

Write-Ahead Logs



- [Impetus](#)
 - [Strategy](#)
 - [Segments](#)
 - [Truncation](#)
 - [Checkpoints](#)
 - [Queueing Checkpoint operations](#)
 - [WAL Record Types](#)
 - [Streams](#)
 - [Logs](#)
 - [Restoration](#)
 - [Deployment](#)
 - [Implementation goals](#)
 - [Alternatives](#)
 - [Use the Cortex WAL](#)
 - [Don't build checkpoints from memory, instead write new WAL elements](#)
-

Author: Owen Diehl - [owen-d](#) ([Grafana Labs](#))

Date: 30/09/2020

Impetus

Loki already takes numerous steps to ensure the persistence of log data, most notably the use of a configurable replication factor (redundancy) in the ingesters. However, this still leaves much to be desired in persistence guarantees, especially for single binary deployments. This proposal outlines a write ahead log (WAL) in order to complement existing measures by allowing storage/replay of incoming writes via local disk on the ingester components.

Strategy

We suggest a two pass WAL implementation which includes an initial recording of accepted writes (`segments`) and a subsequent checkpointing (`checkpoints`) which coalesces the first pass into more efficient representations to speed up replaying.

Segments

Segments are the first pass and most basic WAL. They store individual records of incoming writes that have been accepted and can be used to reconstruct the in memory state of an ingester without any external input. Each segment is some multiple of 32kB and upon filling one segment, a new segment is created. Initially Loki will try 256kb segment sizes, readjusting as necessary. They are sequentially named on disk and are automatically created when a target size is hit as follows:

```
data
├─ wal
│  ├── 000000
│  ├── 000001
│  └─ 000002
```

Truncation

In order to prevent unbounded growth and remove operations which have been flushed to storage from the WAL, it is regularly truncated and all but the last segment (which is currently active) are deleted at a configurable interval (`ingester.checkpoint-duration`). This is where checkpoints come into the picture.

Checkpoints

Before truncating the WAL, we advance the WAL segments by one in order to ensure we don't delete the currently writing segment. The directory will look like:

```
data
├─ wal
│  ├── 000000
│  ├── 000001
│  ├── 000002 <- likely not full, no matter
│  └─ 000003 <- newly written, empty
```

Each in memory stream is iterated across an interval, calculated by `checkpoint_duration / in_memory_streams` and written to the checkpoint. After the checkpoint completes, it is moved from its temp directory to the `ingester.wal-dir`, taking the name of the last segment before it started (`checkpoint.000002`) and then all applicable segments (`000000`, `000001`, `000002`) and any previous checkpoint are deleted.

Afterwards, it will look like:

```
data
└─ wal
   └─ checkpoint.000002 <- completed checkpoint
      └─ 000003 <- currently active wal segment
```

Queueing Checkpoint operations

It's possible that one checkpoint operation will start at the same time another is running. In this case, the existing checkpoint operation should disregard its internal ticker and flush its series as fast as possible. Afterwards, the next checkpoint operation can begin. This will likely create a localized spike in IOPS before the amortization of the following checkpoint operation takes over and is another important reason to run the WAL on an isolated disk in order to mitigate noisy neighbor problems. After we've written/moved the current checkpoint, we reap the old one.

WAL Record Types

Streams

A `Stream` record type is written when an ingester receives a push for a series it doesn't yet have in memory. At a high level, this will contain

```
type SeriesRecord struct {
    []UserID  string
    []Labels  labels.Labels
    []Fingerprint uint64 // label fingerprint
}
```

Logs

A `Logs` record type is written when an ingester receives a push, containing the fingerprint of the series it refers to and a list of `(timestamp, log_line)` tuples, *after* a `Stream` record type is written, if applicable.

```
type LogsRecord struct {
    []UserID  string
    []Fingerprint uint64 // label fingerprint for the series these logs refer to
    []Entries []logproto.Entry
}
```

Restoration

Replaying a WAL is done by loading any available checkpoints into memory and then replaying any operations from successively named segments on top (checkpoint.000003 -> 000004 -> 000005, etc). It's likely some of these operations will fail because they're already included in the checkpoint (due to delay introduced in our amortizations), but this is ok -- we won't lose any data, only try to write some data twice, which will be ignored.

Deployment

Introduction of the WAL requires that ingesters have persistent disks which are reconnected across restarts (this is a good fit for StatefulSets in Kubernetes). Additionally, it's recommended that the WAL uses an independent disk such that it's isolated from being affected by or causing noisy neighbor problems, especially during any IOPS spike(s).

Implementation goals

- Use underlying prometheus wal pkg when possible for consistency and to mitigate undifferentiated heavy lifting. Interfaces handle page alignment and use []byte.
 - Ensure this package handles arbitrarily long records (log lines in Loki's case).
- Ensure our in memory representations can be efficiently moved to/from []byte in order to generate conversions for fast/efficient loading from checkpoints.
- Ensure chunks which have already been flushed to storage are kept around for ingester.retain-period, even after a WAL replay.

Alternatives

Use the Cortex WAL

Since we're not checkpointing from the WAL records but instead doing a memory dump, this isn't bottlenecked by throughput but rather memory size. Therefore we can start with checkpointing by duration rather than accounting for throughput as well. This makes the proposed solution nearly identical to the Cortex WAL approach. The one caveat is that wal segments will accrue between

checkpoint operations and may constitute a large amount of data (log throughput varies). We may eventually consider other routes to handle this if duration based checkpointing proves insufficient.

Don't build checkpoints from memory, instead write new WAL elements

Instead of building checkpoints from memory, this would build the same efficiencies into two distinct WAL Record types: `Blocks` and `FlushedChunks`. The former is a record type which will contain an entire compressed block after it's cut and the latter will contain an entire chunk + the sequence of blocks it holds when it's flushed. This may offer good enough amortization of writes because block cuts are assumed to be evenly distributed and chunk flushes have the same property and use jitter for synchronization.

This could be used to drop WAL records which have already elapsed the `ingester.retain-period`, allowing for faster WAL replays and more efficient loading.

```
type FlushRecord struct {
    Fingerprint uint64 // labels
    FlushedAt uint64 // timestamp when it was flushed, can be used with `ingester.retain-period`
    to either keep or discard records on replay
    LastEntry logproto.Entry // last entry included in the flushed chunk
}
```

It would also allow building checkpoints without relying on an ingester's internal state, but would likely require multiple WALs, partitioned by record type in order to be able to iterate all `FlushedChunks` -> `Blocks` -> `Series` -> `Samples` such that we could no-op the later (lesser priority) types that are superseded by the former types. The benefits do not seem worth the cost here, especially considering the simpler suggested alternative and the extensibility costs if we need to add new record types if/when the ingester changes internally.

```
[[[ #2
[ [[ [ 2026-10-31 13:43:51 UTC
[ [[ [ 2026-10-31 13:43:52 UTC
```