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

November 3, 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. Once the flush is completed the snapshot is discarded.

Generally, the faster the data is accumulated in memory, the 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 propose to compact the memstore data before it is flushed to disk. This is done by maintaining one more in-memory component between active set and snapshot. The new component is read-only and compacted, thus allowing optimal RAM usage.

In addition to profit from efficient memory space usage, the in-memory compaction component is a foundation for future change when the component can be represented in memory in more condensed way. In particular, the in-memory read-only component representation can be moved from an expensive data structure – the skip list – to a data structure that is more

compact in memory and more efficient to access, for example hfile. Flushes could be faster if the format in memory is an hfile.

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

1. Use the in-memory space effectively, by periodically compacting the memstore content.
2. Non-active segments of the data are either compacted or in process of being compacted
3. A flush call may interrupt an in-progress compaction and flush to the disk the compacted part of the data.

External API

set `IN_MEMORY_COMPACTION => true` when creating or altering a table in the shell, e.g.:
hbase(main):003:0> create 't', {NAME => 'f', IN_MEMORY_COMPACTION => 'true'}

High-level design

Our solution modifies the current design, reconstructing some memstore building blocks. 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 in a store segment entity. The changes are threefold:

Memstore structure and behaviour
In the existing implementation a store is composed of segments. The memstore manages the in-memory segments, while disk segments are represented by HStoreFiles. Default memstore has 2 segments (1) active (mutable) segment (2) snapshot (immutable) segment -- the previously active segment which became immutable.

In the new solution a new memstore implementation supports *in-memory flush and compaction*. A compacted memstore has (in addition to the two segments mentioned above) a pipeline of segments which are subject to compaction.
Whenever the size of the active segment exceeds a certain threshold an in-memory flush is called: the active segment is pushed into the pipeline (instead of shifting to snapshot). Like the snapshot, all pipeline segments are immutable; updates only affect the active segment.

We introduced the abstraction capturing a store segment so that a memory segment can be stored in a skip-list as well as in other formats. For example, memory segments which are formatted as HFile comprising flat blocks with low memory overhead that can be easily compressed.
Each segment has a scanner that is suited to its internal implementation and that allows to efficiently traverse through the segment's cells.

The new memstore design means that a memstore scanner may need to scan more than two segments--instead of scanning only the active segment and the snapshot segment, a compacted memstore scanner also scans the pipeline segments.

Flush to disk and in-memory flush

In the existing implementation 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 (blocking-flush-size). After a flush is triggered the stores to be flushed to disk are selected by the flush policy object. One policy selects all stores while the FlushLargeStoresPolicy selects only stores with size larger than HREGION_COLUMNFAMILY_FLUSH_SIZE_LOWER_BOUND.
Finally, the stored selected to be flushed are called to prepare for flush by shifting the active segment to be a snapshot segment and creating a new empty active segemnt. This quick preparation phase is protected by an exclusive region lock, `updatesLock`, and is executed at the scope of the region level.

In the new solution the counter measuring the region memstore size is divided into 2 counters: (1) capturing the size of the *active* segments and (2) for the *overflow* memstore size (e.g., pipeline segments). The total memstores size is the sum of these two counters. An additional flush-total-size is defined as the average of flush-size and blocking-flush-size. For example, if the blocking-flush-size is, say, 3 times the flush size, then the total-flush-size is 2 times the flush-size.
A flush is invoked if the active segments size exceeds the flush-size, or if the total memstore size exceeds the flush-total-size. Flush condition #1 is refined since the total memstore size might exceed the flush size, but should not trigger a flush to disk as data is about to be compacted. After the flush is triggered the existing flush policy decides which stores to actually flush to disk. In case of FlushLargeStoresPolicy, the decision is based on the size of the active segment. To allow compacted memstore have enough slack to manage its compaction pipeline the threshold to apply in-memory flush is set to 0.9*FlushLargeStoresPolicy.HREGION_COLUMNFAMILY_FLUSH_SIZE_LOWER_BOUND. This way we keep the active segment smaller than the large stores lower bound and the store skips the flush to disk unless the conditions are such that the store is forced to flush (for example when all stores are asked to flush to disk.

Upon flush to disk the active segment is pushed to the compaction pipeline, and the segment at the tail of the pipeline shifts to being the snapshot, which is later flushed to disk.

Upon in-memory flush, we take advantage of the existing blocking mechanism -- the active segment is pushed to the pipeline while holding the region `updatesLock` in exclusive mode.

Then, the region `updatesLock` is released and a compaction is applied at the background to all pipeline segments resulting in a single immutable segment. This procedure is executed at the scope of the store level, and specifically in the context of the compacted memstore.

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, multiple versions are flatten into a single version, like when flushing the snapshot to disk. 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.
The “old” segments 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 in-memory-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, estimated memstore duplication ratio, etc.

WAL truncation
In the existing implementation when a region flushes a store, the previous sequence id that was associated with this store in the WAL oldestUnflushedStoreSequenceIds is removed. The first put operation to occur after the flush installs a new sequence id for the store.
The WAL uses this bookeeping when it needs to decide which WAL files can be archived (WAL truncation).
In the new solution upon flush to disk, lowest-unflushed-sequence-id in WAL is cleared (like it used to be). Stores with overflow segments, update this with a lower approximation of the lowest sequence id still in memory. Other stores update this sequence id with the first insert after the flush (like it used to be).
In addition, the compaction thread in compacted memstore is responsible for setting an approximation of the correct sequence id for the store in oldestUnflushedStoreSequenceIds.

How do we compute this sequence id estimation? The compacting memstore maintains a mapping of timestamp to region sequence number (the same sequence numbers that are attached to WAL edits). Whenever applying in-memory flush the memstore adds the current time and current sequence number pair to this mapping.
As an additional artifact of in-memory flush the minimal timestamp that is still present in the memstore is computed. This timestamp is then used to identify the maximal sequence id in the timestamp->seqId mapping for which no entries are left in the memstore. Finally, it uses this approximated sequence number to update the oldestUnflushedStoreSequenceIds set.
This way the WAL is being truncated with some delay with respect to the real sequence number, but the memory overhead if fairly small (only a small map of ts->seq is added to the memstore). If the WAL becomes too big despite this best-effort truncation, the WAL forces a flush to disk. This is taken care of by the existing LogRoller mechanism which awakes from time to time and checks whether WAL has too many files and if so enforces a flush to disk.

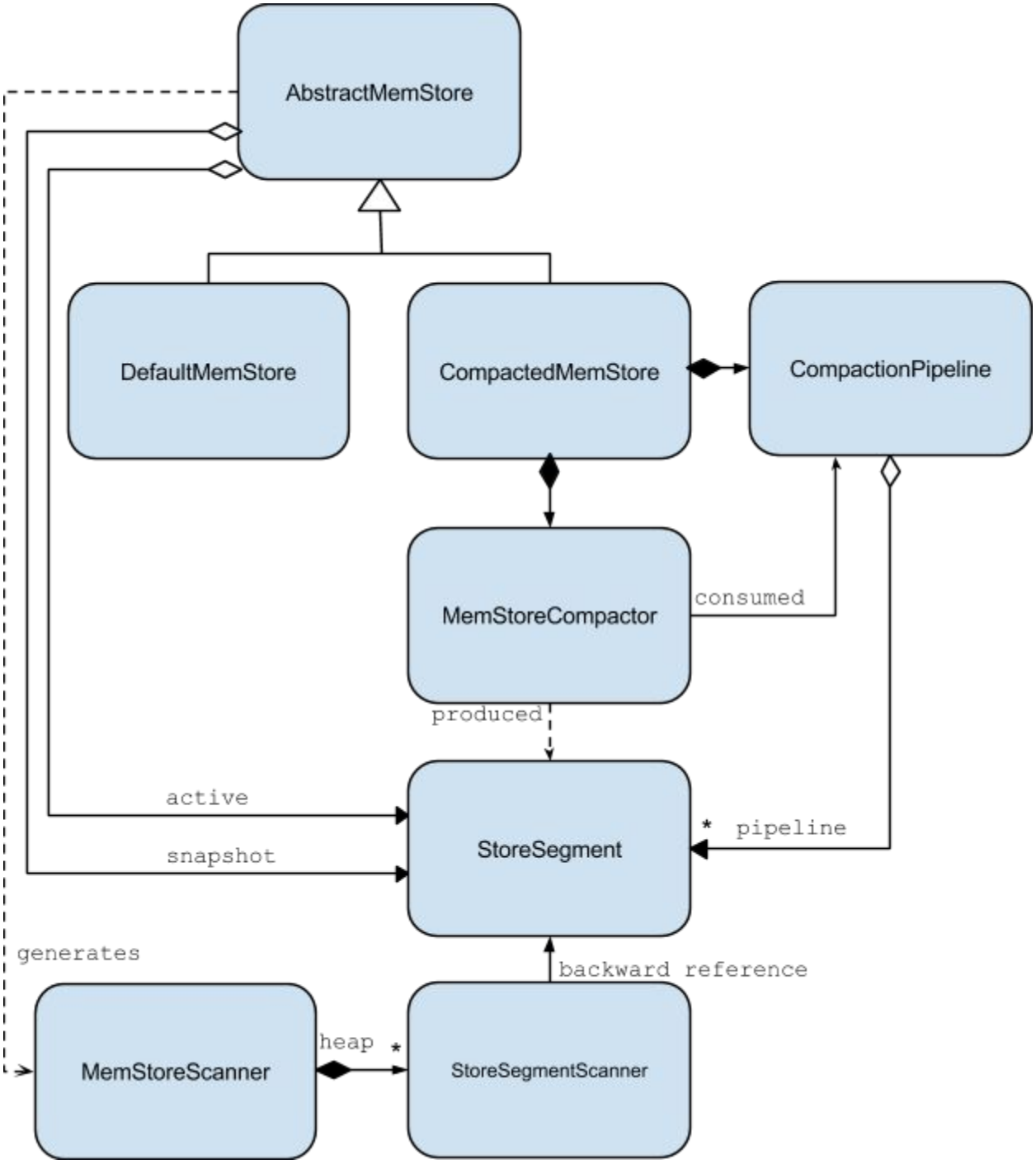
Low-level design

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 StoreSegment component that now wraps all kv-set handling.

The Class Diagram



[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.
An in-memory-flush includes the following steps:

1. push the active StoreSegment into the compaction pipeline
2. submit a compaction requests to the compactor
3. create a new active StoreSegment

A regular flush (to disk) pulls the last StoreSegment from the compaction pipeline and shifts it to snapshot
All methods retrieving information from memstore are overwritten to include access to the new compaction pipeline. Specifically, the memstore scanner includes scanners of all StoreSegments in the compaction pipeline.

[new class] **CompactionPipeline**
Class members are a list of immuable StoreSegments.
Supports the following methods:

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

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 StoreSegmentScanners can be extracted. For example, an application level scan gets a DefaultMemStore object from which it can extract the list of two StoreSegmentScanners (for active set and for snapahot); -
The MemStoreScanner reuses the KeyValueHeap logic to do the actual scanning, without knowing which StoreSegment it actually traverses. (See details of StoreSegment and StoreSegmentScanner hierarchies below)

Compactor design

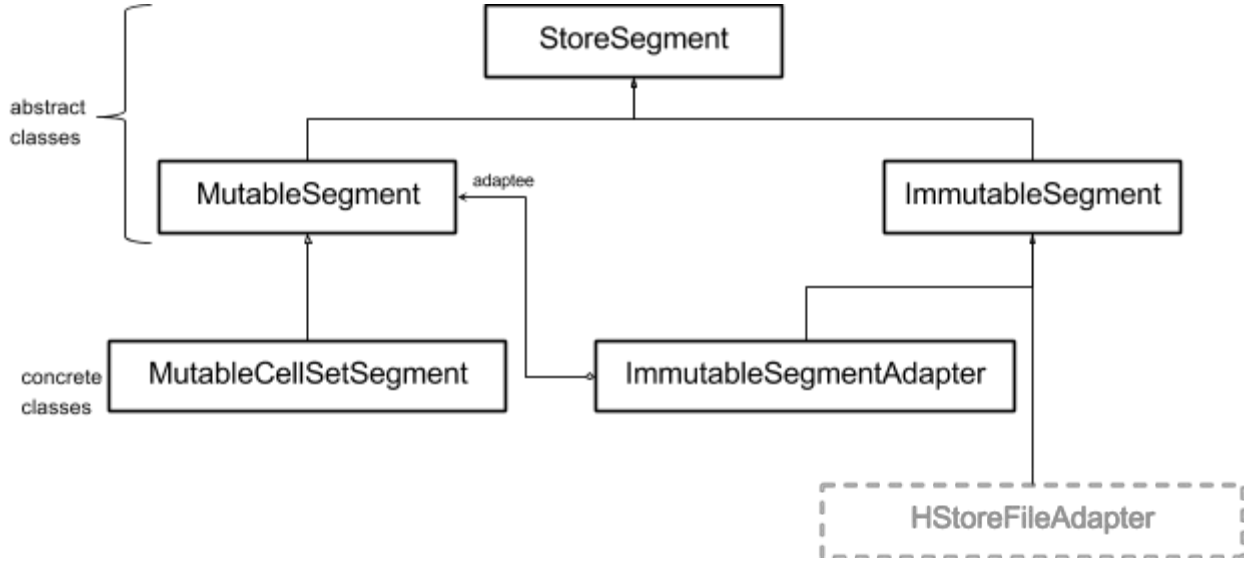
[new class] **MemStoreCompactor** The MemStoreCompactor class is also first citizen class samely as MemStoreScanner. Upon creation MemStoreCompactor gets a subset of StoreSegments, which are subject for compaction (from CompactionPipeline). The result of compaction is a single StoreSegment. The newly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

StoreSegment - new class hierarchy



Abstract classes

StoreSegment is an abstract segment class. It defines the API which is needed by the memstore to manage the segment

MutableSegment is an abstract class which extends the API supported by a store segment, and is not needed for immutable segments, e.g., incSize(). Only the active segment is a mutable segment.

ImmutableSegemnt is an abstract class which extends the API supported by a store segment, and is not needed for a mutable segment. Specifically, the method getScannerForMemStoreSnapshot() builds a special scanner for the MemStoreSnapshot object. In addition, this class overrides methods that are not likely to be supported by an immutable segment, e.g., rollback() and getCellSet() which can be very inefficient. Specific sub-classes of immutable segments can re-override these methods if they are able to support them efficiently.

Concrete classes

MutableCellSetSegment is the cell-set based implementation of a mutable memstore segment. It encapsulates a mutable skip-list cell set structure and its respective memory allocation buffers (MSLAB).

ImmutableSegmentAdapter wraps and adapts a mutable segment. This is used when a mutable segment is moved to being a snapshot or pushed into a compaction pipeline, that consists only of immutable segments. The compaction pipeline may generate different type of mutable segments.

HStoreFileAdapter TBD. This class wraps an HStoreFile to be used as a memory segment.

Segment Factory

The class StoreSegmentFactory is responsible for creating the proper segments. The methods
createImmutableSegment(...)
createMutableSegment(...)
create immutable and mutable segments, respectively. The specific type of the concrete class (as well as other parameters) can be set in the configuration that is passed to the methods.

StoreSegmentScanner - new class Hierarchy

StoreSegmentScanner is an interface for store segment scanner, can serve both memory segment (memstore segment) and disk segments (files). It extends the KeyValueScanner interface with two additional methods:
setSequenceID() - is needed to determine the order of memory segment scanners when there is more than one (or two);
shouldSeek() - helps to filter out the segments that can be skipped during the scan.
A MemStoreScanner (see the original design document) is executing the scan with a list of store segment scanners, one per segment in the memstore.

MutableCellSetSegmentScanner is currently the only class implementing the store segment scanner interface. It supports an efficient scan of a MutableCellSetSegment.

When HStoreFileAdapter is implemented, the **HStoreFileScanner** can be refactored to implement StoreSegmentScanner instead of KeyValueScanner, so it can be used as one of the memstore scanners.