Design Document - Chunky Logs

Requirements

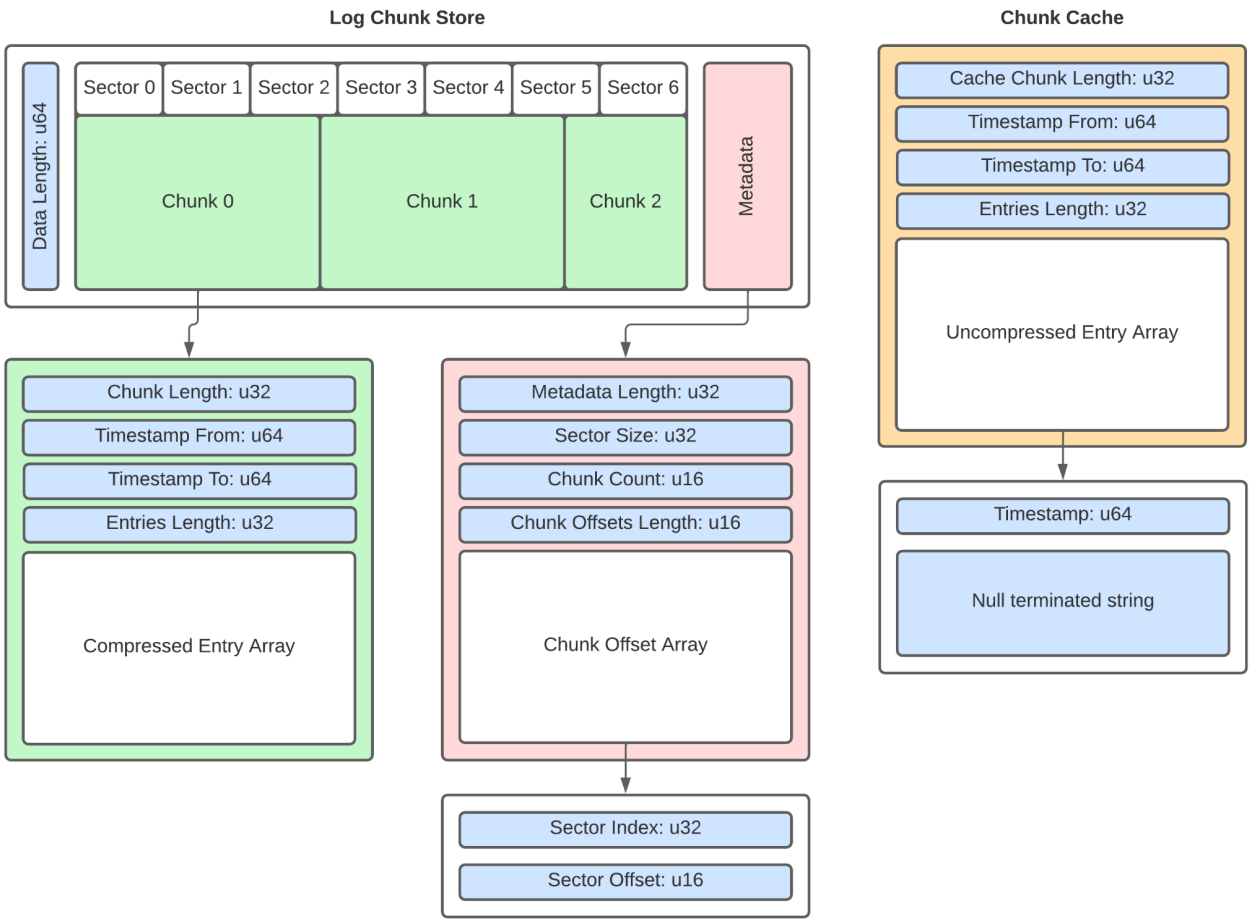
- Write efficient logging
- Cached writing with async flushes
- Chunked compression

Overview

Storage format for high frequency writes for temporally linear data. Data is written in compressed chunks, each chunk has n data elements. Data is initially written to a cache which is periodically flushed to a cache file. Once the cache file is full it is compressed and flushed to the data store.

Details

Structure



Initialising

A new file is created with data length set to 0. After this the metadata header is written in default layout [1]. In addition to this, a new cache file is created in default layout [2].

Loading

Initially the data length is read from the chunk store, then a relative seek of the data length is performed from the current position. From there the metadata length is read and used to process the metadata section.

Next, if a cache file exists, the length is read and used to load it as a mutable memory map. If the file does not exist, a new file is created with a size of 64192 bytes (192B header + 64KB entry array) with default layout [2].

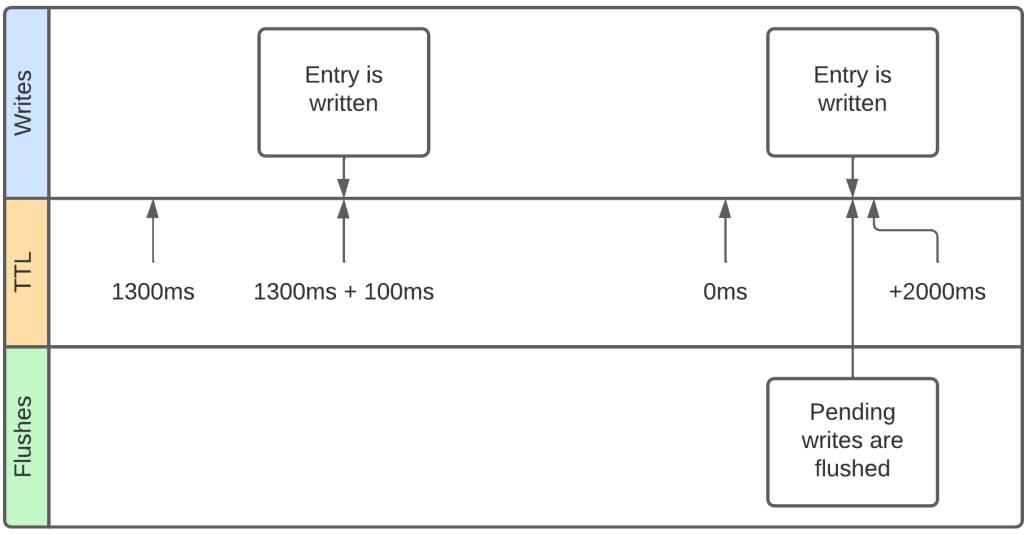
Caching

In order to reduce consistent file IO overhead, an intermediary cache is used to buffer writes and flush them asynchronously to the chunk store. The cache consists of a single file with a memory mapped entry buffer of 64KB. When this buffer is written to, it is periodically flushed asynchronously to the file. If the memory mapped portion of the file is filled containing less than 1000 entries, another 64KB is mapped ahead and writing continues.

Once the file has filled with 1000 entries, the entries array is compressed. The compressed data + header is written to the store on a separate thead and the cache file is restored to its default state. The memory mapped region is re-mapped over the re-initialised cache file.

Flushing

In order to make flushing consistent and periodic (beyond the aforementioned absolute cases of filled buffering), a persistent TTL is used to indicate when a set of writes will be flushed. Initially this TTL is set to 2 seconds and any value added to it will not increment it beyond 2 seconds. Every time a write is performed to the buffered cache, this TTL is incremented by 100ms. If the TTL expires before the next write, the current changes to the memory mapped cache will be flushed asynchronously and the TTL is reset with the new entry written as a pending change.



Using one of the following

```
MmapMut::flush_async(&self) -> Result<()>
MmapMut::flush_async_range(&self, offset: usize, len: usize) -> Result<()>
```

Chunks

Chunks store ordered timestamped entries, this is designed for continuous writing with efficient reachability. Entries are formatted in two sections, a 64 bit UNIX timestamp and a null terminated string. Note that strings should be escaped for storage, otherwise erroneous reads can occur when parsing entries.

Writing

Writes are separated into three different kinds. This allows for flexibility around how users handle creating and adding entries:

- Single entry
- Bulk
- Full chunk

Single Entry

Continuous writes are done on a single entry basis, these are performed via a writer module. Entries are written into the cache and flushed asynchronously to the chunk store for persistence.

Bulk

The writer module also supports providing queries in bulk. These will be written as bulk data in the cache. Note that if the bulk data exceeds the current memory buffer, it will be extended. Additionally, bulk writes beyond the current capacity of the cache will be split into two bulk writes.

Full Chunk

The last supported case of the writer is full chunk writing. This is particularly useful for custom writer handlers and buffering techniques. This API is fundamentally the same as the method used to insert chunks into the store from the cache. You can expect identical performance in that regard and can be used in two ways synchronous and asynchronous.

Reading Chunks

TODO

Searching Chunks

A search is considered finding an entry for a given timestamp or timestamps within a range.

Single Timestamp

First a binary search is performed across all chunk headers, where the desired timestamp is compared with the range inside the current header. Searching across chunks ceases when a matching chunk is found or none of them capture the range.

In the case that a matching chunk is found, it is decompressed and a secondary binary search is performed across the entries. If the timestamp matches the entry is returned.

Ranged Timestamp

A parallel binary search is performed with the start and end timestamps. If both searches fail, then nothing is returned. Otherwise there are three cases that emerge to handle

Case 1: Chunks found for both timestamps

In the first case, both searches resolve to chunks. If there is only one chunk in the range then it is decompressed and a further parallel binary search occurs on the entries. The resolved entries are then buffered and returned with the entry count.

If there is more than one chunk, the first chunk is decompressed and entries are returned and the indexes of the remaining chunks matching the range are returned.

Case 2: From timestamp only matches

When only the “from” timestamp matches, the first matching chunk is decompressed and the entries are buffered and returned. Alongside this, all chunk indices from the next chunk onwards.

Case 3: To timestamp only matches

If only the “to” timestamp matches, the first chunk in the store is decompressed and the entries are buffered and returned. Additionally, all chunk indices up to the matching “to” chunk are returned as well.

Appendix

Default Layouts

[1] Chunk Store Metadata Header

- Metadata length: 96u32
- Sector size: 1000u32
- Chunk count: 0u16
- Chunk offsets length: 0u16
- Chunk offsets array: empty of 0usize footprint

[2] Cache Chunk

- Cache chunk length: 64192u32
- Timestamp from: 0u32
- Timestamp to: 0u32
- Entries length: 0u32
- Entry array: empty with 64KB memory footprint