Accordion:
HBase Goes Realtime with In-Memory Compaction

Disclaimer: Work-in-Progress

By Anastasia Braginsky, Eshcar Hillel and Edward Bortnikov, Yahoo Research

HBase popularity for mass-scale, real-time data processing, serving, and advanced analytics is on continuous rise. Modern technologies and products powered by HBase exhibit ever-increasing expectations from its the read and write performance. Ideally, HBase clients would like to enjoy the speed of in-memory databases without giving up on the reliable persistent storage guarantees. We introduce a new algorithm in HBase 2.0, named Accordion, which takes a significant step towards this goal.

HBase partitions the data into *regions* controlled by a cluster of *region servers.* The internal (vertical) scalability of a region server is crucial for end-user performance as well as for the overall system utilization. Accordion improves the region server scalability via a better use of RAM. It accommodates more data in memory and retains it there longer compared to the earlier implementation. This manifests in two desirable phenomena. First, the data can be served from RAM more often, therefore the ***read latencies are reduced***. Second, the disk writes become less frequent, therefore HBase updates less bytes on disk per logical update, i.e., its ***write amplification is reduced,*** too. Traditionally, these two metrics were considered at odds, and tuned at each other's expense. With Accordion, both are improved simultaneously.

Accordion is inspired by the Log-Structured-Merge (LSM) tree design pattern that governs the HBase storage organization. An HBase region is stored as a sequence of searchable key-value maps, called levels. The topmost level is a mutable in-memory store, called *MemStore*, which absorbs the recent write (*put*) operations. The rest are immutable HDFS files, called *HFile*s. Once a MemStore overflows, it is flushed to disk, creating a new HFile. HBase adopts the multi-versioning approach to concurrency control, that is, MemStore stores all data modifications as separate versions. Multiple versions of one key may therefore reside in MemStore and HFile levels. A read (*get*) operation, which retrieves the value by key, scans the levels top-down, seeking for the latest version. To reduce the number of disk accesses, HFiles are merged in the background. This process, called *compaction*, creates larger files and eliminates redundancies.

LSM trees delivers superior write performance by transforming random application-level I/O to sequential disk I/O. However, their traditional design makes no attempt to compact the in-memory data. Accordion addresses exactly this aspect. It reapplies the LSM principle to MemStore, in order to eliminate redundancies while the data is still in RAM. Doing so

decreases the frequency of flushes (thereby reducing the write amplification), and increases the probability of reads retrieving data from memory (thereby reducing the tail latencies).

The algorithm is most useful for applications with high data churn. Examples include producer-consumer queues, shopping carts, shared counters, etc. All these use cases feature frequent updates of the same keys, which generate multiple redundant versions that Accordion takes advantage of.

How To Configure

The in-memory compaction parameter is configured per table/column family pair.

HBase shell syntax:
create '<tablename>', {NAME => '<cfname>', IN_MEMORY_COMPACTION => true}[1]

How Accordion Works

Data Structure. Accordion introduces *CompactingMemStore* - a MemStore implementation that applies compaction internally. Contrast to the default MemStore, which maintains all data in one monolithic data structure, Accordion manages it as a sequence of *segments*. The youngest segment, called *active*, is mutable; it absorbs the put operations. Upon overflow (by default, 32MB - 25% of the MemStore size bound), the active segment is moved to an in-memory *pipeline*, and becomes immutable. We call this *in-memory flush.* Get operations scan through these segments and the HFiles (the latter are accessed through the block cache, as usual in HBase).

Figure 1 illustrates CompactingMemStore versus the traditional design.

[1] *This syntax is experimental, and subject to changes.*

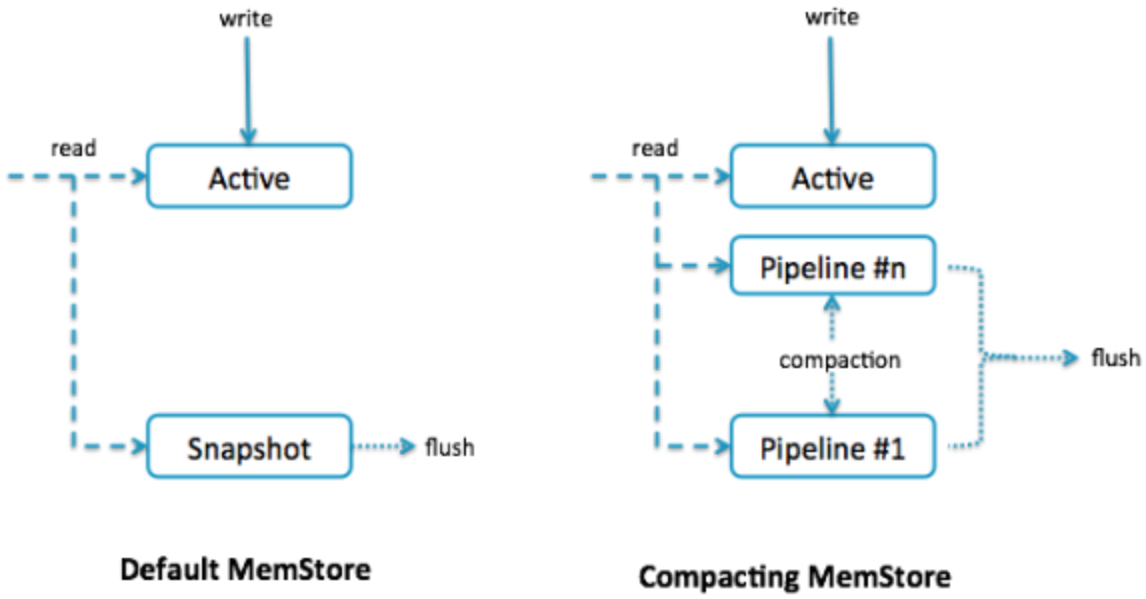

Figure 1. CompactingMemStore vs DefaultMemStore

Similarly to the default MemStore, CompactingMemStore maintains an index on top of cell storage, to allow fast search by key. Traditionally, the index was implemented as a Java skiplist - a dynamic but wasteful data structure that manages a lot of small objects. CompactingMemStore uses a space-efficient flat layout for immutable segment indexes. Once a segment is added to the pipeline, the store serializes its index into a sorted array (in our terminology, *CellArray*) that is amenable to fast binary search.

HBase supports memory management in two flavors - the standard Java heap and a custom allocator that avoids GC costs. The latter manages cell storage in MemStore-Local Allocation Buffers (MSLABs), which can be either on-heap or off-heap. CompactingMemStore gets along well with both memory managers.

Figure 2 provides a zoom-in on the structure of an immutable segment with CellArray index and MLSAB storage.

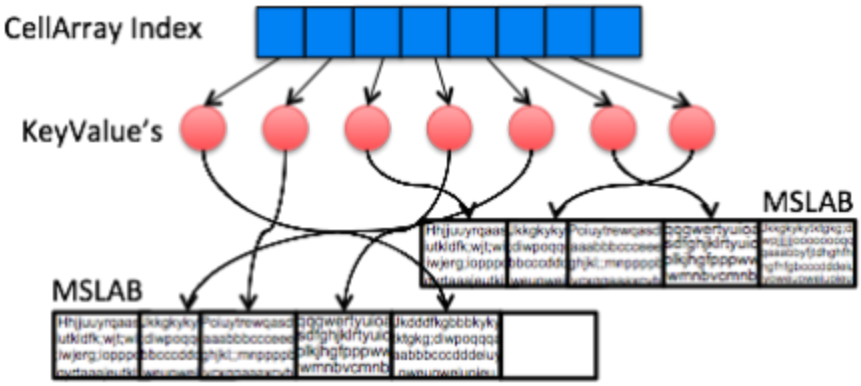

Figure 2. Immutable segment with CellArray index

In-Memory Compaction. CompactingMemStore may merge multiple immutable segments in the background from time to time, creating larger and leaner segments. The pipeline is therefore "breathing" (expanding and contracting), similar to accordion bellows. When the overall size of the active and pipelined segments exceeds the limit (default: 128MB), one or more of the older segments are compacted and streamed to a new HFile, instead of creating a new segment. This flush may clean the whole pipeline.

The compaction algorithm applies two tools: *index compaction* and *data compaction.* Index compaction merges multiple index arrays into one; index entries keep referencing the same data. By contrast, data compaction filters out redundant versions (via scan), and stores only the references to live versions in the new index. When data is stored in MSLABs, data compaction copies the survivors to a new MSLAB. Obviously, data compaction delivers the most value when the data is highly redundant, but comes at a cost of an extra copy.

In-memory compaction is applied with deliberation, to balances between two needs: (1) keep the number of segments small (to reduce memory accesses upon reads and eliminate redundancies), and (2) keep the compaction overhead small (in terms of data copy and incurred GC)[2]. Ideally, Accordion should be self-tuning, making the right decisions with zero extra configuration. The compaction triggering policy is still subject to experimentation.

Off-heap Storage. In order to make the most of the MSLAB storage, CompactingMemStore supports a segment index structure that embeds directly into MLSABs[3]. This index type is called *CellChunkMap.* Figure 3 depicts a segment with CellChunkMap index.

[2] The exact tuning of the in-memory compaction policy is under development.
[3] This feature is under development.

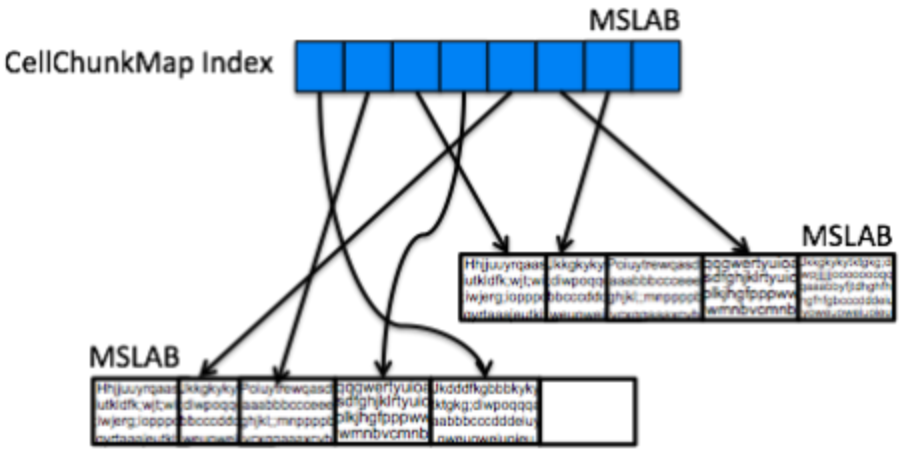

Figure 3. Immutable segment with CellChunkMap index

MSLAB-resident indexes save the overhead of managing multiple small key-value objects (see Figure 2), and more importantly, opens the way to storing most of the CompactingMemStore data offheap. This write path offheap storage optimization complements the similar mechanism for the read-path BucketCache, and further improves the region server's vertical scalability.

Performance Evaluation

TBC