Fault tolerant writer for Timeline v2

Requirement

A timeline writer that can be resistant to backend storage down time and timeline collector failures. During the failure of either sides, the writer should not lose data on its best effort.

Overview

A staged timeline writer that can store to a redo-log storage system (file system or HDFS) ,and later replay timeline entities from the log, if the backend storage (HBase) becomes available. The writer is able to reclaim spaces used by redo-logs after the data is written to the backend storage.

Design

The fault tolerant writer is consist of two storage layer implementations: a file system based logger (redo-logger) that will temporarily buffer incoming timeline entities and an HBase timeline writer (storage writer) that will asynchronously write timeline entities to HBase.

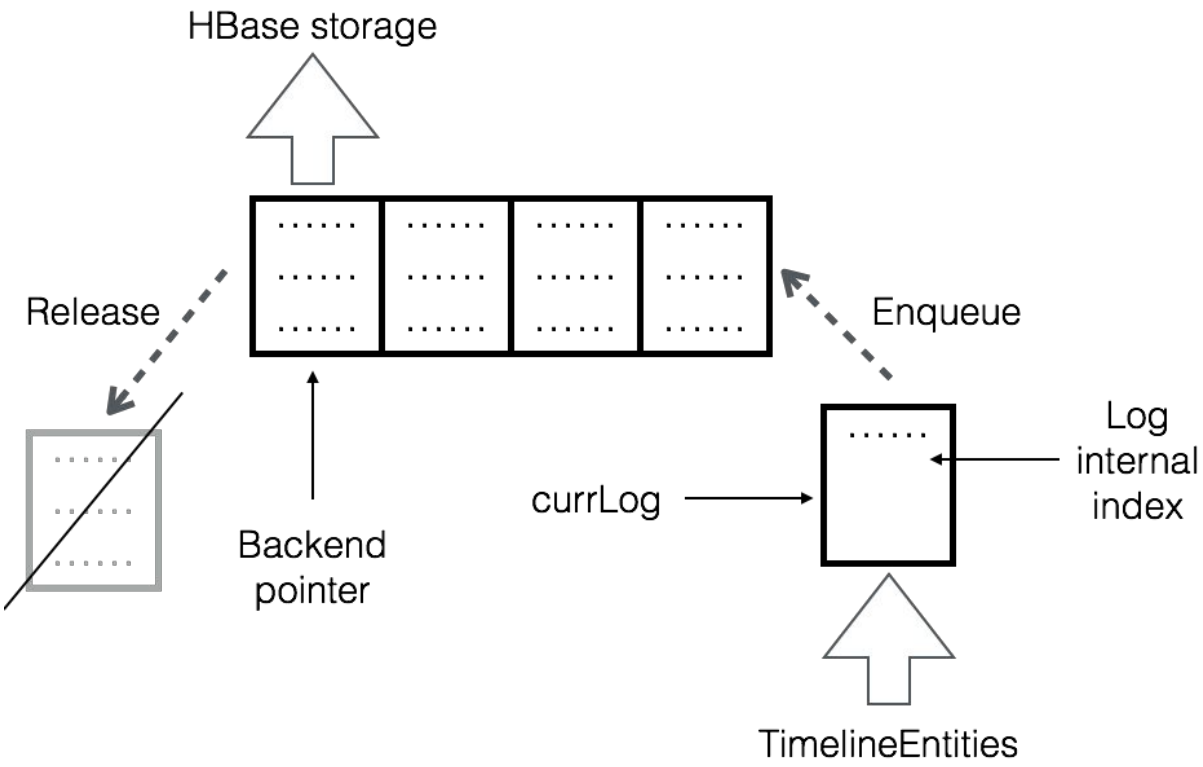
The two writers synchronize through a queue of redo-log segments. The redo-logger can persist the incoming timeline entities onto a file-system based redo-log segment. When one log segment reaches a predefined size, or a time-trigger, or an explicit flush call happens, it is published to a log queue. The storage writer waits on the log queue, dequeue log segments from the queue, write all timeline entities to the backend storage, and release the segment. The redo-logger and the storage writer never work on the same redo-log segment.

Implementation

The fault tolerant writer can be a composition of two timeline storage implementations. For now we're targeting on a new file system logger as the redo-logger, and the HBase timeline writer as the storage writer. Each redo-log segment may be a separate file under the same root directory.

The fault tolerant writer manages the overall status through 3 pointers: a pointer to the redo-log segment the logger is currently working on, a pointer to the redo-log segment that the storage writer is writing back, and a pointer to the current working index within a log segment that the logger is working on.

The storage writer asynchronously dequeue log segments from the queue, read out all available timeline entities in the segment through the reader interface of the redo-logger, write all of them back into storage, and remove the log segment (in our scenario we can simply delete the segment file).



Discussion

Backend storage failures

If the backend storage system is down, the storage writer may fail or block on an entity write. In this case, we are not losing data because the timeline entity has already been persisted to the redo-log, which are yet to be released. The storage writer may wait and retry the write until the backend storage system is online again.

Collector failures and rolling upgrades

Since we store most of our data on disk, we can rebuild the status of the writer after a collector crash and rolling upgrade (except for the log's internal pointer, which we may simply ignore and initialize a new log). We can then "replay" any data that has already been committed to the log queue, just as the failures/upgrades has never happen. The only unsafe data during this replay is in the very first redo-log segment in the queue, since upon the failure the storage writer may have already started writing part of this log. Therefore, if we directly replay all writes in the first redo-log segment in the queue, we may override some previous writes with stale data, which may cause inconsistent state in aggregation results. To avoid this situation, we do not replay ***metrics*** in the first log of the log queue if the writer is created with a non-empty log queue--only life-cycle events, config updates etc are replayed.

Local disk space running out

We propose to use a log queue, rather than a circular buffer, to store ongoing redo-log segments. The advantage of a linear-shaped log queue is that normally it will not lose data (unlike a ring buffer which may lose data if the number of incoming segments is greater than the ring buffer size). The disadvantage of the log queue is that under rare circumstances, if the backend storage is offline for an excess amount of time, it may use up disk space.

Note that currently we're launching writers for each CollectorManager on each NM. If the average life cycle of a YARN application is ~30 secs and the average size of an application is ~1k containers, in every minute there will be around 1k * 2 containers created, and 1k * 2 containers finished. This will modify 4k timeline entities. The size of each change is relatively small, and we assume it's about 1k conservatively. So every minutes there will be about 4M data generated for container lifecycle events. In every 5-15 seconds, an AM may generate a timeline entity to report metrics. So on a normal node with < 10 active AMs running, there will be at most 20 * 10 = 200 timeline entities generated for storing metrics (we only store metrics on a per-app level). Suppose each update contains 50k data (~100 metrics with ~500 bytes for each), space consumption will be about 10M. With a 3 GB disk space budget, this could hold a backend storage failure for around 3 hours. We may employ better compactions to improve space efficiency as well.

Local disk failures

If we implement the buffer writer with a local file system, on disk failures we cannot recover our log-writer. However, we do not foresee any significant problems to implement an HDFS based buffer writer if disk failure is a significant problem.

Consistency

Readers may only read out data from the backend storage systems. So some write operations that successfully returned on the writer side may not be seen immediately on the reader side. Right now we plan to implement a thread that periodically flush buffer. Notice that since we write back through a FIFO queue, we will not override new writes with old writes.

Batched writes

We now write to the backend storage system with a tunable buffer size. In general, increase the buffer size increase the latency between a write call returns and its value is visible to reads. The flush thread can perform flush every 2-5 seconds. By default some buffer sizes around 200-500 (85 entities/second * 2 to 85 * 5) may work.