Prometheus TSDB (Part 3): Memory Mapping of Head Chunks from Disk

October 2, 2020 · 9 min read

Ganesh Vernekar
Software Engineer @ Grafana Labs

Introduction

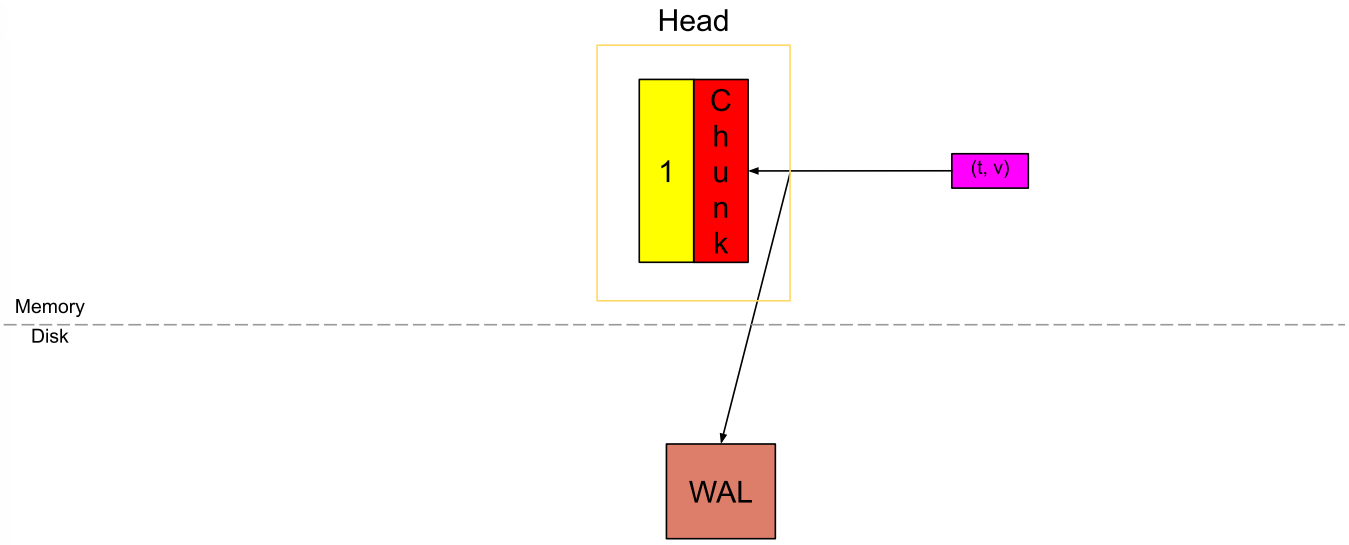
In the Part 1 of the TSDB blog series I mentioned that once a chunk is "full", it is flushed to the disk and memory mapped. This helps in reducing the memory footprint of the Head block and also helps speed up the WAL replay that we discussed in Part 2. We will be diving deeper into how this is designed in Prometheus in this blog post.

As this is a part of the Prometheus TSDB blog series that I am writing, you are recommended to read the Part 1 to know where these memory mapped chunks fit into TSDB (or the Head block) and Part 2 to understand the WAL replay.

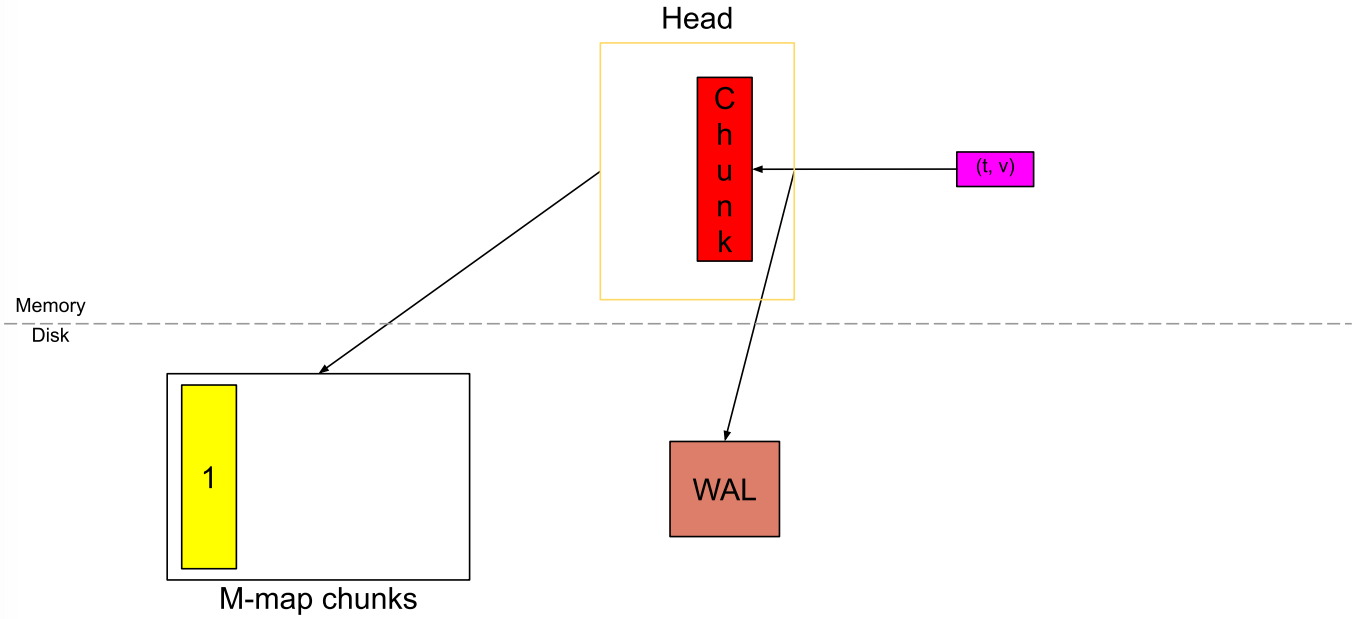
I have also given a KubeCon talk on this which explains this at a little higher level.

Writing these chunks

Recapping from Part 1, when a chunk is full, we cut a new chunk and the older chunks become immutable and can only be read from (the yellow block below).



And instead of storing it in memory, we flush it to disk and store a reference to access it later.



This flushed chunk is the memory-mapped chunk from disk. The immutability is the most important factor here else rewriting compressed chunks would have been too inefficient for every sample.

Format on disk

The format can also be found on GitHub.

The File

These chunks stay in its own directory called `chunks_head` and have a file sequence similar to WAL (except it starts with 1). For example:

```
data
├── chunks_head
│   ├── 000001
│   └── 000002
└── wal
    ├── checkpoint.000003
    │   ├── 000000
    │   └── 000001
    ├── 000004
    └── 000005
```

The max size of the file is kept at 128MiB. Now diving deeper into a single file, the file contains a header of 8B.

`Magic Number` is any number that can uniquely identify the file as a memory-mapped head chunks file. As I implemented this feature, I set it to my birth date :). `Chunk Format` tells us how to decode the chunks in the file. The extra padding is to allow any future header options that we might require.

After the file header, what follows is chunks.

Chunks

A single chunk looks like this

The `series ref` is the same series reference that we talked about in Part 2, it is the series id used to access the series in the memory. The `mint` and `maxt` are the minimum and maximum timestamp seen in the samples of the chunk. `encoding` is the encoding used to compress the chunks. `len` is the number of bytes that follow from here and `data` are the actual bytes of the compressed chunk.

`CRC32` is the checksum of the above content of the chunk used to check the integrity of the data.

Reading these chunks

For every chunk, the Head block stores the mint and maxt of that chunk along with a reference in the memory to access it.

The reference is 8 bytes long. The first 4 bytes tell the file number in which the chunk exists, and the last 4 bytes tell the offset in the file where the chunk starts (i.e. the first byte of the `series ref`). If the chunk was in the file `00093` and the `series ref` starts at byte offset `1234` in the file, then the reference of that chunk would be `(93 << 32) | 1234` (left shift bits and then bitwise OR).

We store the `mint` and `maxt` in Head so that we can select the chunk without having to look at the disk. When we do have to access the chunk, we only access the encoding and the chunk data using the reference.

In the code, the file looks like yet another byte slice (one slice per file) and accessing the slice at some index to get the chunk data while the OS maps the slice in the

memory to the disk under the hood. Memory-mapping from disk is an OS feature which fetches only the part of disk into memory which is being accessed and not the entire file.

Replaying on startup

In Part 2 we talked about WAL replay where we replay each individual sample to re-create the compressed chunk. Now that we have the compressed full chunks on disk, we don't need to go through recreation of these chunks while we still need to create chunks from WAL which were not full. Now with these memory-mapped chunks from disk, the replay happens as follows.

At startup, first we iterate through all the chunks in the `chunks_head` directory and build a map of `series ref -> [list of chunk references along with mint and maxt belonging to this series ref]` in the memory.

We then continue with the WAL replay as described in Part 2 but with few modification:

- When we come across the `Series` record, after creation of the series, we look for the series reference in the above map and if any memory-mapped chunks exist, we attach that list to this series.
- When we come across the `Samples` record, if the corresponding series for the sample has any memory-mapped chunks and if the sample falls into the time ranges that it covers, then we skip the sample. If it does not, then we ingest that sample into the Head block.

Enhancements that this brings in

What's the use of this additional complexity while we could get away with storing chunks in the memory and the WAL? This feature was added recently in 2020, so let's see what this brings in. (You can see some benchmark graphs in this Grafana Labs blog post)

Memory savings

If you had to store the chunk in the memory, it can take anywhere between 120 to 200 bytes (or even more depending on compressibility of the samples). Now this is replaced with 24 bytes - 8 bytes each of chunks reference, min time, and max time of the chunk.

While this may sound like 80-90% reduction in memory, the reality is different. There are more things that the Head needs to store, like the in-memory index, all the symbols (label values), etc, and other parts of TSDB that take some memory.

In the real world, we can see a 15-50% reduction in the memory footprint depending on the rate at which samples are being scraped and the rate at which new series are being created (called "churn"). Another thing to note is that, if you are running some queries which touch a lot of these chunks on disk, then they need to be loaded into the memory to be processed. So it's not an absolute reduction in peak memory usage.

Faster startup

The WAL replay is the slowest part of startup. Mainly, (1) decoding of WAL records from disk and (2) rebuilding the compressed chunks from individual samples, are the slow parts in the replay. The iteration of memory-mapped chunks is relatively fast.

We cannot avoid decoding of records as we need to check all the records. As you saw above in the replay, we are skipping the samples which are in the memory-mapped chunks range. Here we avoid re-creating those full compressed chunks, hence save some time in the replay. It has been seen to reduce the startup time by 15-30%.

Garbage collection

The garbage collection in memory happens during the Head truncation where it just drops the reference of the chunks which is older than the truncation time T. But the files are still present on the disk. As with WAL segments, we also need to delete old m-mapped files regularly.

For every memory-mapped chunk file present (which means also open in TSDB), we store in the memory the absolute maximum time among all the chunks present in the file. For the live file (the one in which we are currently writing the chunks), we update this maximum time in the memory as and when we are adding new chunks. During a

restart, as we iterate all the memory-mapped chunks, we restore the maximum time of the files in the memory there.

So when the Head truncation is happening for data before time `T`, we call truncation on these files for time `T`. The files whose maximum times is below `T` (except the live file) are deleted at this point while preserving the sequence (if the files were `5, 6, 7, 8` and if files `5` and `7` were beyond time `T`, only `5` is deleted and the remaining sequence would be `6, 7, 8`).

After truncation, we close the live file and start a new one because in low volume and small setups, it might take a lot of time to reach the max size of the file. So rotating the files here will help deletion of old chunks during the next truncation.

Code reference

`tsdb/chunks/head_chunks.go` has all the implementation of writing chunks to disk, accessing it using a reference, truncation, handling the files, and way to iterate over the chunks.

`tsdb/head.go` uses the above as a black box to memory-map its chunks from disk.

More blog posts to expect in the future on TSDB

1. Persistent blocks along with its index.
2. Compaction.
3. Snapshotting of in-memory chunks for faster restarts when it's in.
4. Queries in TSDB.

I will update this section with links to the blog posts whenever they are up. If I have missed anything in this list, let me know!

Tags: Prometheus TSDB

Next Post
Prometheus TSDB (Part 2): WAL and Checkpoint »

GitHub

Twitter

LinkedIn

Instagram

Flickr

Blog

RSS feed

Atom feed