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 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).

Accordion currently provides two levels of in-memory compaction - *basic* and *eager.* The former applies simple optimizations that are good for all data update patterns, and delivers moderate value. The latter is most useful for applications with high data churn, like 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 the algorithm takes advantage of. On the flip side, eager optimization may incur compute overhead (more memory copies and garbage collection), which may affect response times under very high write loads. Future implementations may tune the optimal compaction policy automatically, based on the observed workload.

How To Use

The in-memory compaction level can be configured both globally and per column family. The supported levels are ***none*** (legacy implementation), ***basic***, and ***eager***.

By default, all tables apply ***basic*** in-memory compaction. This global configuration can be overridden in *hbase-site.xml,* as follows:

```
<property>
        <name>hbase.hregion.compacting.memstore.type</name>
        <value><none|basic|eager></value>
 </property>
```

The level can be also configured in HBase shell per column family, as follows:

```
create '<tablename>',
{NAME => '<cfname>', IN_MEMORY_COMPACTION => '<NONE|BASIC|EAGER>'}
```

How Accordion Works

High Level Design. 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 via the block cache, as usual in HBase).

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), all segments are moved to a composite *snapshot,* merged, and streamed to a new HFile.

Figure 1 illustrates the structure of CompactingMemStore versus the traditional design.

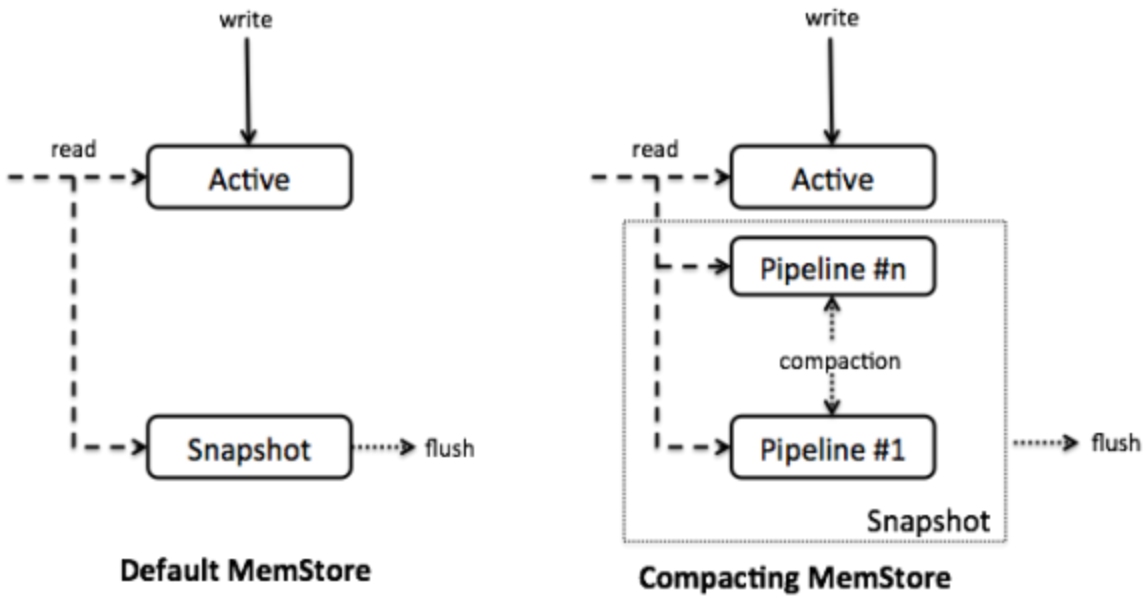

Figure 1. CompactingMemStore vs DefaultMemStore

Segment Structure. Similarly to the default MemStore, CompactingMemStore maintains an index on top of cell storage, to allow fast search by key. Traditionally, this 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 that is amenable to fast binary search.

Two flavors of flat indexes are supported - *CellArrayMap* for on-heap cell storage, and *CellChunkMap* for for off-heap storage[1]. CellArrayMap supports both direct allocation of cells from the Java heap and custom allocation from MemStore-Local Allocation Buffers (MSLABs). The implementation differences are abstracted away via the helper KeyValue objects that are referenced from the index (see Figure 2a). CellChunkMap supports off-heap MSLABs storage only, and embeds the index itself in MSLAB, thereby eliminating the extra referencing layer (Figure 2b).

[1] *This feature is under development.*

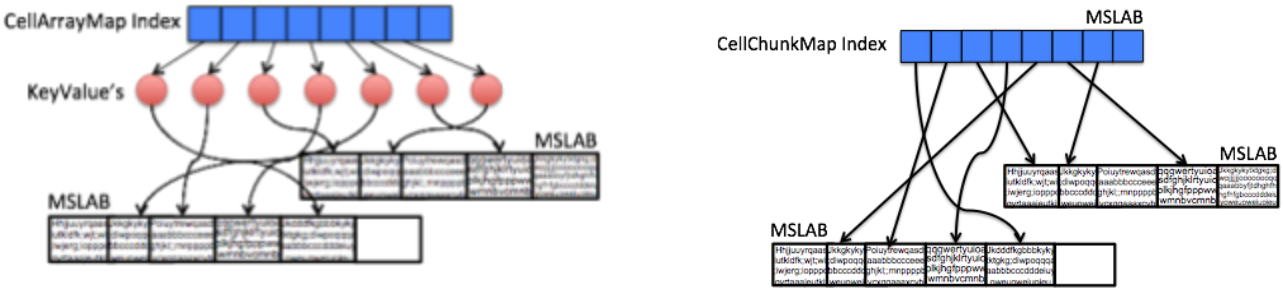

Figure 2. Immutable segment with a flat index and MSLAB cell storage: (a) CellArrayMap (on-heap), (b) CellChunkMap (off-heap).

Compaction Algorithms. Basic compaction flattens the cell index upon in-memory flush. This optimization saves some space, especially when the data items are small. The active segment size cap of 1/4 MemStore size ensures the number of segments remains small, therefore the impact on get latency is immaterial.

Eager compaction merges multiple indexes into one, and filters out the redundant data versions. This is accomplished via a simultaneous scan of all segments in the pipeline, similarly to the way on-disk compaction works. In this context, if cells are directly allocated from heap, only references to surviving KeyValue objects are copied to the new index. Under MSLAB allocation, the data itself is copied to the newly created MSLAB(s). Obviously, the higher the data redundancy, the further in time the disk flushes are pushed, the less data is written to disk, and consequently, the less is processed by the further on-disk compactions. However, these advantages come at a cost of extra copy and incurred GC.

Future implementations of compaction might automate the choice between the basic and eager compaction policies. For example, the algorithm might try eager compaction once in awhile, and schedule the next compaction based on the value delivered (i.e., fraction of data eliminated). Such an approach could relieve the system administrator from deciding a-priori, and adapt to changing access patterns.

Summary

In this blog post, we covered Accordion's basic principles, configuration, and some details of the in-memory compaction algorithms. The next posts will focus on performance evaluation and on system internals for HBase developers.