** modules:
  - Master manages in-memory objects, serving requests
  - Backup stores copies of other masters in its local disk
- Central coordinator for config



Motivation

For a high-performance storage system serving multiple applications over a long time, a memory allocator has to support:

- Fast **allocation/deallocation**
- High **memory utilization**
- Handle **changing workloads**

Problem: Memory Utilization

Existing allocators under changing workloads waste at least 50% memory:

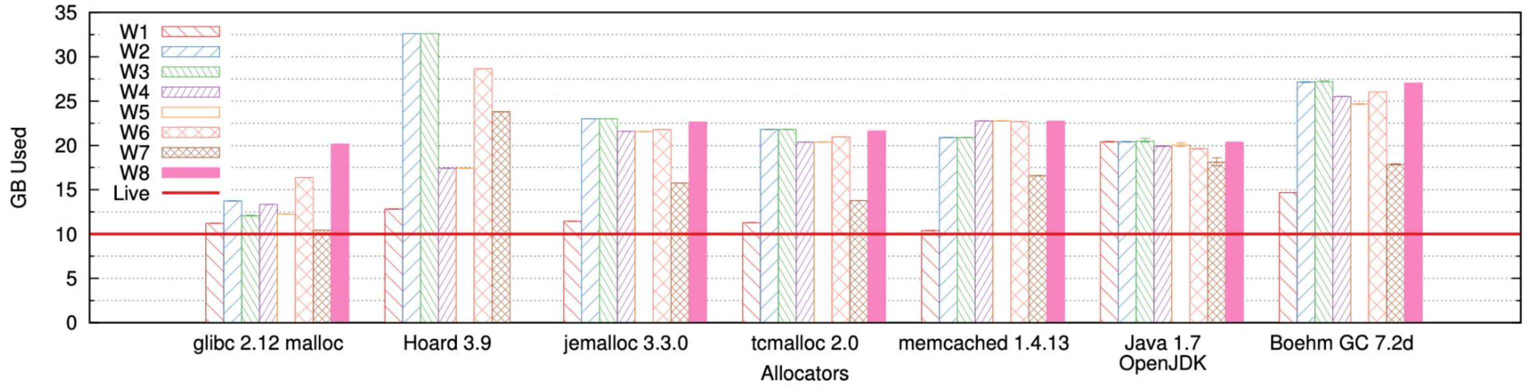
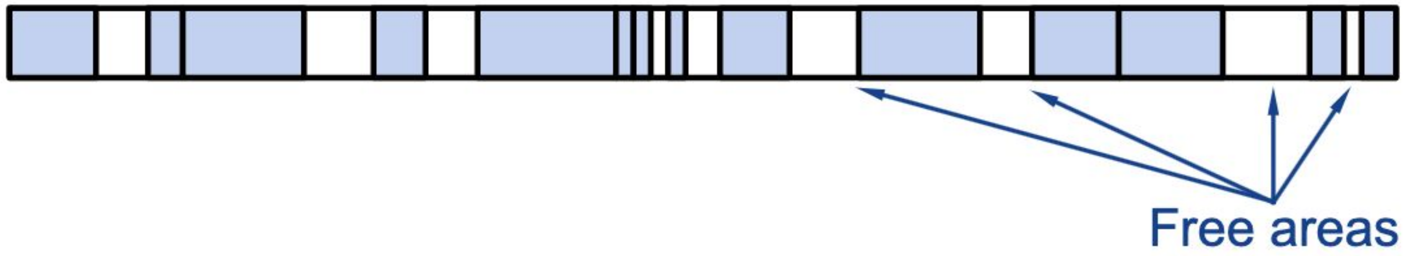


Figure 1: Total memory needed by allocators to support 10 GB of live data under the changing workloads described in Table 1 (average of 5 runs). "Live" indicates the amount of live data, and represents an optimal result. "glibc" is the allocator typically used by C and C++ applications on Linux. "Hoard" [10], "jemalloc" [19], and "tcmalloc" [1] are non-copying allocators designed for speed and multiprocessor scalability. "Memcached" is the slab-based allocator used in the memcached [2] object caching system. "Java" is the JVM's default parallel scavenging collector with no maximum heap size restriction (it ran out of memory if given less than 16 GB of total space). "Boehm GC" is a non-copying garbage collector for C and C++. Hoard could not complete the W8 workload (it overburdened the kernel by *mmap*ing each large allocation separately).

Why Malloc Does Not Work

Non-copying allocators (e.g. malloc)

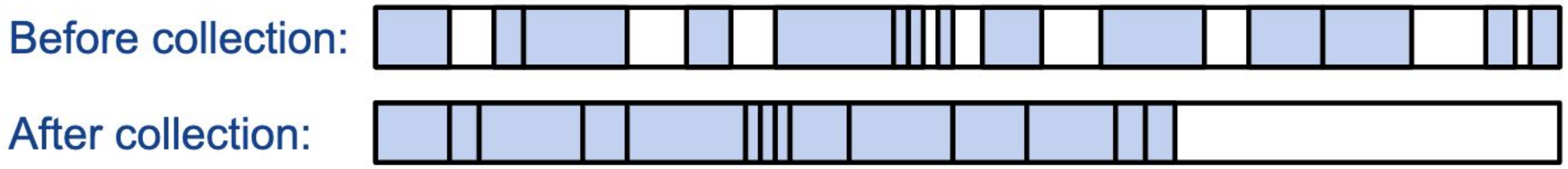
- Cannot move objects once allocated - **general pointer usage**
- Works well for apps with a consistent distribution of object sizes
- Vulnerable to **fragmentation**, thus lack efficient use of memory



Why Garbage Collector Does Not Work

Copying allocators (e.g. JVM garbage collector)

- Inevitably have to **scan all memory** to update pointers - expensive
- Full scan postponed to accumulate garbage - **low utilization**
- **Long pause time** - hundreds of microseconds at best, too slow compared with request latency (a couple of microseconds)



High-level Approach

Ideal memory allocator for a DRAM-based storage system:

- **Copying** - no fragmentation
- **Incremental** garbage collection - no full scan, free small regions independently

Key insight: In storage systems, **pointers are confined** to index structures where they can be located easily. (In this case, simply a hash table.)

High-level Approach

Ideal memory allocator for a DRAM-based storage system:

- **Copying** - no fragmentation
- **Incremental** garbage collection - no full scan, free small regions independently

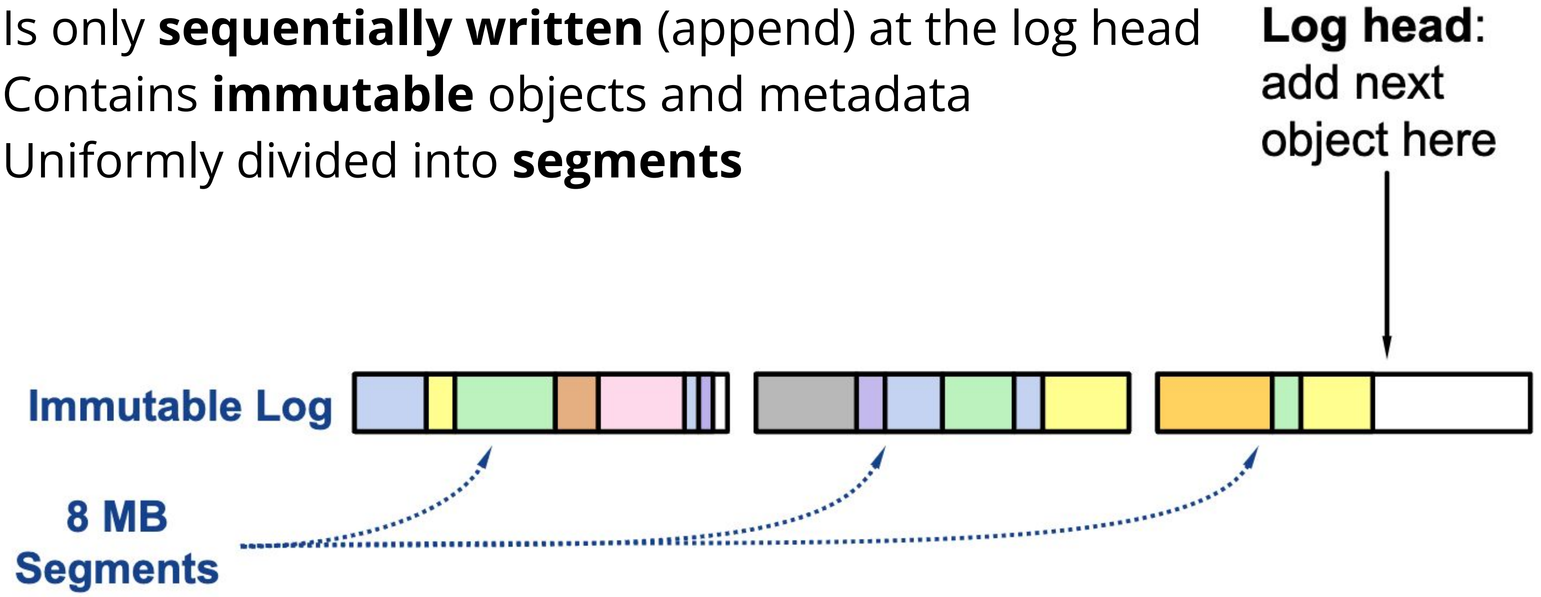
Key insight: In storage systems, **pointers are confined** to index structures where they can be located easily. (In this case, simply a hash table.)

Question: Why general-purpose pointer usage couldn’t work? Any way to improve?

What Is A Log?

Master server organizes its DRAM as a log, a data structure that:

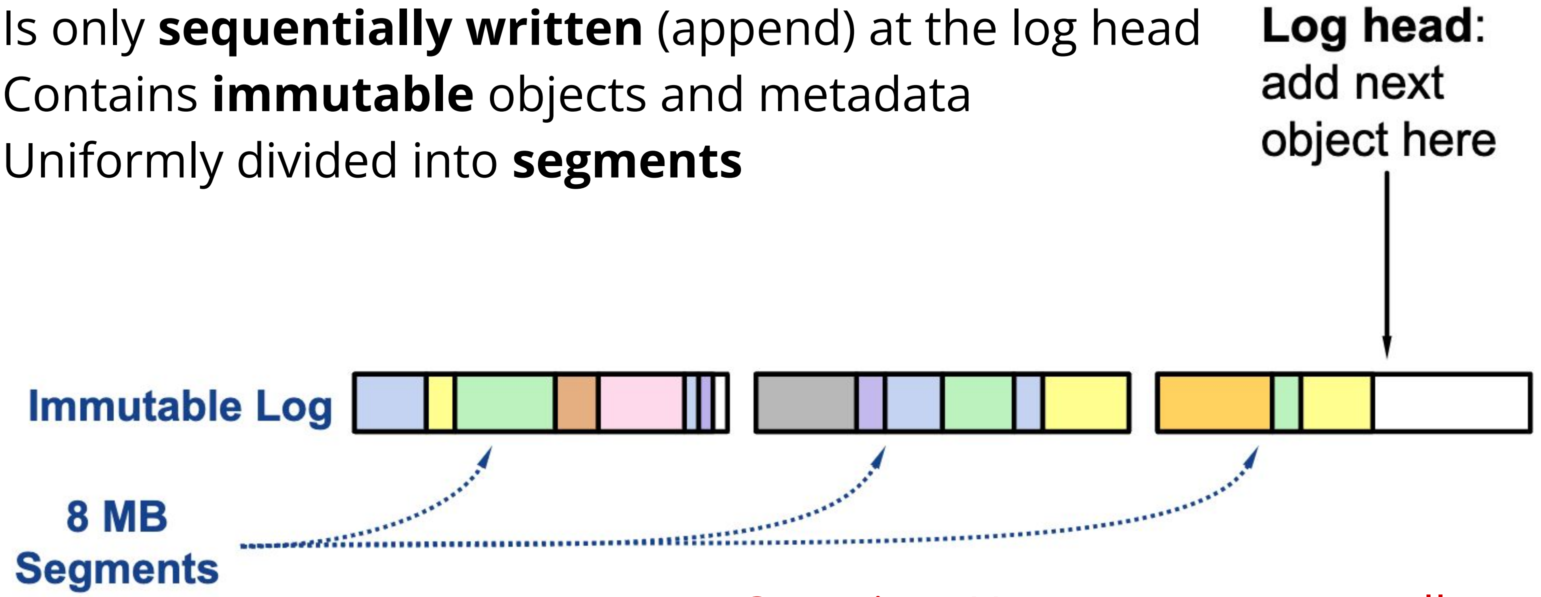
- Is only **sequentially written** (append) at the log head
- Contains **immutable** objects and metadata
- Uniformly divided into **segments**



What Is A Log?

Master server organizes its DRAM as a log, a data structure that:

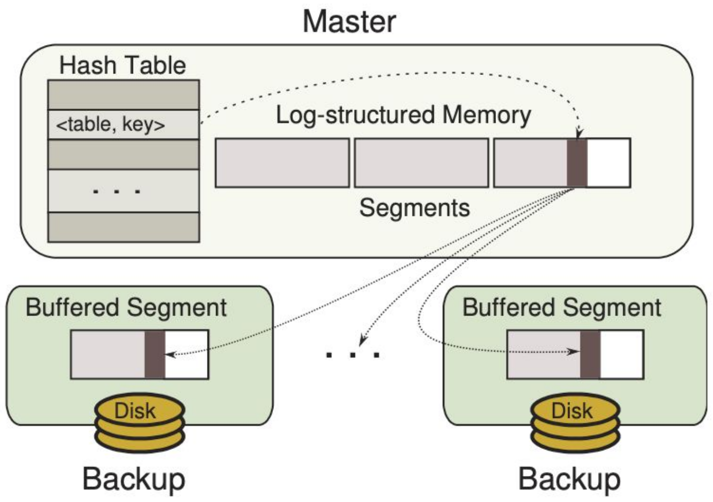
- Is only **sequentially written** (append) at the log head
- Contains **immutable** objects and metadata
- Uniformly divided into **segments**



Question: How are segments allocated?

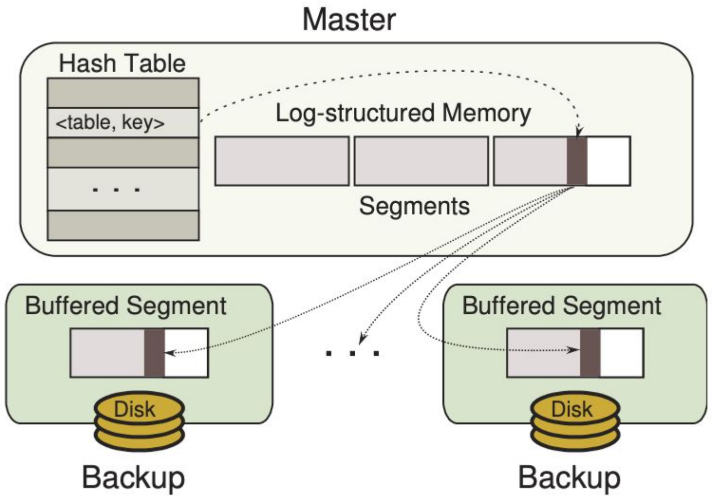
How Does The Log Work?

- A hashtable maps (table, key) to the location of an object. **Each live object has exactly one pointer** in the entry
- Each segment is replicated on a different set of backups, so replicas of a log are scattered across the entire cluster
- Backups **buffer data** and write to disk in segments:
  - Writes do not wait for disk I/O
  - Disk bandwidth is used efficiently



How Does The Log Work?

- A hashtable maps (table, key) to the location of an object. **Each live object has exactly one pointer** in the entry
- Each segment is replicated on a different set of backups, so replicas of a log are scattered across the entire cluster
- Backups **buffer data** and write to disk in segments:
    - Writes do not wait for disk I/O
    - Disk bandwidth is used efficiently



Question: How do backups ensure buffers survive crashes?

Log Metadata

In addition to object data, three types of metadata are needed:

- Each object contains its **self-identification**: table ID, key, version number, etc., to rebuild the hash table during crash recovery
- Each new segment has a **log digest**, a list of all live segments' IDs, to avoid a central repository of log information
- Deleted objects are marked by **tombstones** appended in the log, to avoid surrection during recovery (tombstones are garbage collected only when the deleted objects are removed)

Log Metadata

In addition to object data, three types of metadata are needed:

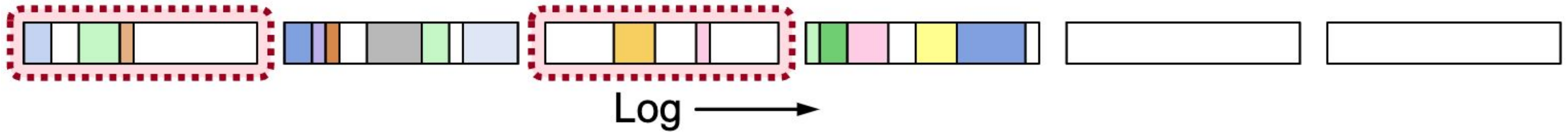
- Each object contains its **self-identification**: table ID, key, version number, etc., to rebuild the hash table during crash recovery
- Each new segment has a **log digest**, a list of all live segments' IDs, to avoid a central repository of log information
- Deleted objects are marked by **tombstones** appended in the log, to avoid surrection during recovery (tombstones are garbage collected only when the deleted objects are removed)

Question: If no crash recovery required (e.g. using NVRAM), can save most metadata overhead?

Log Cleaner

Incremental copying garbage collector:

- Cost-benefit segment selection
  - Choose segments with lower utilization to free
  - Avoid edge cases where cleaning uses more space than it frees

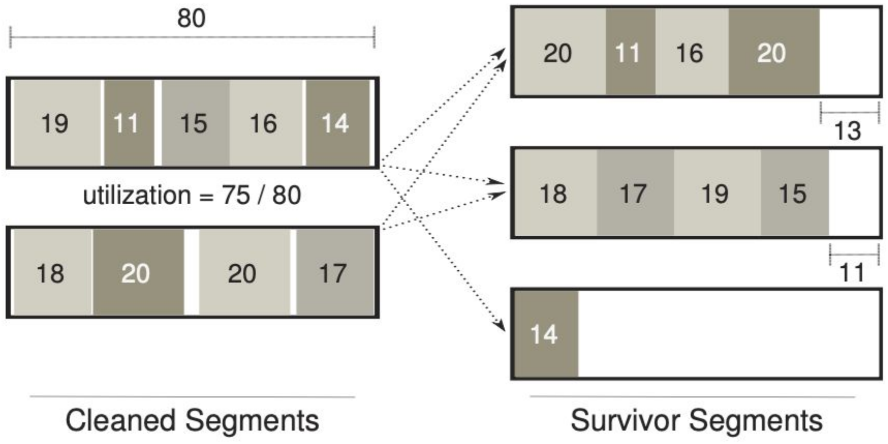


Log Cleaner

Incremental copying garbage collector:

- Cost-benefit segment selection
  - Choose segments with lower utilization to free
  - Avoid edge cases where cleaning uses more space than it frees

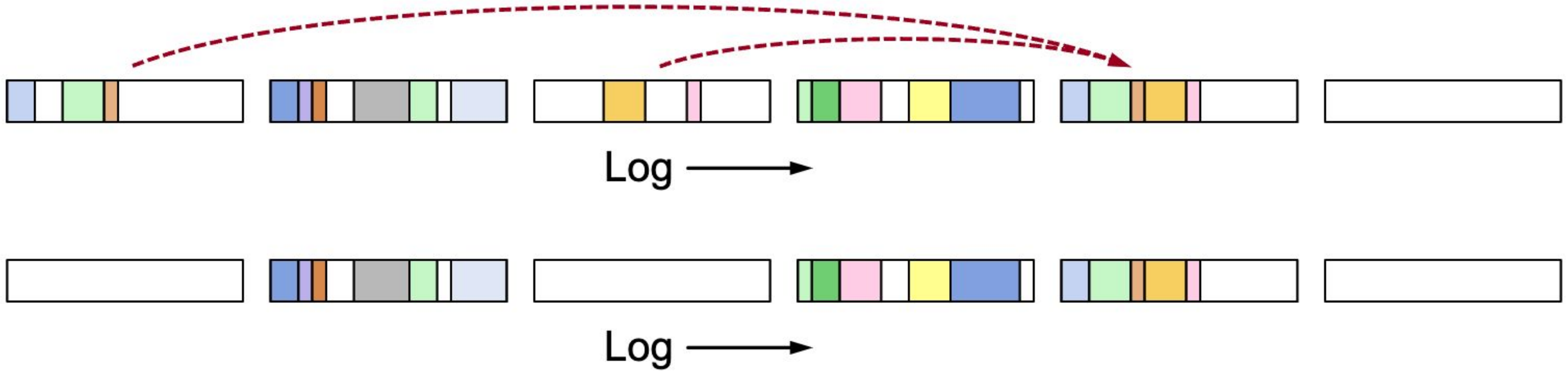
Question: Is log structure still prone to fragmentation?



Log Cleaner

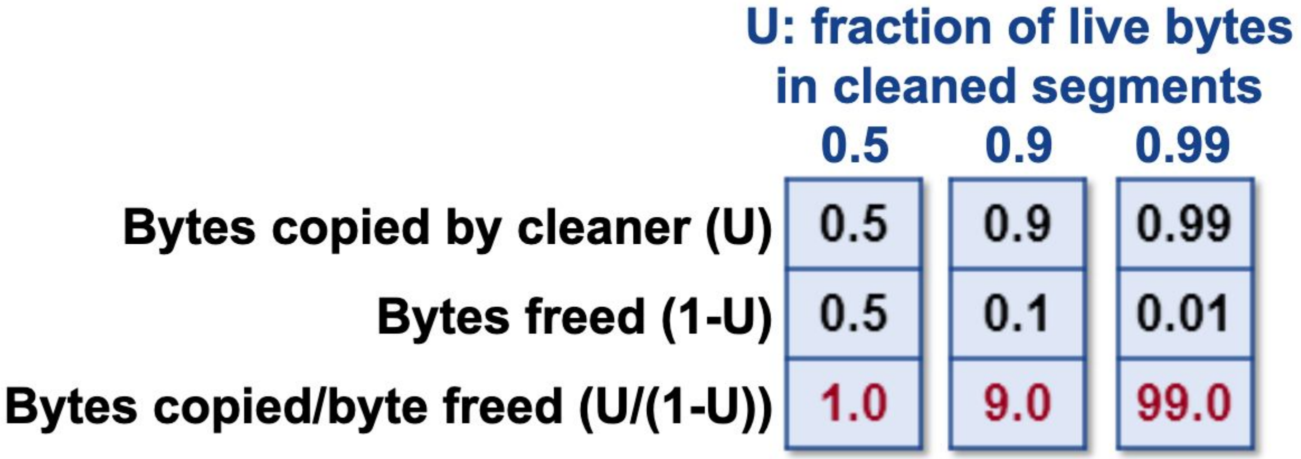
Incremental copying garbage collector:

- Concurrent log updates
  - Write survivor data to different segments (**side log**) than the log head
  - **Multiple cleaner threads** run in parallel, update the log using log digest
  - Separate segments avoid contention for single set of backup disks



Cleaning Cost

A fundamental trade off between space and time: **the higher the memory utilization, the more expensive the cleaning cost**, eventually the system will run out of bandwidth

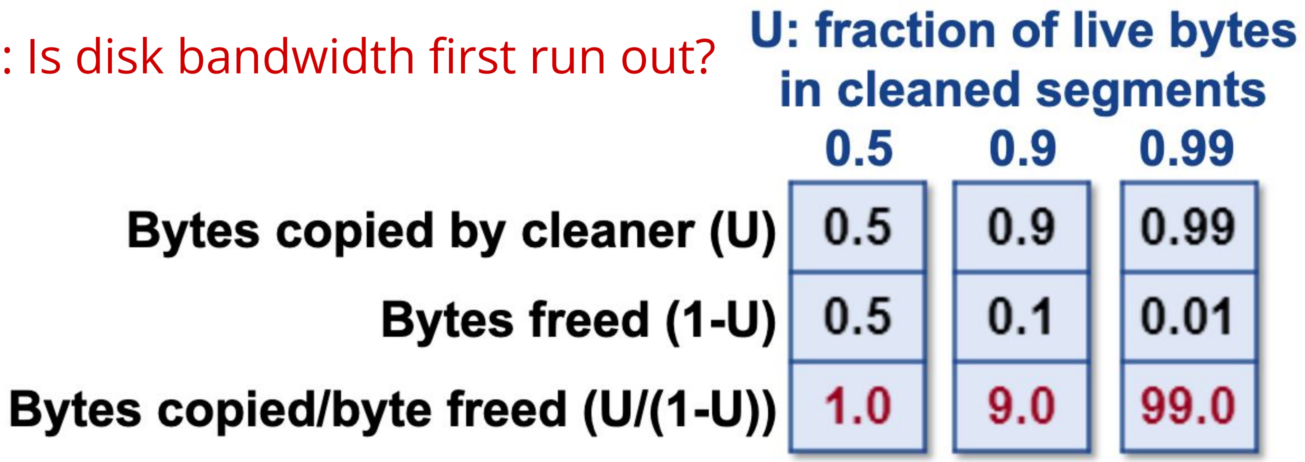


	U: fraction of live bytes in cleaned segments		
	0.5	0.9	0.99
Bytes copied by cleaner (U)	0.5	0.9	0.99
Bytes freed (1-U)	0.5	0.1	0.01
Bytes copied/byte freed (U/(1-U))	1.0	9.0	99.0

Cleaning Cost

A fundamental trade off between space and time: **the higher the memory utilization, the more expensive the cleaning cost**, eventually the system will run out of bandwidth

Question: Is disk bandwidth first run out?

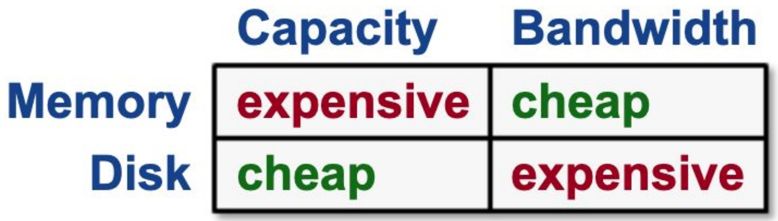


	U: fraction of live bytes in cleaned segments		
	0.5	0.9	0.99
Bytes copied by cleaner (U)	0.5	0.9	0.99
Bytes freed (1-U)	0.5	0.1	0.01
Bytes copied/byte freed (U/(1-U))	1.0	9.0	99.0

Two-level Cleaning

Key insight: **conflicting needs** between memory and disk, different policies are desired for the two levels of storage

- Disks are bottlenecked by bandwidth, memory by utilization
- **Memory can be cleaned without reflecting the updates on backups**
- **Memory can afford high bandwidth** for cleaning, **disks can tolerate lower utilization** due to less frequent cleaning

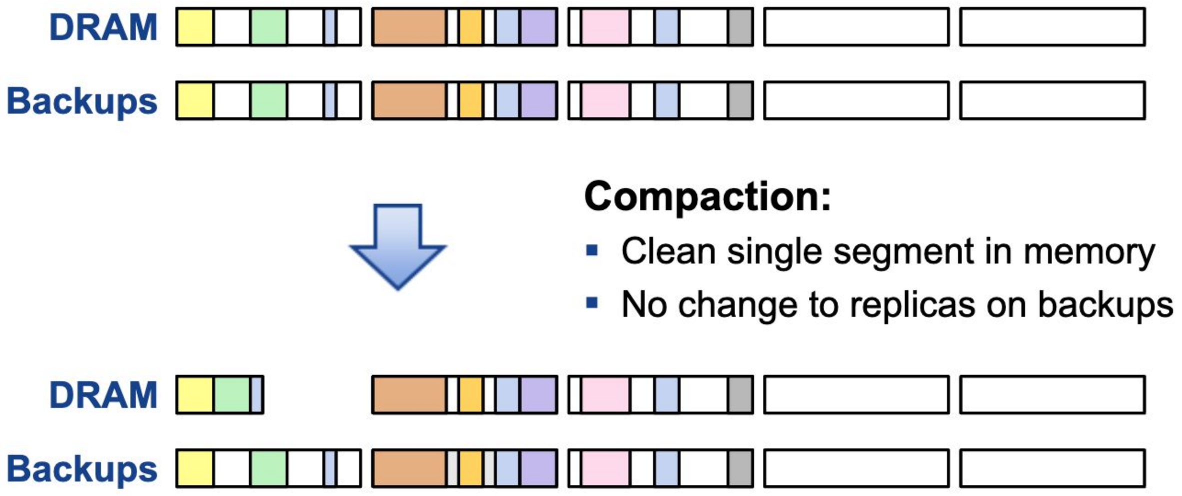


	Capacity	Bandwidth
Memory	expensive	cheap
Disk	cheap	expensive

Segment Compaction

Compacts a single segment at a time by copying live objects to a smaller region of memory and freeing the original segments - save disk bandwidth

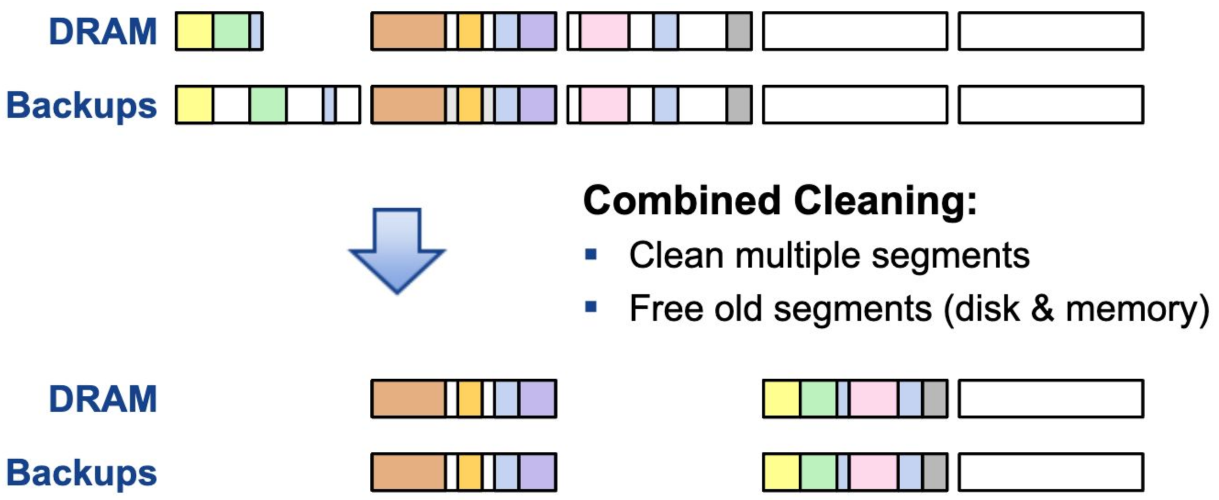
Problem: **Variable-length segments** - introducing seglets (64KB fixed-size)


DRAM
Backups
Compaction:
Clean single segment in memory
No change to replicas on backups
DRAM
Backups

Combined Cleaning

Frees cleaned segments on both memory and disk

Advantage: **Combined cleaning is postponed** and the effect of cleaning is accumulated



Parallel Cleaning

There are only three synchronization points in the system:

- Log head - introducing **side log**
- Update the hash table - **fine grained locks** on individual hash buckets
- Must not free segments in active use by service threads - **no additional locks required**, fixed by *wait-for-readers* primitive and *generations*

Deadlock Prevention

Cleaners use free space to free space, will deadlock otherwise

Solution: Reserving a **special pool** of seglets for cleaners, freed space are first used to replenish the cleaner pool

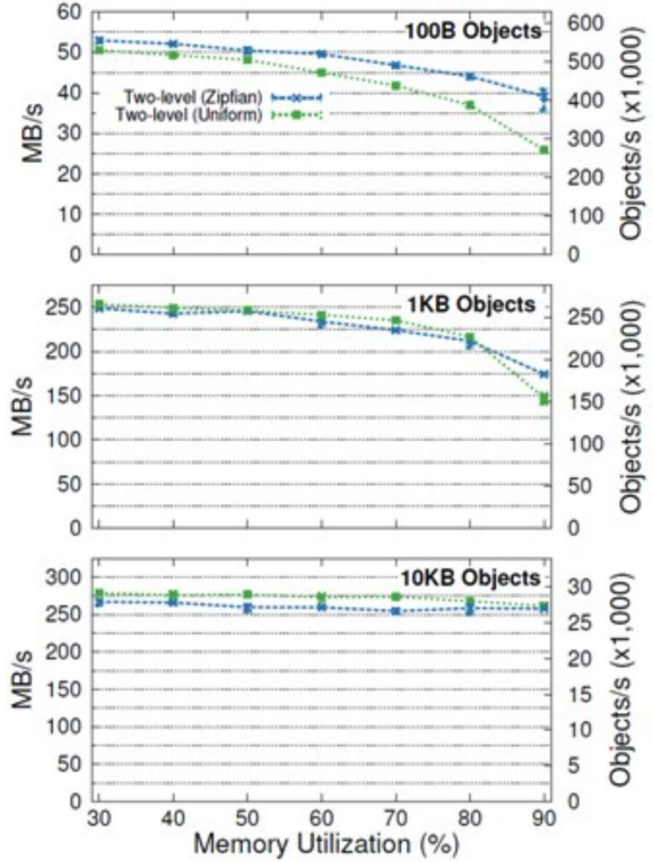
Cleaner appends a new log digest to free segments, can deadlock

Solution: Reserving two special **emergency head segments** that contain only log digests, alternate between the two

Evaluation: Throughput Vs. Memory Utilization

1 master, 3 backups, 1 client, concurrent multi-writes

- On high memory utilization (80% - 90%), tolerable performance degradation
- However, throughput is not very impressive? (Recall FDS achieves up to 100MB/s on single disk)

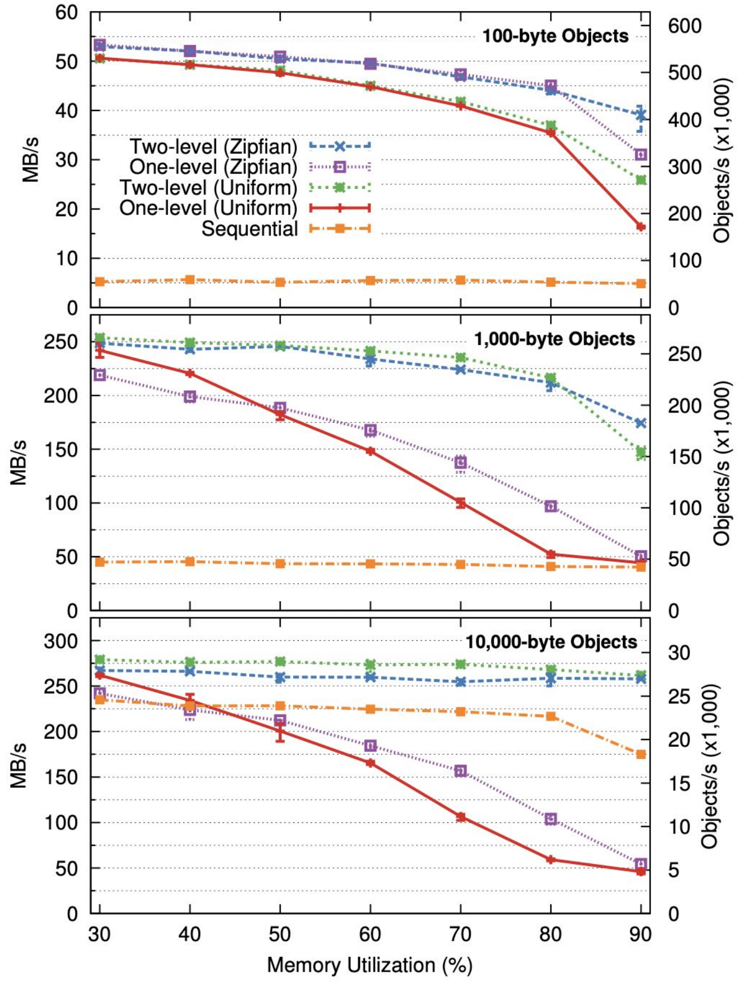


Memory Utilization	Performance Degradation
80%	17-27%
90%	26-49%
80%	14-15%
90%	30-42%
80%	3-4%
90%	3-6%

Evaluation: Two-level Cleaning

1 master, 3 backups, 1 client, concurrent multi-writes (except the “Sequential” curve)

- Two-level cleaning significantly improves performance, especially on high utilization (80% - 90%)

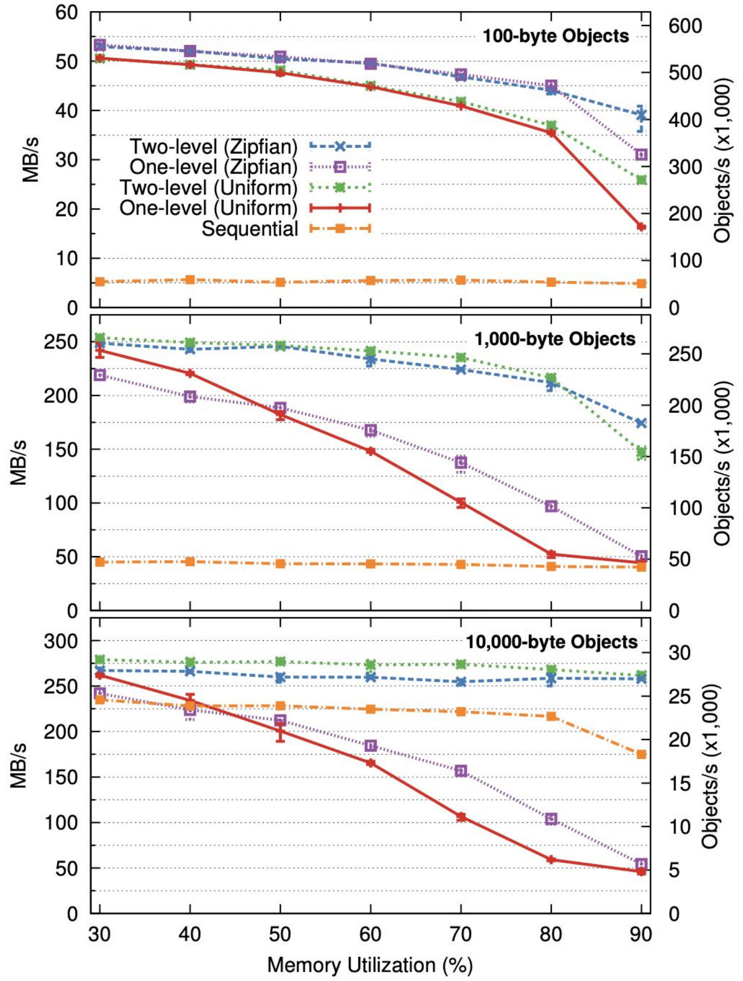


Evaluation: Two-level Cleaning

1 master, 3 backups, 1 client, concurrent multi-writes (except the "Sequential" curve)

- Two-level cleaning significantly improves performance, especially on high utilization (80% - 90%)

Question: Assuming bottlenecked by disk bandwidth on one-level cleaning (or even on two-level cleaning), would a faster disk-storage layer (e.g. FDS) help?

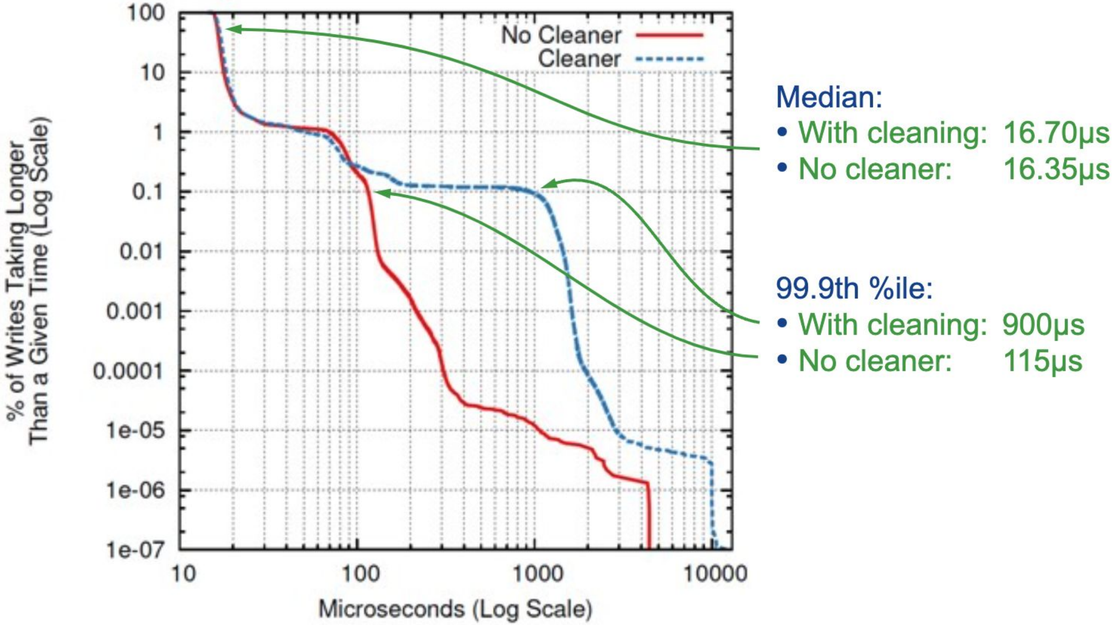


Evaluation: Latency

With vs. without cleaner, request latency remains almost the same up to 99.9th percentile

- Negligible median latency introduced
- Tail latency sources: contention for the NIC, RPC queueing delays in the single-threaded backup servers

1 client, sequential 100B overwrites, no locality, 90% utilization



Median:
- With cleaning: 16.70μs
- No cleaner: 16.35μs

99.9th %ile:
- With cleaning: 900μs
- No cleaner: 115μs

Discussion

- Can log-structured memory be adapted for more general usage than storage systems? How?
- Better ways of replication?
  - A separate layer of disk storage, like FDS?
  - Using NVRAM?
- Metadata overhead - already very good, or still can be improved?
- Compression? - another form of space-time trade-off
- Workloads change over time - cleaning more aggressively during periods of low load?

Log-structured Memory For DRAM-based Storage

Paper by Stephen M. Rumble et.al.
Pics credit to the original paper and slides

Presenter: **Qing Feng**
MIT 6.5810

Nov 14, 2022