[HBASE-13408] HBase In-Memory Memstore Compaction: Design Document

July 14, 2015

Eshcar Hillel (eshcar@yahoo-inc.com)
Anastasia Braginsky (anastas@yahoo-inc.com)

Motivation

A *store* unit holds a column family in a region, where the *memstore* is its in-memory component. The memstore absorbs all updates to the store; from time to time these updates are flushed to a file on disk, where they are compacted. Unlike disk components, the memstore is not compacted until it is written to the filesystem and optionally to block-cache. This may result in underutilization of the memory due to duplicate entries per row, for example, when hot data is continuously updated. In turn, this may slow down data retrieval as the data sinks to disk very fast in this scenario.

The current default implementation of memstore absorbs all updates in a dedicated *active set* data structure. Insert and modify operations add a new record to this set, delete operations add tombstone records to it. Upon flush updates are blocked while executing the *prepare-for-flush* phase in which the active set shifts to being a snapshot and a new active set is created to serve new updates. This quick preparation phase is protected by an exclusive region lock, `updatesLock`. Once the flush is completed the snapshot is discarded.

A region triggers a flush of its stores based on two basic conditions: (1) memstore size of a region exceeds the *flush-size,* (2) size of all memstores in a region server exceed a *global size*. In addition, updates may be blocked through the entire duration of the flush if the global memstore size or the memstore size of a specific region exceed upper limits (like blocking-flush-size).

Generally, the faster the data is accumulated in memory, more flushes are triggered, the data sinks to disk more frequently, slowing down retrieval of data, even if very recent.
In high-churn workloads, compacting the memstore can help maintain the data in memory, and thereby speed up data retrieval.

Proposal

We suggest a new *compacted memstore* with the following principles:

1. The data is kept in memory for as long as possible, by periodically compacting the memstore content.
2. The non-active memstore data is either compacted or in process of being compacted
3. Allow a *force flush mode,* which may interrupt an in-progress compaction and force a flush of part of the memstore, e.g., before updates are blocked due to memstore size.

The in-memory memstore compaction solution may help in some scenarios, however it might also add unnecessary overhead in other scenarios without any performance gains, like when there are no in-memory duplicate records most of the time. Therefore, we suggest applying this optimization *only to in-memory column families*. In addition to the reserved space these columns have in block-cache, the memstore compaction can increase the probability of the columns to reside only in-memory.

Flush condition #1 (flush-size) is relaxed, since the memstore size might exceed the flush size, but should not trigger a flush as it is about to be compacted.

High-level design

Our solution incrementally modifies the current design. In order to easier assimilate the new concept of in-memory compaction, we reconstruct some building blocks of a memstore. We suggest to encapsulate the common parts such as the active set and the snapshot in an abstract memstore.The existing default memstore and the new compacted memstore extend the abstract memstore. In addition, we encapsulate all kv-set and memory management services under one entity. The changes are threefold:

Memstore structure and behaviour

We propose to add a new memstore implementation which supports its in-memory compaction. The new entity of a *compaction pipeline* is added to the active set and the snapshot data structures. The compaction pipeline consists of a list of the read-only kv-sets that are subject to compaction.

The semantics of the prepare-for-flush phase is changed. Instead of shifting the current active kv-set to snapshot, the active kv-set is pushed into the pipeline. Like the snapshot, all pipeline components are read-only; updates only affect the active kv-set. To ensure this property we take advantage of the existing blocking mechanism -- the active kv-set is pushed to the pipeline while holding `updatesLock` in exclusive mode.

Memstore Scanners

This new memstore design means that a memstore scanner may need to scan more than two kv-set components. Instead of scanning only the active set and the snapshot, the new memstore scanner scans also the pipeline components.

Memstore Compaction

Periodically, a compaction is applied at the background to all pipeline components resulting in a single read-only component. The “old” components are discarded when no scanner is reading them. If the memory buffer pool mechanism is enabled, the memory can be reused immediately after the last scanner finishes. Currently, each flush triggers a compaction request, which results in dispatching the compaction, unless the compaction is already ongoing. Alternatively, as future enhancement, the compactor may decide whether or not to run the memstore compaction based on different policies: timeout, pipeline size, estimating memstore duplication ratio is high, etc. In a force flush mode, any in-progress compaction is interrupted, and the component at the tail of the compaction pipeline shifts to being the snapshot, which is later flushed to disk.

Memstore compaction is similar to minor compaction in the sense that it works only on a subrange of the history and therefore needs to keep tombstones records. However, the multiversion diversity of keys is flatten for single version, using the same logic as it is done when snapshot is flushed. To get more out of memstore compaction, as a future enhancement it is possible to consult the hfiles bloom filters to identify record history on disk, and decide if it is safe to remove the tombstone.

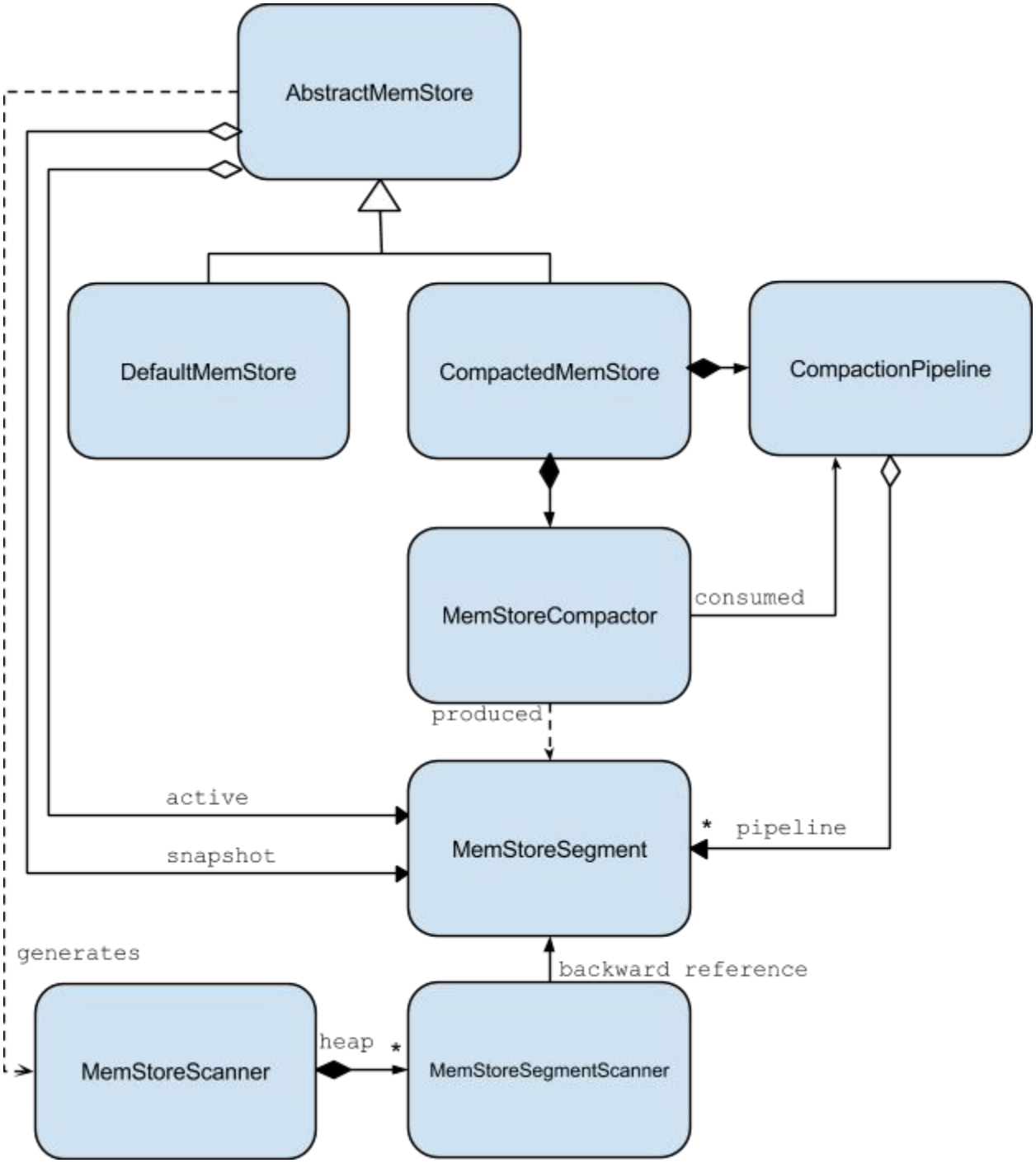
Flush policy

An additional force-flush-size is defined as the average of flush-size and blocking-flush-size. For example, if the blocking-flush-size is, say, 4 times the flush size (as in our experiments), then the force-flush-size is 2.5 times the flush-size.

The counter measuring the region memstore size is divided into 2 counters: one measuring the size of the active segments and the other for the additional memstore size (e.g., pipeline segments). The total memstore size is the sum of these two counters. If the total size exceeds the force-flush-size all stores in the region are set to force flush mode, and a flush request is triggered. Otherwise, if the memstore (active) size exceeds the flush-size a flush is triggered (without setting force flush mode). In addition, when forcing flush the method takes a boolean flag parameter indicating whether the flush is allowed to wait for memstore compaction to complete or to force the flush even if a compaction is in progress. Specifically, when a region server is being closed we prefer not to wait for the compaction but to immediately force the flush to disk.

Low-level design

The Class Diagram



Memstore design

[new class] **AbstractMemStore**.
An abstract class, which implements the behaviour shared by all concrete memstore instances.

DefaultMemStore extends AbstractMemStore
The DefaultMemSroreextends AbstractMemStore and thus only maintains the code that is not shared with other concrete memstores. Some additional changes in the DefaultMemStore come from utilizing the new MemStoreSegment component that now wraps all kv-set handling.

[new class] **MemStoreSegment**
This class facilitates the management of the compaction pipeline and the shifts of cell sets and memory allocation buffers (MSLAB) from active set to snapshot set. It mainly encapsulates the kv-set and its respective MSLAB.

[new class] **CompactedMemStore** extends AbstractMemStore
In addition to AbstractMemStore class members, CompactedMemStore includes compaction pipeline and a memstore compactor which runs as a background thread.
A prepare-for-flush phase includes the following steps:

1. push the active MemStoreSegment into the compaction pipeline
2. if not in a force flush mode, submit a compaction requests to the compactor
3. if in force flush mode, pull the last MemStoreSegment in the compaction pipeline and shift it to snapshot
4. create a new active MemStoreSegment

All methods retrieving information from memstore are overwritten to include access to the new compaction pipeline. Specifically, the memstore scanner includes scanners of all MemStoreSegments in the compaction pipeline.

[new class] **CompactionPipeline**
Class members are a list of read-only MemStoreSegments, and a boolean flag to indicate the force flush mode.
Supports the following methods:

- push-in: adds a new component into pipeline
- get scanners to all pipeline components
- swap subset of components with a new set of (compacted) components
- pull-out: removes the last component from the compaction pipeline

Scanners design

[new class] **MemStoreSegmentScanner**

A simple scanner to scan the kv-set in a MemStoreSegment, preserves the majority of the code from the old MemStoreScanner class

MemStoreScanner is no longer an inner class of DefaultMemStore. First citizen class, instead of being highly coupled with the implementation of DefaultMemStore. At creation gets an AbstractMemStore object, from which a collection of MemStoreSegmentScanners can be extracted. For example, for an application level scan, gets a DefaultMemStore object from which it can extract the list of two MemStoreSegmentScanners (for active set and for snapahot); - The MemStoreScanner reuses the KeyValueHeap logic to do the actual scanning, without knowing which MemStoreSegment it actually traverses.

Compactor design

[new class] **MemStoreCompactor** The MemStoreCompactor class is also first citizen class samely as MemStoreScanner. Upon creation MemStoreCompactor gets a subset of MemStoreSegments, which are subject for compaction (from CompactionPipeline). The result of compaction is a single MemStoreSegment. The newly created MemStoreSegment is going to replace the MemStoreSegment list used for compaction (update of CompactionPipeline).

MemStoreCompactor process has the ability to be dispatched with low priority and then discarded with no effect (in case of a force flush mode). The MemStoreCompactor needs no synchronization as its data sources are read only.

The main startCompact() method of this class uses the MemStoreScanner with a subset of MemStoreSegments, which are subject for compaction. According to the policy the scanner may skip deleted entries (or not) depending on Bloom filter, skip expired cells, cells to be removed because of version count, etc. Currently, the compaction policy is the one implemented via the instance of the ScanQueryMatcher which is also used for compaction in time when snapshot is flushed. After, the valid entries are going back to a newly created MemStoreSegment that replaces the MemStoreSegments subset used for scanning.