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

April 5, 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) size of a specific memstore 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 size of a specific memstore exceed upper limits.

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
2. Memstore data is either compacted or in process of being compacted
3. Allow a *panic mode,* which may interrupt an in-progress compaction and force a flush of part of the memstore, e.g., when 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) may need to be relaxed for this specific column family, since the memstore size might exceed the flush size, but should not trigger a flush as it is about to be compacted. In-memory column family can be the context for relaxing flush condition #1.

The changes are threefold:

Memstore structure and behaviour

We propose to add a memstore implementation which supports its in-memory compaction. A *compaction pipeline* is added between the active set and the snapshot data structures; it consists of a list of kv-sets that are subject to compaction (see Figures 1 and 2 below).
The semantics of the prepare-for-flush phase are changed. Instead of shifting the current active set to snapshot, the active set is pushed into the pipeline. Like the snapshot, all pipeline components are read-only; updates only affect the active set. To ensure this property we take advantage of the existing blocking mechanism -- the active 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 components, instead of scanning only the active set and the snapshot, also scan the pipeline components.

Memstore Compaction

Periodically, a compaction is applied in 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. Each flush triggers a compaction request. 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 panic 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. To get the most out of memstore compaction, we suggest to consult hfiles bloom filters to identify record history on disk, and decide if it is safe to remove the tombstone.

Low-level design

Memstore design

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

DefaultMemStore extends AbstractMemStore
The DefaultMemSrore is going to remain almost the same, but to extend the AbstractMemStore and thus to only maintain the code that is not shared with other concrete memstores.

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

[new class] **CompactMemStore** extends AbstractMemStore
Class members are 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 set into the compaction pipeline
2. if in panic mode, interrupt in-progress compaction (if any), pull the last component of the compaction pipeline and shift it to snapshot
3. create a new active set
4. submit a compaction requests to the compactor thread

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

[new class] **MemstoreCompactionPipeline**
private inner class of CompactMemStore

Class members are a list of read-only kv-sets (CellSetMgr), and a boolean flag to indicate the panic mode.
Supports the following methods:

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

Scanners design

[new class] **CellSetScanner**
A simple scanner to scan the kv-set in a CellSetMgr

MemStoreScanner is no longer an inner class of DefaultMemStore. First citizen class, instead of being highly coupled with the implementation of DefaultMemStore. At creation either gets a MemStore object or a collection of CellSetScanners. For example, for an application level scan, gets a MemStore object from which it can extract the list of CellSetScanners; for in-memory compaction gets only a list of the compaction pipeline CellSetScanners.
The MemStoreScanner reuses the KeyValueHeap logic to do the actual scanning, without knowing which cell-sets it actually traverses.

Compactor design

[new class] **MemStoreCompactor** doesn't inherit from Compactor, similarly as MemStoreScanner doesn't inherit from StoreScanner. The MemStoreCompactor class is also first citizen class samely as MemStoreScanner. Upon creation MemStoreCompactor gets a subset of CellSetMgrs, which are subject for compaction. The result of compaction is a single CellSetMgr. The user may decide what to do with the result, usually the newly created CellSetMgr is going to replace the CellSetMgr list used for compaction.

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

The main compact() method of this class uses the MemStoreScanner with a subset of CellSetMgrs, which are subject for compaction. According to the policy the scanner is skipping deleted entries (or not) depending on Bloom filter, skipping expired cells, cells to be removed because of version count, etc. Then the valid entries are going back to a newly created CellSetMgr that replaces the CellSetMgr's subset used for scanning.

Figure 1: The current MemStore design

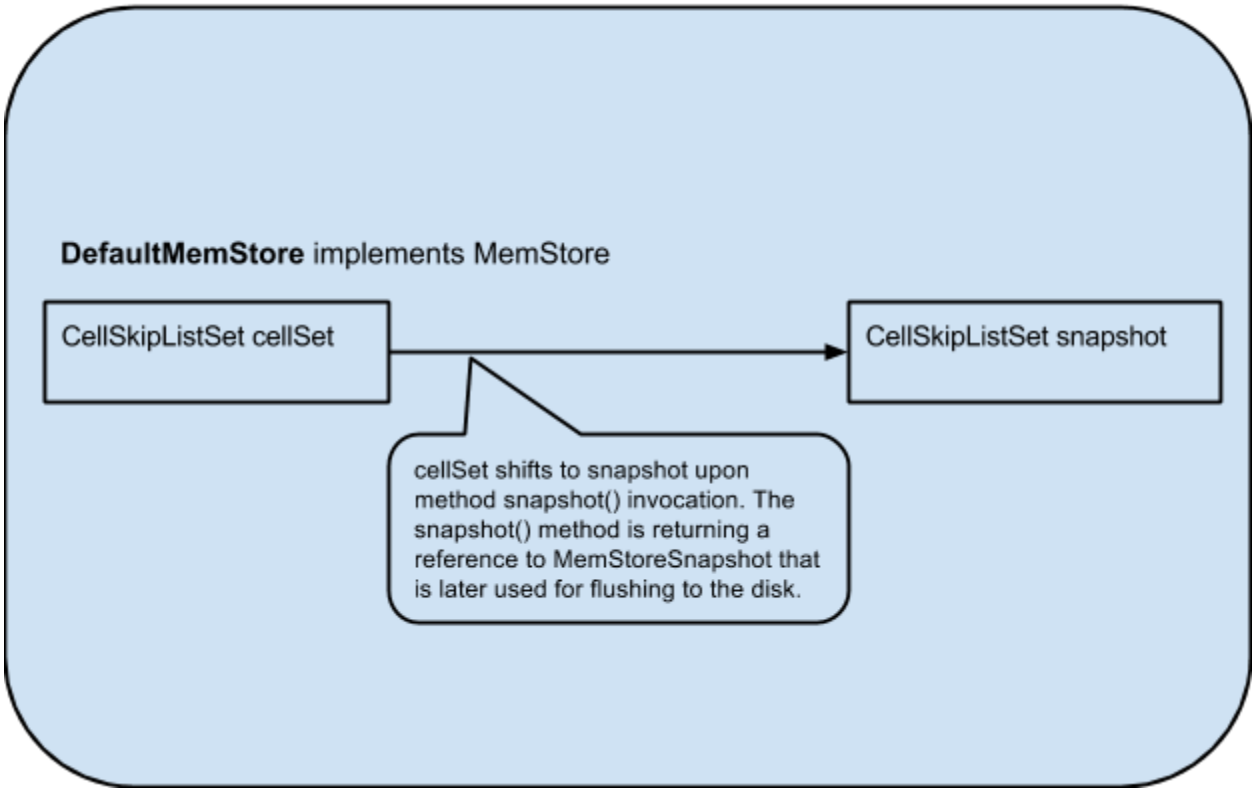

Figure 2: The suggested MemStore design