Scalable Log Management in Plasma

Overview

The Plasma storage engine uses log structured storage model for storing data on persistent media. Data updates are always appended to the tail of the log. Update to a data page results in writing a new page segment and potentially making the older page segment versions garbage. The latest version of the pages on the log are referenced by the index structures and older versions become garbage. Plasma uses a copying garbage collector to clean the log by starting from the head of the log towards the tail of the log. For a disk oriented workload with no write caching, the log cleaner will become the bottleneck for scaling Plasma. We discuss the three major issues related to log cleaning in Plasma.

Problem 1: Single threaded log cleaner
A single thread is used for performing the log cleaning operation. When data size is higher than the available memory, there is no opportunity for performing write caching to reduce the write amplification. Hence, write amplification is high as well as the number of page segment blocks written to the log is going to be higher. Plasma writers can almost scale linearly by increasing

the number of writer threads. The single threaded log cleaner thread fails to catch-up cleaning the garbage generated by many writer threads for a disk oriented workload.

Problem 2: Read bandwidth usage
The log cleaner reads from the head of the log towards the tail of the log. It evaluates the blocks in the log for valid blocks. If a valid block is found, it rewrites the page corresponding to the block to the tail of the log. To evaluate a block for its validity, it requires only the pageID for the page in the block. But, currently the log cleaner reads the entire block to obtain the PageID consuming a large chunk of the available read bandwidth.

Problem 3: Sequential cleaning
The log cleaning in Plasma follows cleaning from head towards tail of the log, essentially cleaning the old data towards recent data. This model works great for a data update workload which is uniformly distributed. But for a workload where data updates are skewed and focussed to a small subset such as 1%, cleaning becomes very inefficient. The cleaning will have to move a large chunk of valid data to clean the garbage. The data movement consumes significant read and write bandwidth.

Problem 4: Expensive item validation during cleaning for value separation (Non GSI use case)
When values are separated from pages, the page holds a value offset pointers along with the item key instead of storing the value along with the key. During log cleaning, it has to verify if the value if valid by doing a lookup operation in the page and check if any of the item's value pointer points to the current value offset. When pages are mostly resident only on the log/disk, every item validity check requires reading the page from the log and it is expensive.

This document aims at proposing scalable solutions to the above problems.

Scalable Log Design

Log Segments

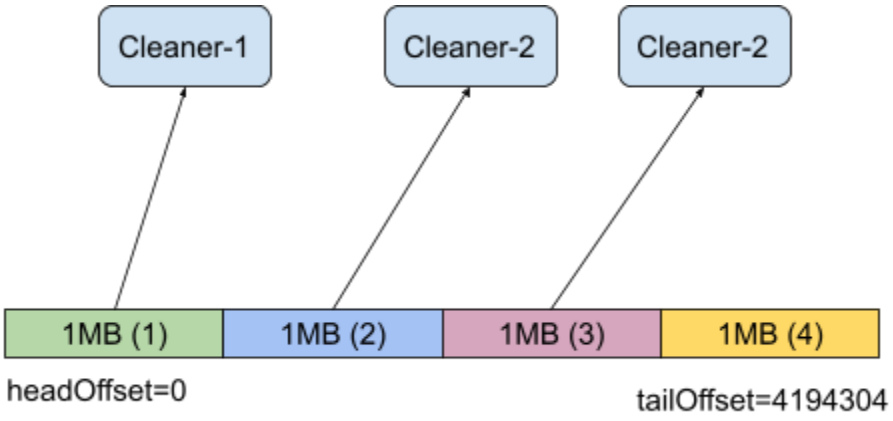
A log segment is a fixed size physical storage unit of the log. The log segment size should be in the multiples of SSD erase block size. A segments maps to a range of data offset in the log. The default size will be 1MB. A log is a linked list of segments.

By introducing the segments to the log, we are able to independently operate on different log segments concurrently. The log contains variable size blocks. Hence, it is difficult to find the starting offset of a block by randomly reading from an offset in a log. Since a segment is fixed size, we can randomly pick two different segments for processing. Each segment could also store metadata information about the segment.

The segments could contribute to internal fragmentation. The log writer consists of a tail flush buffer. The data is flushed to the log only when the flush buffer is full. But, when we need to do a synchronous log persistence, log segments may cause an extra padding block equal to the segment size. Synchronous writes are performed only during recovery point creation.

Concurrent Garbage Collector

The addition of log segments allows plasma to independently operate on different log segments as it eliminates the need for sequential block by block operation during log cleaning.
We introduce a cleaner tail offset and a cleaner head offset. When the fragmentation threshold goes above the limit, any log cleaner worker can pick-up the cleaning work by atomically incrementing the cleaner tail by segment size.



The cleaning work is straightforward and follows the same copying garbage collection technique used in the current implementation. The tricky part is to maintain the head and tail offset metadata of the log. Different workers concurrently operate on different segments and they operate at different speed. As the cleaning progresses, we need to update the head offset. If the thread processing segment-3 finishes before threads processing segment-2 or segment-1, we cannot update the log head as segment-3's offset. It will lead to log corruption. The log head can be updated only in the sequential manner as it's log offset. Even though segment-3 cleaning is completed, log head can be updated to segment-3 after finishing segment-1 and segment-2 cleaning. A reference count based efficient parent-child relationship can be build for the segments to implement sequential head offset update. Similar technique has been employed in the lock-free flush buffer implementation for log.

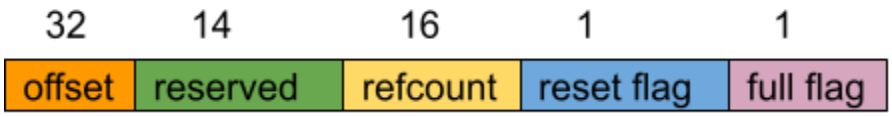
Each segment should have a small metadata header to indicate the size of the segment. If a block stored in the log is greater than default 1MB size, this metadata indicates that it uses n number of segments.

Segment Summary Block

The log cleaner reads the log data from head of the log towards the tail. Log cleaner only looks at the PageID and the PageVersion from the lss blocks. PageID and PageVersion only consumes tiny fraction of the log space. Log reads for fetching the PageID and PageVersion consumes about 30% of the read bandwidth. The bandwidth consumption can be reduced significantly by adding a summary block to the log segments. The summary block will summarise the list of PageIDs and PageVersions for the blocks in the log segment.

The key challenge is to maintain the summary block in a lock-free manner. All writes are performed into the lockfree flush buffer. Earlier, when a block is reserved in the lockfree flush buffer, the offset was incremented atomically with complex lockfree reference counting scheme.

The lockfree buffer maintain a 64 bit state variable. The layout of the state is as follows:

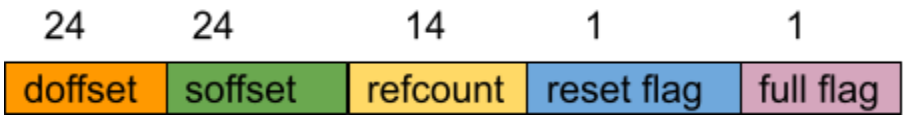


The log data can be stored at the head of the log segment and the summary can be stored at the tail of the segment. They can grow in opposite direction until they meet. Maximum space is limited by 1MB.

Log segment with data growing from left to right and summary growing from right to left

Since we need to track both data portion and summary portion in the log segment, we need two offsets. DataOffset and SummaryOffset. Since the log segment size is now restricted to 1MB, we do not need 32 bits to represent the data.

The state variable layout can be changed to accommodate summary block information as follows:

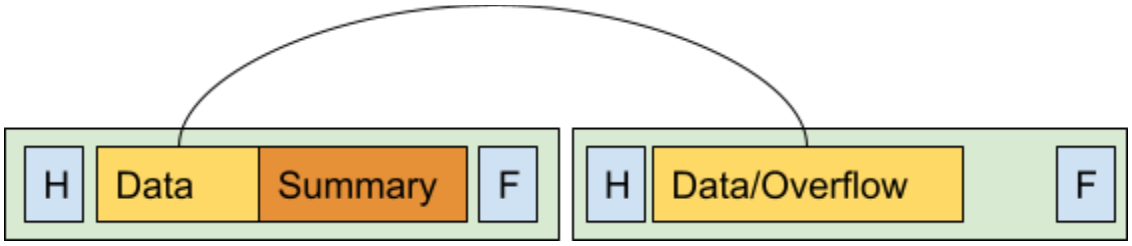


The data offset and summary offset can represent up to 16MB log segment. Maximum number of accessors possible is 16384.

The segment has a fixed size header and footer block. The header and footer stores the following information:

1. Size of data
2. Size of summary
3. Is the block an overflow block

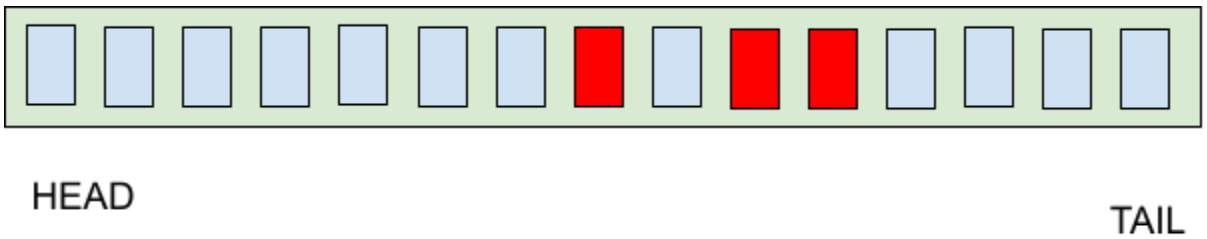
Plasma does not restrict the size of data stored in the log. If an object is bigger than segment size of 1MB, we have to store it using overflow log segments.



When the log cleaner observes a log segment with overflow, the footer indicates to ignore that log segment for cleaning.

Garbage Aware Cleaning

The fundamental problem with sequential log cleaning is that it will have to perform lot of additional work when large portion of the data is cold and a small percent is hot data. In the following diagram, the log is 20 % fragmented. But the garbage data (red blocks) are in the middle of the log. Now, in order to reclaim space, we have to read and relocate 50% of the existing data in the log. This consumes significant read and write bandwidth.



Garbage aware cleaning aims at a new design for log storage by adding capabilities for granular garbage collection. Earlier sections of this document discusses about splitting the log into log segments. Instead of cleaning the log segments in the order in which they were created, the cleaning is performed on log segments which have got the highest garbage. This significantly improves the storage bandwidth utilization. The garbage aware cleaning requires checkpointing in Plasma for it to work.

This design also enables Plasma to work directly on a block device and avoids any need for a filesystem.

The log segments are physical unit of the log and they have a starting and ending physical offsets. The log segments are connected in the order of their writes in a chain called log chain.

Log segment metadata management

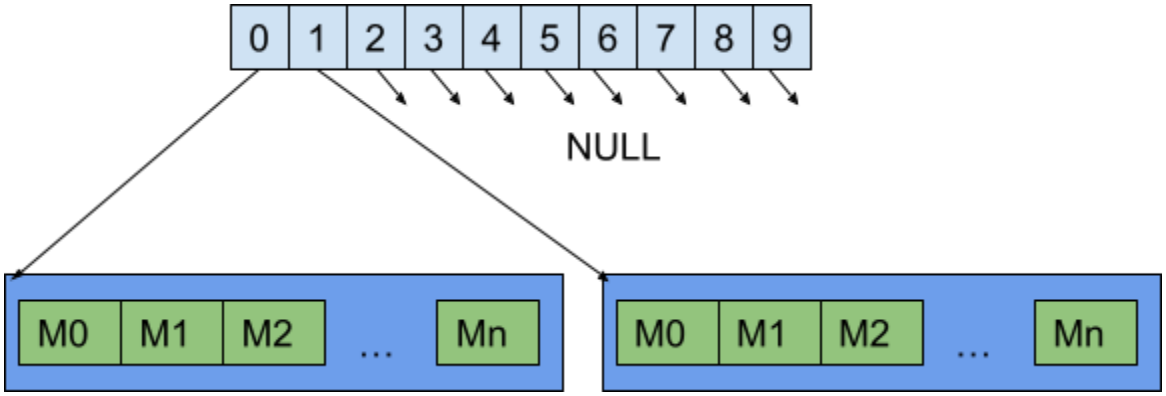
Each log segment has a metadata object held in memory. The metadata consists of:

1. Valid data size
2. Valid item count
3. Total item count
4. Seqno
5. State information (is_free, cleaning)

Each log segment is identified by the logID. The log segments can be stored in single file or multiple files. The logIDs are generated by using a combination of file number and file offset. To locate the in memory metadata object for a log segment, the metadata objects are stored in chunks of blocks of memory directly addressable by performing modulo operation. Using virtual memory reservations, we could potentially reserve a large memory area without allocating memory to those locations. Even though virtual memory reservation based technique is optimal and easy, we need a platform independent implementation.

The challenge here is that the log can grow based on the total data growth. Before initializing the log, we do not know the maximum size of the log to preallocate the log segment metadata object arrays. Even if we estimate the total, we cannot afford to allocate memory for the future growth. We need the metadata memory usage to grow along with the data size.

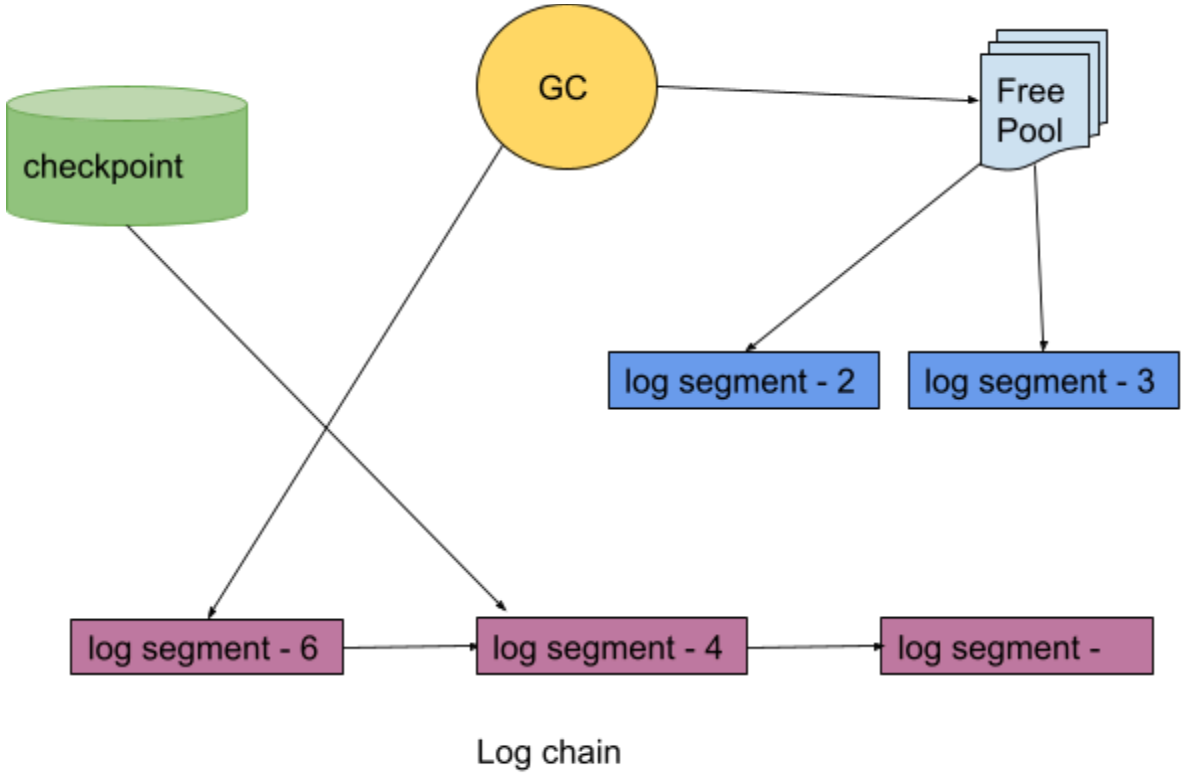
We can estimate the maximum possible data size to be 100TB. We can implement a sparse index structure (BTree like) in memory as follows:



If we limit the maximum log space to 100TB, we can build an array index with 100 entries pointing to the metadata object arrays (M0..Mn). Each entry will account for 1TB of log space.

If the metadata object for the log is 32 bytes, 1TB log space requires 32MB of metadata. Initially only index 0 points to the log metadata array with 32MB in size. Once the log space grows above 1TB, a new metadata array of 32MB will be allocated and index position 1 will be filled. The lazy allocation approach limits the memory growth for the metadata objects.

Log chain management



Log chain

The log segments are chained together in the order of their writes in a log chain. When a log segment is written to the durable storage, the next log in the chain is decided and written to the header metadata of the log segment being written. This enforces log chain durability. When a log segment is written to the durable storage, it is assigned a monotonically incrementing sequence number. All the log segments in the chain can be ordered by the sequence number.

When a new log segment needs to be allocated, the log writer will lookup the log segment free pool or create a new log segment if free pool is empty. Free log segments will be fed into free pool after garbage collection of log segments.

Data size accounting

The valid data size is tracked in each of the flush delta in the page. For store level accounting, global data size counter is maintained. When a page segment becomes stale, the size is decremented from the global counter.

With log segment metadata, instead of store level flush data accounting, data size will be accounted in the log segment metadata. This introduces some additional requirements. When a page segment needs to be marked as stale, the data size should be decremented on the log segment metadata where the page segment resides. Currently, we do not maintain individual offsets of each of the each of the page segments. We need to additionally maintain sizes and offsets for all the page segments to correctly account for the data size. This adds extra memory overhead for swapout delta. Swapout delta needs to track all the page segment sizes and its offsets.

Garbage Collection

A background gc worker scans the metadata objects for all the segments periodically and generates a list of segments ordered by highest amount of garbage. The log cleaning workers picks up the segments and clean them and put back to the free segments pool.

After cleaning a segment, the segment cannot be put back to the free segments pool. During the relocation operations performed during the cleaning, it will be written to the lockfree flushbuffer of the tail of the log chain. If we happen to reuse the block that just got cleaned, we may overwrite the data on the durable storage before the old versions of the data blocks are relocated and rewritten to the durable storage. Hence, the segment should be marked as in use and marked and moved to free pool when the relocated segment is written to the durable storage.

When a segment is cleaned and moved to the free pool, the log chain is broken. This will become an issue during crash recovery. This problem can be eliminated by using checkpoints for recovery. We need to maintain log chain only from the point of checkpoint. The log chain relationship of the segments does not need to be maintained any segments written before the checkpoint.

The garbage collector should only consider the segments with seqno less than the current checkpoint sequence number as the candidates for garbage collection. Otherwise, we won’t be able to maintain the log chain.

Safe Log Segment Reclamation

When garbage collector cleans a segment, the segment cannot be immediately put back into the free pool. Since all the operations happen in lockfree manner, the segment needs to be parked in the safe memory reclaimer and put back to free pool once the reclaim session has ended.

Checkpointing

During checkpointing, the current logID needs to be stored. It will be the head log segment for log chain used for log replay during the recovery. Once a checkpoint is created, the garbage collection segment sequence number can be moved the checkpoint sequence number.

Crash Recovery

During recovery, Plasma needs to perform checkpoint based recovery. The log segment metadata structures needs to be rebuilt from each page during the recovery. The checkpointing also should store all the segment offsets and their sizes for each of the pages. Once index layer is rebuild after checkpoint, log chain replay has to be performed. One of the challenge for log chain is to identify the end of the log chain segment. Since we store log segment sequence number, it can be used to detect the end. The end segment would be the next segment in the chain having seq number not equal to curr_seq+1.

All the segments which are not referenced during the recovery will be added to the free segments pool.

Future in-place update scheme on segments

The segment based cleaning approach also enables to perform inplace updates to the segments with garbage. If we add sufficient additional metadata/bitmap per segment metadata object, we could perform inplace updates and avoid cleaning. The writer could choose a segment with garbage and overwrite the data and reuse it. This technique called threading can be found in log structured file systems. A hybrid log management scheme can be used. If the log is running above configured threshold fragmentation, the writers could perform inplace update writes if the cleaning is expensive and consumes more bandwidth.

Multiple logs for Hot/Cold page separation

Instead of keeping all pages to a single log, we could implement multiple logs and categorize them into hot and cold logs. When a segment is cleaned, all the survivor blocks are the blocks which are long lived. Garbage blocks have been rewritten to the tail of the log at some point. Survivor blocks can be treated as cold blocks and they can be moved to a different log based on simple statistics. This enables classification of hot and cold data blocks to different log and reduces the log fragmentation.

Faster validity check for value separation

Each log segment has an in-memory metadata object consisting of information required for performing efficient garbage aware cleaning. When a value needs to be checked against its validity, current scheme involves performing a lookup into the page with item key and verify if the value pointer for the item in the page points to the current value offset. This may involve disk reads when there is only little memory available for caching and it is expensive.

The item validity check can be significantly improved by using a bitmap in memory. Let's call it validation bitmap. A segment is written to disk only when it is full. When it is ready to be written, we know exactly how many value objects have been written to the current log segment. We can construct a value bitmap (1 bit per value) and store it in the in-memory segment object. When a value is written, we store the position information. Ie., n - for an nth value in the segment.

When a value is garbage collected from the page, it locates the segment in-memory meta object from the offset and sets the bitmap atomically. Position of the bit in bitmap is obtained from the value pointer. When the log cleaner runs, it uses the bitmap to check validity of the values instead of performing a lookup into the page.

If the 1MB segment contains 1024 1KB values, it would require 128 bytes to store the bitmap. ie., 128 MB per 1TB.

During checkpointing, the bitmap can be persisted to the disk. During the log replay after recovery, deleted objects after checkpointing and be incrementally updated. Implementation can take lazy approaches to update the bitmap. If a garbage collected value is not marked in the bitmap, it does not cause any functional incorrectness. It would need an additional page lookup to validate the value.

Backward compatibility and upgrade

The new log format and management breaks backward compatibility with the existing log management scheme. We will introduce this scheme only for newly created indexes.