Write Ahead Log

- Disclaimer and WAL nuances
  - Backpressure
  - Metrics
- Changes to deployment
- Changes in lifecycle when WAL is enabled
- Disk space requirements
- Migrating from stateless deployments
- How to scale up/down
  - Scale up
  - Scale down
- Additional notes
  - Kubernetes hacking
    - Scaling Down Using /flush_shutdown Endpoint and Lifecycle Hook
  - Non-Kubernetes or baremetal deployments

Ingesters temporarily store data in memory. In the event of a crash, there could be data loss. The Write Ahead Log (WAL) helps fill this gap in reliability.

The WAL in Grafana Loki records incoming data and stores it on the local file system in order to guarantee persistence of acknowledged data in the event of a process crash. Upon restart, Loki will "replay" all of the data in the log before registering itself as ready for subsequent writes. This allows Loki to maintain the performance and cost benefits of buffering data in memory *and* durability benefits (it won't lose data once a write has been acknowledged).

This section will use Kubernetes as a reference deployment paradigm in the examples.

Disclaimer and WAL nuances

The Write Ahead Log in Loki takes a few particular tradeoffs compared to other WALs you may be familiar with. The WAL aims to add additional durability guarantees, but *not at the expense of availability*. Particularly, there are two scenarios where the WAL sacrifices these guarantees.

1. Corruption/Deletion of the WAL prior to replaying it
   In the event the WAL is corrupted/partially deleted, Loki will not be able to recover all of its data. In this case, Loki will attempt to recover any data it can, but will not prevent Loki from starting.
   You can use the Prometheus metric `loki_ingester_wal_corruptions_total` to track and alert when this happens.
2. No space left on disk
   By default, Loki monitors disk usage on the volume that stores the WAL. If usage reaches 90% of capacity, the ingester rejects new writes instead of accepting data it cannot persist. This check runs every 10 seconds and measures the entire filesystem that holds the WAL directory, not just the size of the WAL files. Because of this, you should not share the WAL volume with other data. Writes resume automatically once disk usage drops back below the threshold. You can change the threshold with the `--ingester.wal-disk-full-threshold` flag, or set it to `0` to disable this protection. This throttling behavior isn't available on Windows.
   If you disable this threshold, or if a write reaches the disk in the short window before the check runs, the underlying disk can still become completely full. In that case, Loki will not fail the incoming write, but neither will it log the entry to the WAL. In this case, the persistence guarantees across process restarts will not hold for that entry.
   You can use the Prometheus metric `loki_ingester_wal_disk_full_failures_total` to track and alert when this happens. This counter increases both when disk usage crosses the throttling threshold and when a WAL write fails because the disk is completely full, so a nonzero value means the WAL disk needs attention, but it does not count each individual rejected write. You can also monitor the gauge `loki_ingester_wal_disk_usage_percent` to see current disk usage and how close it is to the threshold.

Backpressure

The WAL also includes a backpressure mechanism to allow a large WAL to be replayed within a smaller memory bound. This is helpful after bad scenarios (i.e. an outage) when a WAL has grown past the point it may be recovered in memory. In this case, the ingester will track the amount of data being replayed and once it's passed 90% of the `ingester.wal-replay-memory-ceiling` threshold, will flush to storage. When this happens, it's likely that the Loki attempt to deduplicate chunks via content addressable storage will suffer. We deemed this efficiency loss an acceptable tradeoff considering how it simplifies operation and that it should not occur during regular operation (rollouts, rescheduling) where the WAL can be replayed without triggering this threshold. Setting `ingester.wal-replay-memory-ceiling` to `0` disables backpressure entirely.

Metrics

The following metrics are available for monitoring the WAL:

- `loki_ingester_wal_corruptions_total`: Total number of WAL corruptions encountered

- `loki_ingester_wal_disk_full_failures_total`: Increments when disk usage crosses the `--ingester.wal-disk-full-threshold` throttling threshold, and when a WAL write fails because the disk is completely full. It does not count each individual throttled write.
- `loki_ingester_wal_disk_usage_percent`: Current disk usage, from 0.0 to 1.0, for the volume that stores the WAL. This is only reported when `--ingester.wal-disk-full-threshold` is greater than `0`.
- `loki_ingester_wal_records_logged_total`: Counter for WAL records logged
- `loki_ingester_wal_logged_bytes_total`: Total bytes written to WAL
- `loki_ingester_wal_replay_active`: Set to 1 while an ingester is replaying its WAL and checkpoint on startup, 0 otherwise
- `loki_ingester_wal_replay_duration_seconds`: Time taken to replay the checkpoint and the WAL

Changes to deployment

1. Since ingesters need to have the same persistent volume across restarts/rollout, all the ingesters should be run on StatefulSet with fixed volumes.
2. The following flags are relevant when deploying with a persistent volume:
   - `--ingester.wal-enabled` enables writing to WAL during ingestion. It defaults to `true`, so this typically doesn't need to be set explicitly.
   - `--ingester.wal-dir` sets the directory where the WAL data should be stored and/or recovered from. Note that this should be on the mounted volume.
   - `--ingester.checkpoint-duration` to the interval at which checkpoints should be created.
   - `--ingester.wal-replay-memory-ceiling` (default 4GB) may be set higher/lower depending on your resource settings. It handles memory pressure during WAL replays, allowing a WAL many times larger than available memory to be replayed. This is provided to minimize reconciliation time after very bad situations, i.e. an outage, and will likely not impact regular operations/rollouts *at all*. We suggest setting this to a high percentage (~75%) of available memory.
   - `--ingester.wal-disk-full-threshold` (default 0.90) sets the fraction of disk usage (0.0 to 1.0) at which the ingester throttles incoming writes to protect WAL durability. Set to `0` to disable this protection. For details, refer to Disclaimer and WAL nuances.

Changes in lifecycle when WAL is enabled

Flushing of data to chunk store during rollouts or scale down is disabled. This is because during a rollout of statefulset there are no ingesters that are simultaneously leaving and joining, rather the same ingester is shut down and brought back again with updated config. Hence flushing is skipped and the data is recovered from the WAL. If you need to ensure that data is always flushed to the chunk store when your pod shuts down, you can set the `--ingester.flush-on-shutdown` flag to `true`.

Disk space requirements

Based on tests in real world:

- Numbers from an ingester with 5000 series ingesting ~5mb/s.
- Checkpoint period was 5mins.
- disk utilization on a WAL-only disk was steady at ~10-15GB.

You should not target 100% disk utilization.

Migrating from stateless deployments

The ingester *Deployment without WAL* and *StatefulSet with WAL* should be scaled down and up respectively in sync without transfer of data between them to ensure that any ingestion after migration is reliable immediately.

Let's take an example of 4 ingesters. The migration would look something like this:

1. Bring up one stateful ingester `ingester-0` and wait until it's ready (accepting read and write requests).
2. Scale down the old ingester deployment to 3 and wait until the leaving ingester flushes all the data to chunk store.
3. Once that ingester has disappeared from `kc get pods ...`, add another stateful ingester and wait until it's ready. Now you have `ingester-0` and `ingester-1`.
4. Repeat step 2 to reduce remove another ingester from old deployment.
5. Repeat step 3 to add another stateful ingester. Now you have `ingester-0 ingester-1 ingester-2`.
6. Repeat step 4 and 5, and now you will finally have `ingester-0 ingester-1 ingester-2 ingester-3`.

How to scale up/down

Scale up

Scaling up is same as what you would do without WAL or StatefulSets. Nothing to change here.

Scale down

When scaling down, we must ensure existing data on the leaving ingesters are flushed to storage instead of just the WAL. This is because we won't be replaying the WAL on an ingester that will no longer exist and we need to make sure the data is not orphaned.

Consider you have 4 ingesters `ingester-0 ingester-1 ingester-2 ingester-3` and you want to scale down to 2 ingesters, the ingesters which will be shut down according to StatefulSet rules are `ingester-3` and then `ingester-2`.

Hence before actually scaling down in Kubernetes, port forward those ingesters and hit the `/ingester/shutdown?flush=true` endpoint. This will flush the chunks and remove itself from the ring, after which it will register as unready and may be deleted.

After hitting the endpoint for `ingester-2 ingester-3`, scale down the ingesters to 2.

Also you can set the `--ingester.flush-on-shutdown` flag to `true`. This enables chunks to be flushed to long-term storage when the ingester is shut down.

Additional notes

Kubernetes hacking

StatefulSets are significantly more cumbersome to work with, upgrade, and so on. Much of this stems from immutable fields on the specification. For example, if one wants to start using the WAL with single store Loki and wants separate volume mounts for the WAL and the TSDB, you may see immutability errors when attempting updates the Kubernetes StatefulSets.

In this case, try `kubectl -n <namespace> delete sts ingester --cascade=false`. This will leave the Pods alive but delete the StatefulSet. Then you may recreate the (updated) StatefulSet and one-by-one start deleting the `ingester-0` through `ingester-n` Pods *in that order*, allowing the StatefulSet to spin up new pods to replace them.

Scaling Down Using `/flush_shutdown` Endpoint and Lifecycle Hook

1. **StatefulSets for Ordered Scaling Down**: The Loki ingesters should be scaled down one by one, which is efficiently handled by Kubernetes StatefulSets. This ensures an ordered and reliable scaling process, as described in the Deployment and Scaling Guarantees documentation.
2. **Using PreStop Lifecycle Hook**: During the Pod scaling down process, the PreStop lifecycle hook triggers the `/flush_shutdown` endpoint on the ingester. This action flushes the chunks and removes the ingester from the ring, allowing it to register as unready and become eligible for deletion.
3. **Using terminationGracePeriodSeconds**: Provides time for the ingester to flush its data before being deleted, if flushing data takes more than 30 minutes, you may need to increase it.
4. **Cleaning Persistent Volumes**: Persistent volumes are automatically cleaned up by leveraging the enableStatefulSetAutoDeletePVC feature in Kubernetes.

By following the above steps, you can ensure a smooth scaling down process for the Loki ingesters while maintaining data integrity and minimizing potential disruptions.

Non-Kubernetes or baremetal deployments

- When the ingester restarts for any reason (upgrade, crash, etc), it should be able to attach to the same volume in order to recover back the WAL and tokens.
- 2 ingesters should not be working with the same volume/directory for the WAL.
- A rollout should bring down an ingester completely and then start the new ingester, not the other way around.

□□□ #2
□ □□□ □□□ 2026-10-31 13:47:12 UTC
□ □□□ □□□ 2026-10-31 13:47:13 UTC