Accordion: Better Memory Organization for LSM Key-Value Stores

Edward Bortnikov
Yahoo Research
ebortnik@oath.com

Anastasia Braginsky
Yahoo Research
anastas@oath.com

Eshcar Hillel
Yahoo Research
eshcar@oath.com

Idit Keidar
Technion and Yahoo Research
idish@ee.technion.ac.il

Gali Sheffi
Yahoo Research
gsheffi@oath.com

ABSTRACT

Log-structured merge (LSM) stores have emerged as the technology of choice for building scalable write-intensive key-value storage systems. An LSM store replaces random I/O with sequential I/O by accumulating large batches of writes in a *memory store* prior to flushing them to log-structured disk storage; the latter is continuously re-organized in the background through a *compaction* process for efficiency of reads. Though inherent to the LSM design, frequent compactions are a major pain point because they slow down data store operations, primarily writes, and also increase disk wear. Another performance bottleneck in today's state-of-the-art LSM stores, in particular ones that use managed languages like Java, is the fragmented memory layout of their dynamic memory store.

In this paper we show that these pain points may be mitigated via better organization of the memory store. We present Accordion – an algorithm that addresses these problems by re-applying the LSM design principles to memory management. Accordion is implemented in the production code of Apache HBase, where it was extensively evaluated. We demonstrate Accordion's double-digit performance gains versus the baseline HBase implementation and discuss some unexpected lessons learned in the process.

PVLDB Reference Format:
Edward Bortnikov, Anastasia Braginsky, Eshcar Hillel, Idit Keidar, and Gali Sheffi. Accordion: Better Memory Organization for LSM Key-Value Stores. *PVLDB*, 11 (12): 1863 - 1875, 2018.
DOI: https://doi.org/10.14778/3229863.3229873

1. INTRODUCTION

1.1 LSM Stores

Persistent NoSQL *key-value stores (KV-stores)* have become extremely popular over the last decade, and the range of applications for which they are used continuously increases. A small sample of recently published use cases includes massive-scale online analytics (Airbnb/ Airstream [2], Yahoo/Flurry [7]), product search and recommendation (Alibaba [13]), graph storage (Facebook/Dragon [5], Pinterest/Zen [19]), and many more.

The leading approach for implementing write-intensive key-value storage is *log-structured merge (LSM)* stores [31]. This technology is ubiquitously used by popular key-value storage platforms [9, 14, 16, 22, 4, 1, 10, 11]. The premise for using LSM stores is the major disk access bottleneck, exhibited even with today's SSD hardware [14, 33, 34].

An LSM store includes a *memory store* in addition to a large *disk store*, which is comprised of a collection of files (see Figure 1). The memory store absorbs writes, and is periodically *flushed* to disk as a new immutable file. This approach improves write throughput by transforming expensive random-access I/O into storage-friendly sequential I/O. Reads search the memory store as well as the disk store.

The number of files in the disk store has adverse impact on read performance. In order to reduce it, the system periodically runs a background *compaction* process, which reads some files from disk and merges them into a single sorted one while removing redundant (overwritten or deleted) entries. We provide further background on LSM stores in Section 2.

1.2 LSM Performance Bottlenecks

The performance of modern LSM stores is highly optimized, and yet as technology scales and real-time performance expectations increase, these systems face more stringent demands. In particular, we identify two principal pain points.

First, LSM stores are extremely sensitive to the rate and extent of compactions. If compactions are infrequent, read performance suffers (because more files need to be searched and caching is less efficient when data is scattered across multiple files). On the other hand, if compactions are too frequent, they jeopardize the performance of both writes and reads by consuming CPU and I/O resources, and by indirectly invalidating cached blocks. In addition to their performance impact, frequent compactions increase the disk write volume, which accelerates device wear-out, especially for SSD hardware [27].

Second, as the amount of memory available in commodity servers rises, LSM stores gravitate towards using larger memory stores. However, state-of-the-art memory stores are

Proceedings of the VLDB Endowment, Vol. 11, No. 12
Copyright 2018 VLDB Endowment 2150-8097/18/8.
DOI: https://doi.org/10.14778/3229863.3229873

organized as dynamic data structures consisting of many small objects, which becomes inefficient in terms of both cache performance and *garbage collection (GC)* overhead when the memory store size increases. This has been reported to cause significant problems, which developers of managed stores like Cassandra [4] and HBase [1] have had to cope with [12, 3]. Note that the impact of GC pauses is aggravated when timeouts are used to detect faulty storage nodes and initiate fail-over.

Given the important role that compaction plays in LSM stores, it is not surprising that significant efforts have been invested in compaction parameter tuning, scheduling, and so on [6, 18, 17, 32]. However, these approaches only deal with the aftermath of organizing the disk store as a sequence of files created upon memory store overflow events, and do not address memory management. And yet, the amount of data written by memory flushes directly impacts the ensuing flush and compaction toll. Although the increased size of memory stores makes flushes less frequent, today's LSM stores do not physically delete removed or overwritten data from the memory store. Therefore, they do not reduce the amount of data written to disk.

The problem of inefficient memory management in managed LSM stores has received much less attention. The only existing solutions that we are aware of work around the lengthy JVM GC pauses by managing memory allocations on their own, either using pre-allocated object pools and local allocation buffers [3], or by moving the data to off-heap memory [12].

1.3 Accordion

We introduce Accordion, an algorithm for memory store management in LSM stores. Accordion re-applies to the memory store the classic LSM design principles, which were originally used only for the disk store. The main insight is that the memory store can be partitioned into two parts: (1) a small dynamic segment that absorbs writes, and (2) a sequence of static segments created from previous dynamic segments. Static segments are created by *in-memory flushes* occurring at a higher rate than disk flushes. Note that the key disadvantage of LSM stores – namely, the need to search multiple files on read – is not as significant in Accordion because RAM-resident segments can be searched efficiently, possibly in parallel.

Accordion takes advantage of the fact that static segments are immutable, and optimizes their index layout as flat; it can further *serialize* them, i.e., remove a level of indirection. The algorithm also performs *in-memory compactions*, which eliminate redundant data *before* it is written to disk. Accordion's data organization is illustrated in Figure 2; Section 3 fleshes out its details.

The new design has the following benefits:

- *Fewer compactions.* The reduced footprint of immutable indices as well as in-memory compactions delay disk flushes. In turn, this reduces the write volume (and resulting disk wear) by reducing the amount of disk compactions.

- *More keys in RAM.* Similarly, the efficient memory organization allows us to keep more keys in RAM, which can improve read latency, especially when slow HDD disks are used.

- *Less GC overhead.* By dramatically reducing the size of the dynamic segment, Accordion reduces GC overhead, thus improving write throughput and making performance more predictable. The flat organization of the static segments is also readily amenable to off-heap allocation, which further reduces memory consumption by allowing serialization, i.e., eliminating the need to store Java objects for data items.

- *Cache friendliness.* Finally, the flat nature of immutable indices improves locality of reference and hence boosts hardware cache efficiency.

Accordion is implemented in HBase production code. In Section 4 we experiment with the Accordion HBase implementation in a range of scenarios. We study production-size datasets and data layouts, with multiple storage types (SSD and HDD). We focus on high throughput scenarios, where the compaction and GC toll are significant.

Our experiments show that the algorithm's contribution to overall system performance is substantial, especially under a heavy-tailed (Zipf) key access distribution with small objects, as occurs in many production use cases [35]. For example, Accordion improves the system's write throughput by up to 48%, and reduces read tail latency (in HDD settings) by up to 40%. At the same time, it reduces the write volume (excluding the log) by up to 30%. Surprisingly, we see that in many settings, disk I/O is *not* the principal bottleneck. Rather, the memory management overhead is more substantial: the improvements are highly correlated with reduction in GC time.

To summarize, Accordion takes a proactive approach for handling disk compaction even before data hits the disk, and addresses GC toll by directly improving the memory store structure and management. It grows and shrinks a sequence of static memory segments resembling accordion bellows, thus increasing memory utilization and reducing fragmentation, garbage collection costs, and the disk write volume. Our experiments show that it significantly improves end-to-end system performance and reduces disk wear, which led to the recent adoption of this solution in HBase production code; it is generally available starting the HBase 2.0 release. Section 5 discusses related work and Section 6 concludes the paper.

2. BACKGROUND

HBase is a distributed key-value store that is part of the Hadoop open source technology suite. It is implemented in Java. Similarly to other modern KV-stores, HBase follows the design of Google's Bigtable [22]. We sketch out their common design principles and terminology.

Data model KV-stores hold data items referred to as *rows* identified by unique row keys. Each row can consist of multiple *columns* (sometimes called fields) identified by unique column keys. Co-accessed columns (typically used by the same application) can be aggregated into *column families* to optimize access. The data is multi-versioned, i.e., multiple versions of the same row key can exist, each identified by a unique *timestamp*. The smallest unit of data, named *cell*, is defined by a combination of a row key, a column key, and a timestamp.

The basic KV-store API includes *put* (point update of one or more cells, by row key), *get* (point query of one or more

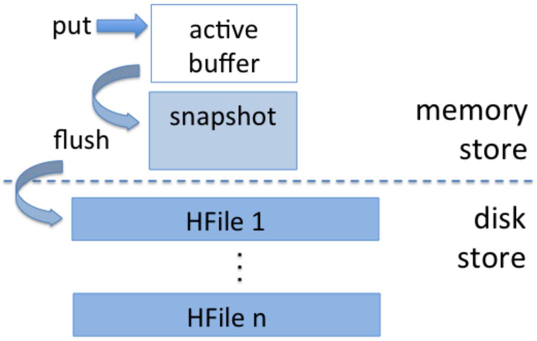


Figure 1: **A log-structured merge (LSM) store consists of a small memory store (*MemStore* in HBase) and a large disk store (collection of *HFiles*). Put operations update the MemStore. The latter is double-buffered: a flush creates an immutable *snapshot* of the active buffer and a new active buffer. The snapshot is then written to disk in the background.**

columns, by row key, and possibly column keys), and *scan* (range query of one or more columns, of all keys between an ordered pair of begin and end row keys).

Data management KV-stores achieve scalability by sharding tables into range partitions by row key. A shard is called *region* in HBase (*tablet* in Bigtable). A region is a collection of *stores*, each associated with a column family. Each store is backed by a collection of files sorted by row key, called *HFiles* in HBase (*sst files* in Bigtable).

In production, HBase is typically layered on top of Hadoop Distributed Filesystem (HDFS), which provides a reliable file storage abstraction. HDFS and HBase normally share the same hardware. Both scale horizontally to thousands of nodes. HDFS replicates data for availability (3-way by default). It is optimized for very large files (the default block size is 64MB).

Data access in HBase is provided through *region servers* (analogous to *tablet servers* in Bigtable). Each region server controls multiple stores (tens to hundreds in production settings). For locality of access, the HBase management plane (*master server*) tries to lay out the HFiles in HDFS so that their primary replicas are collocated with the region servers that control them.

LSM stores Each store is organized as an LSM store, which collects writes in a memory store – *MemStore* in HBase – and periodically flushes the memory into a disk store, as illustrated in Figure 1. Each flush creates a new immutable HFile, ordered by row key for query efficiency. HFiles are created big, hence flushes are relatively infrequent.

To allow puts to proceed in parallel with I/O, MemStore employs double buffering: it maintains a dynamic *active* buffer absorbing puts, and a static *snapshot* that holds the previous version of the active buffer. The latter is written to the filesystem in the background. Both buffers are ordered by key, as are the HFiles.

A flush occurs when either the active buffer exceeds the region size limit (128MB by default) or the overall footprint of all MemStores in the region server exceeds the global size limit (40% of the heap by default). Flush first shifts the current active buffer to be the snapshot (making it immutable) and creates a new empty active buffer. It then replaces the reference to the active buffer, and proceeds to write the snapshot as a new HFile.

In order to guarantee durability of writes between flushes, updates are first written to a *write-ahead log* (WAL). HBase implements the logical WAL as collection of one or more physical logs per region server, called *HLogs*, each consisting of multiple log files on HDFS. As new data gets flushed to HFiles, the old log files become redundant, and the system collects them in the background. If some HLog grows too big, the system may forcefully flush the MemStore before the memory bound is reached, in order to enable the log's truncation. Since logged data is only required at recovery time and is only scanned sequentially, HLogs may be stored on slower devices than HFiles in production (e.g., HDDs vs SSDs). Real-time applications often trade durability for speed by aggregating multiple log records in memory prior to asynchronously writing them to WAL in a bulk.

To keep the number of HFiles per store bounded, *compactions* merge multiple files into one, while eliminating redundant (overwritten) versions. If compactions cannot keep up with the flush rate, HBase may either throttle or totally block puts until compactions successfully reduce the number of files.

Most compactions are *minor*, in the sense that they merge a subset of the HFiles. *Major* compactions merge all the region's HFiles. Because they have a global view of the data, major compactions also eliminate *tombstone* versions that indicate row deletions. Typically, major compactions incur huge performance impact on concurrent operations. In production, they are either carefully scheduled or performed manually [6].

The get operation searches for the key in parallel in the MemStore (both the active and the snapshot buffers) and in the HFiles. The HFile search is expedited through Bloom filters [22], which eliminate most redundant disk reads. The system uses a large RAM cache for popular HFile blocks (by default 40% of the heap).

Memory organization The MemStore's active buffer is traditionally implemented as a dynamic index over a collection of cells. For the index, HBase uses the standard Java concurrent skiplist map [8]. Data is multi-versioned – every put creates a new immutable version of the row it is applied to, consisting of one or more cells. Memory for data cells is allocated in one of two ways as explained below.

This implementation suffers from two drawbacks. First, the use of a big dynamic data structure leads to an abundance of small objects and references, which inflate the in-memory index and induce a high GC cost. The overhead is most significant when the managed objects are small, i.e., the metadata-to-data ratio is big [35]. Second, the versioning mechanism makes no attempt to eliminate redundancies prior to flush, i.e., the MemStore size steadily grows, independently of the workload. This is especially wasteful for heavy-tailed (e.g., power law) key access distributions, which are prevalent in production workloads [24].

Memory allocation HBase implements two schemes for allocating data cells in the MemStore: either (1) each cell is a standard Java object called *pojo* – Pure Old Java Object;

or (2) cells reside in a bulk-allocated byte array managed by the *MSLAB* – MemStore Local Allocation Buffer [3] – module. In HBase 2.0, MSLAB is mostly used for off-heap memory allocation. With this approach, the MemStore's memory consists of large blocks, called *chunks*. Each chunk is fixed in size and holds data for multiple cells pertaining to a single MemStore.

3. Accordion

We describe Accordion's architecture and basic operation in Section 3.1. We then discuss in-memory compaction policies in Section 3.2 and implementation details and thread synchronization in Section 3.3. Section 3.4 discusses the index layout in off-heap allocation.

3.1 Overview

Accordion introduces a *compacting* memory store to the LSM store design framework. In contrast to the traditional memory store, which maintains RAM-resident data in a single monolithic data structure, Accordion manages data as a *pipeline* of *segments* ordered by creation time. Each segment contains an index over a collection of data cells. At all times, the most recent segment, called *active*, is mutable; it absorbs put operations. The rest of the segments are immutable. In addition to searching data in the disk store, get and scan operations traverse all memory segments, similarly to a traditional LSM store read from multiple files. Memory segments do not maintain Bloom filters like HFiles do, but they maintain other metadata like key ranges and time ranges to eliminate redundant reads. In addition, it is possible to search in multiple memory segments in parallel.

Figure 2 illustrates the Accordion architecture. It is parameterized by two values:

- A – fraction of the memory store allocated to the active segment; and
- S – upper bound on the number of immutable segments in the pipeline.

As our experiments show (Section 4), the most effective parameter values are quite small, e.g., $0.02 \leq A \leq 0.05$, and $2 \leq S \leq 5$.

Once the active segment grows to its size bound (a fraction A of the memory store's size bound), an *in-memory flush* is invoked. The in-memory flush makes the active segment immutable and creates a new active segment to replace it.

In case there is available space in the pipeline (the number of pipeline segments is smaller than S), the replaced active segment is simply *flattened* and added to the pipeline. Flattening a segment involves replacing the dynamic segment index (e.g., skiplist) by a compact ordered array suitable for immutable data, as shown in Figure 3. The indexed data cells are unaffected by the index flattening.

The flat index is more compact than a skiplist, and so reduces the MemStore's memory footprint, which delays disk flushes, positively affecting both read latency (by increasing the MemStore's hit rate) and write volume. It is also cache- and GC-friendly, and supports fast lookup via binary search. In managed environments, it can be allocated in off-heap (unmanaged) memory, which can improve performance predictability as discussed in Section 3.4 below.

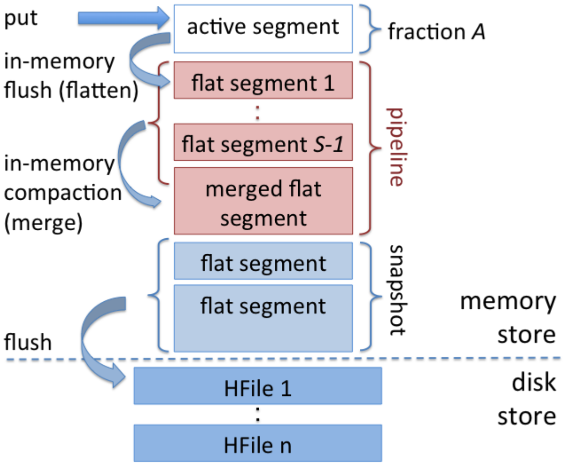


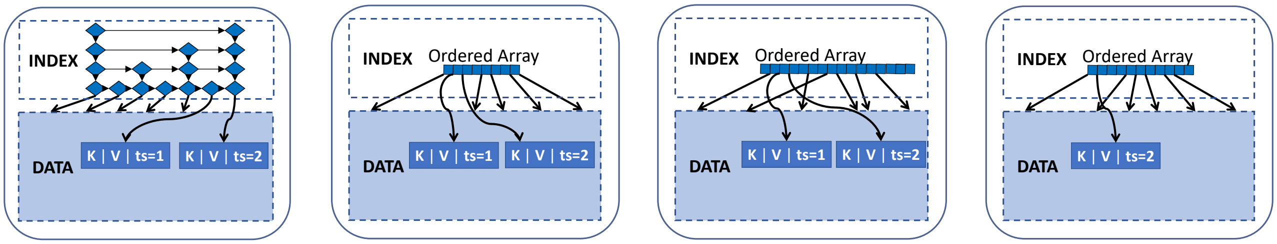
Figure 2: **Accordion's compacting memory store architecture adds a pipeline of flat segments between the active segment and the snapshot. The memory store includes a small dynamic active segment and a pipeline of flat segments. A disk flush creates a snapshot of the pipeline for writing to disk.**

Once the number of immutable segments exceeds S, the segments in the pipeline are processed to reduce their number. At a minimum, an *in-memory merge* replaces the indices of multiple segments by a single index covering data that was indexed by all original segments, as shown in Figure 3c. This is a lightweight process that results in a single segment but does not eliminate redundant data versions. For example, if a cell with the same row key and column key is stored in two different segments in the pipeline (with two different timestamps) then after the merge both cells will appear consecutively in the merged segment.

Optionally, an *in-memory compaction* can further perform *redundant data elimination* by creating a single flat index with no redundancies and disposing redundant data cells, as shown in Figure 3d. In the example above only the more recent cell will appear in the compacted segment. In case the memory store manages its cell data storage internally (via MSLAB), the surviving cells are relocated to a new chunk (to avoid internal chunk fragmentation). Otherwise, the redundant cells are simply de-referenced, allowing the garbage-collector to reclaim them. The choice whether to eliminate redundant data (i.e., perform compaction) or not (perform only merge) is guided by the policies described in Section 3.2 below.

Flushes to disk work the same way as in a standard LSM store: A disk flush first shifts all pipeline segments to the snapshot, which is not part of the pipeline, while the pipeline is emptied so that it may absorb new flat segments. A background flush process merges all snapshot segments while eliminating redundancies, and streams the result to a new file. After the file is written, the snapshot segments are freed.

In case the disk flush process empties the pipeline while an in-memory compaction is attempting to merge some segments, the latter aborts. This behavior is valid since in-memory compactions are an optimization.



(a) Active segment (b) Flat segment (c) Merged segment (d) Compacted segment

Figure 3: **The active segment has a skiplist index. After a segment is added to the pipeline its index is flattened into an ordered array index. When the pipeline consists of S segments, they are either merged or compacted. During a flattening or merge processes the data remains untouched. During a compaction indices are merged and redundancies in the data are eliminated. The key K was updated twice on timestamp $ts = 1$ and later on timestamp $ts = 2$. Only in compacted segment the redundant version ($ts = 1$) is eliminated.**

3.2 Compaction Policies

Redundant data elimination induces a tradeoff. On the one hand, merging only search indices without removing redundancies is a lighter-weight process. Moreover, this approach is friendly to the managed memory system because the entire segment is freed at once, whereas removing (redundant) cells from existing segments burdens the memory management system (in particular, the garbage collector) by constantly releasing small objects. On the other hand, by forgoing redundant data elimination we continue to consume memory for overwritten data; this is significant in production-like heavy-tailed distributions where some keys are frequently overwritten. Removing these redundancies further delays disk flushes, which both improves read latency (thanks to more queries being satisfied from memory) and reduces the write volume.

Our HBase 2.0 implementation includes the two extreme memory compaction policies:

Basic (low-overhead) never performs redundant data elimination. Rather, once a segment becomes immutable, flattens its index, and once the pipeline size exceeds S, merges all segment indices into one.

Eager (high-overhead, high-reward under self-similar workloads) immediately merges a segment that becomes immutable with the current (single) pipeline segment, while eliminating redundant data.

Our experiments (reported in the next section) show that the *Eager* policy is typically too aggressive, in particular when A is small, and the benefits from reducing the memory footprint are offset by the increased management (and in particular, garbage collection) overhead. We therefore present in this paper a third policy:

Adaptive (the best of all worlds) a heuristic that chooses whether to eliminate redundant data (as in *Eager*) or not (as in *Basic*) based on the level of redundancy in the data and the perceived cost-effectiveness of compaction. *Adaptive* works at the level of a single LSM store, i.e., triggers redundancy elimination only for those stores where positive impact is expected.

Adaptive uses two parameters to determine whether to perform data redundancy elimination:

1. A throttling parameter t grows with the amount of data that can benefit from redundancy elimination. Initially, $t = 0.5$; it then grows exponentially by 2% with the number of in-memory flushes, and is reset back to the default value (namely 0.5) upon disk flush. Thus, t is bigger when there is more data in the MemStore.

2. The *uniqueness* parameter, u, estimates the ratio of unique keys in the memory store based on the fraction of unique keys encountered during the previous merge of segment indices.

Note that the accuracy of u at a given point in time depends on the number of merges that occurred since the last disk flush or data-merge. Initially, u is zero, and so the first in-memory compaction does not employ data-merge. Then, the estimate is based on the S merged components, which at the time for the second in-memory compaction is roughly one half of the relevant data, since the pipeline holds $S - 1$ unmerged components. Over time, u becomes more accurate while t grows.

Adaptive triggers redundancy elimination with probability t if the fraction of redundant keys $1 - u$ exceeds a parameter threshold R. The rationale for doing so is that prediction based on u becomes more accurate with time, hence compactions become more important because the component is bigger and more space can be saved.

WAL Truncation Since in-memory compaction delays disk flushes, it can cause HLog files (WAL) to become longer and in some rare cases even unbounded. Although the system protects against such scenarios by forcing flush to disk whenever the HLog size exceeds a certain threshold, this threshold is fairly high, and it is better to try and keep the HLog files as small as possible without losing data. To facilitate effective WAL truncation, Accordion maintains the oldest version number (smallest timestamp) of any key in the MemStore. This number monotonically increases due to in-memory compactions, which eliminate old versions (and specifically, the oldest version of each key). Whenever this number increases, the WAL is notified and old log files are collected.

3.3 Concurrency

A compacting memstore is comprised of an active segment and a double-ended queue (pipeline) of inactive segments. The pipeline is accessed by read APIs (get and scan), as well as by background disk flushes and in-memory compactions. The latter two modify the pipeline by adding, removing, or replacing segments. These modifications happen infrequently.

The pipeline's readers and writers coordinate through a lightweight copy-on-write, as follows. The pipeline object is versioned, and updates increase the version.

Reads access the segments lock-free, through the version obtained at the beginning of the operation. If an in-memory flush is scheduled in the middle of a read, the active segment may migrate into the pipeline. Likewise, if a disk flush is scheduled in the middle of a read, a segment may migrate from the pipeline to the pre-flush snapshot buffer. The correctness of reads is guaranteed by first taking the reference of the active segment then the pipeline segments and finally the snapshot segments. This way, a segment may be encountered twice but no data is lost. The scan algorithm filters out the duplicates.

Each modification takes the following steps: (1) promotes the version number; (2) clones the pipeline, which is a small set of pointers, (3) performs the update on the cloned version, and (4) uses a compare-and-swap (CAS) operation to atomically swap the global reference to the new pipeline clone, provided that its version did not change since (1). Note that cloning is inexpensive – only the segment references are copied since the segments themselves are immutable. For example, in-memory compaction fails if a disk flush concurrently removes some segments from the pipeline.

3.4 Off-Heap Allocation

As explained above, prior to Accordion, HBase allocated its MemStore indicies on-heap, using a standard Java skiplist for the active buffer and the snapshot. Accordion continues to use the same data structure – skiplist – for the active segment, but adopts arrays for the flat segments and snapshot. Each entry in an active or flat segment's index holds a reference to a cell object, which holds a reference to a buffer holding a key-value pair (as pojo or in MSLAB), as illustrated in Figure 4a. HBase MSLAB chunks may be allocated off-heap.

Accordion's *serialized* version takes this approach one step further, and allocates the flat segment index using MSLAB as well as the data. A serialized segment has its index and data allocated via the same MSLAB object, but on different chunks. Each MSLAB may reside either on- or off-heap. The frequent in-memory flushes are less suitable for serialized segments, because upon each in-memory flush the new serialized segment and thus new index chunk is allocated. When the index is not big enough to populate the chunk, the chunk is underutilized.

Serialized segments forgo the intermediate cell objects, and have array entries point directly to the the chunks holding keys and values, as illustrated in Figure 4b. Removing the cell objects yields a substantial space reduction, especially when data items are small, and eliminates the intermediate level of indirection.

Offloading data from the Java heap has been shown to be effective for read traffic [13]. However, it necessitates recreating temporary cell objects to support HBase's internal scan APIs. Nevertheless, such temporary objects consume a small amount of space on-demand, and these objects are deallocated rapidly, making them easier for the GC process to handle.

4. PERFORMANCE STUDY

We fully implemented Accordion in HBase, and it is generally available in HBase 2.0 and up. We now compare Accordion in HBase to the baseline HBase MemStore implementation. Our evaluation explores Accordion's different policies and configuration parameters. We experiment with two types of production machines with directly attached SSD and HDD storage. We exercise the full system with multiple regions, layered over HDFS as in production deployments.

We present the experiment setup in Section 4.1 and the evaluation results in Section 4.2.

4.1 Methodology

Experiment setup Our experiments exploit two clusters with different hardware types. The first consists of five 12-core Intel Xeon 5 machines with 48GB RAM and 3TB SSD storage. The second consists of five 8-core Intel Xeon E5620 servers with 24GB RAM and 1TB HDD storage. Both clusters have a 1Gbps Ethernet interconnect. We refer to these clusters as SSD and HDD, respectively.

In each cluster, we use three nodes for HDFS and HBase instances, which share the hardware. The HDFS data replication ratio is 3x. HBase exploits two machines as region servers, and one as a master server. The workload is driven by the two remaining machines, each running up to 12 client threads.

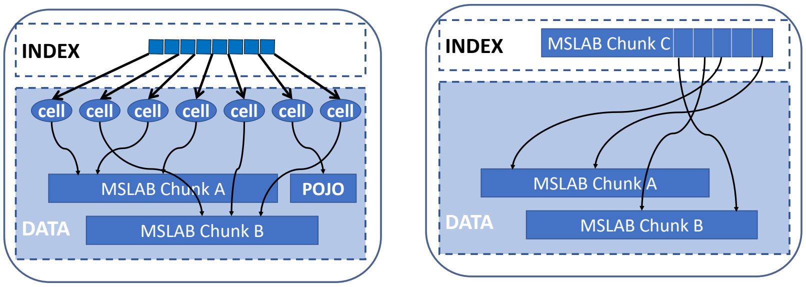
A region server runs with a 8GB heap, under G1GC memory management. For serialized segment experiments, we use MSLAB memory management. We use the default memory layout, which allocates 40% of the heap (roughly 3GB) to the MemStore area, and 40% more to the read-path block cache. We apply an asynchronous WAL in order to focus on real-time write-intensive workloads (a synchronous WAL implies an order-of-magnitude slower writes). The log aggregation period is one second.

Workloads Data resides in one table, pre-split into fifty regions (i.e., each region server maintains twenty-five regions). The table has a single column family with four columns. We populate cells with 25-byte values, and so each row comprises 100 bytes; this is a typical size in production workloads [35].

We seek to understand Accordion's performance impact in large data stores. We therefore perform 300M–500M writes, generating 30–50GB of data in each experiment. As a result, experiments are fairly long – the duration of a single experiment varies from approximately 1.5 hours to over 12 hours, depending on the setting (workload, algorithm, and hardware type). In addition, given that Accordion mostly impacts the write-path, we focus our study mainly on write-intensive workloads. For completeness, we also experiment with read-dominated workloads, in order to show that reads (gets and scans) benefit from or are indifferent to the change in the write-path.

We use the popular YCSB benchmarking tool [23] to generate put, get, and scan requests. Each put writes a full row (4 cells, 100 bytes). Each get retrieves a single cell, while scans retrieve all four cells of every scanned row. The length of each scan (number of retrieved cells) is chosen uniformly at random in the range 1–100. In order to produce a high load, updates are batched on the client side in 10KB buffers.

In each experiment, all operations draw keys from the



(a) Flat segment

(b) Serialized segment

Figure 4: **The difference between the flat segment and the serialized segment. The ordered array of the cell objects cannot be directly serialized and streamed into an off-heap chunk. The transformation shrinks the cell object into just 20 bits, and writes on the index chunk.**

same distribution over a key range of 100M items. We experiment with two distributions: heavy-tailed (Zipf) and uniform (the latter is less representative of real workloads and is studied for reference only). The Zipf distribution is generated following the description in [26], with $\theta = 0.99\%$ (YCSB standard).

Measurements Our experiments measure HBase's throughput and latency in the different settings. To reason about the results, we explore additional signals such as I/O statistics and GC metrics. While in write-only workloads we measure performance throughout the experiment, we precede each mixed-workload experiment (read/write or scan/write) with a write-only load phase that populates the table; performance of the load-phase is not reported.

We repeated each experiment (with a particular workload, particular hardware, and particular parameter settings) a handful of times (usually five) to verify that the results are not spurious. The median result is reported.

4.2 Evaluation Results

We compare the different Accordion policies (*Basic*, *Adaptive*, and *Eager*) to the legacy MemStore implementation, to which we refer as *NoCompaction*. Here, *Adaptive* is studied under multiple redundancy thresholds: $R = 0.2$, $R = 0.35$ and $R = 0.5$ (the smaller the ratio, the more aggressively the policy triggers in-memory compaction).

Accordion's parameters A (active segment fraction bound) and S (pipeline number of segments bound) are tuned to values that optimize performance for the *Basic* version of Accordion. We use $A = 0.02$ and $S = 5$. Section 4.2.4 presents our parameter exploration.

4.2.1 Write-Only Workloads

Our first benchmark set exercises only put operations. It starts from an empty table and performs 500M puts, in parallel from 12 client threads. We study the Zipf and uniform key distributions. The experiments measure write throughput (latency is not meaningful because puts are batched), as well as disk I/O and GC metrics.

Write throughput Figure 5 depicts the performance speedup of *Basic* and *Adaptive* over *NoCompaction*. Table 1 provides the absolute throughput numbers. For the Zipf benchmark, the maximal speedup is 47.6% on SSD and 25.4% on HDD. For the uniform benchmark, which has neither redundancy nor locality, they are 23.8% and 8.9%, respectively. The gain is significantly larger for the system with SSD storage, which is much faster on the I/O side. Being more CPU-bound than I/O-bound, its write throughput is more dependent on the MemStore speed than on background writes to the file system.

Surprisingly, the *Basic* policy, which flattens and merges segment indices but avoids redundancy elimination, yields the largest throughput gain. We explain this as follows. By reducing the active segment size to $A = 2\%$ of the MemStore size, all Accordion policies improve insertion time of new versions into the dynamic index through improved locality of access and search time in the skiplist. However, they affect the garbage collection in different ways. *Basic* adds negligible overhead to garbage collection by recycling the index arrays. Similarly to *NoCompaction*, it releases memory in big bursts upon disk flush.

Eager is not included in this figure, because its performance with these parameters is below the baseline, as we show in the sequel. It eliminates redundant data at a constant high rate, which is less friendly for generational GC. And as we further show below, GC overhead is a strong predictor for overall performance.

Adaptive strikes a balance between *Basic* and *Eager*. Its performance is much closer to *Basic* because it is workload-driven and selective about the regions in which it performs in-memory compaction.

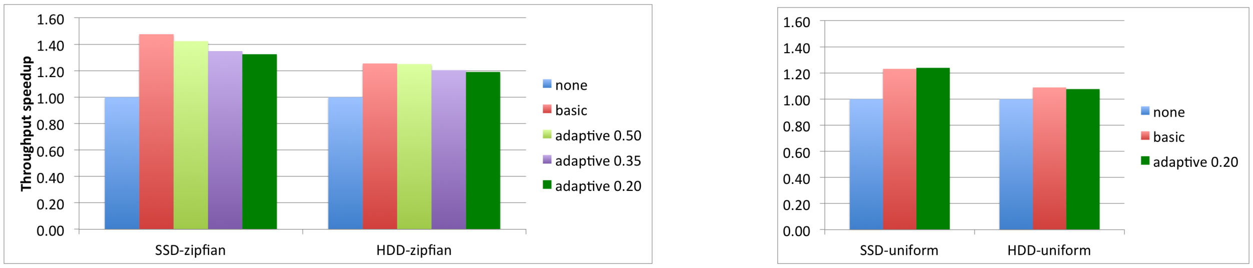
Note that *Adaptive* is only marginally slower than *Basic*'s (3% to 7% throughput for the Zipf distribution on SSD, immaterial in other cases). In what follows, we substantiate its savings on the I/O side. Going forward, we focus on the more realistic Zipf workload.

I/O metrics With respect to write amplification, the picture is reversed. In this regard, *Basic* provides modest savings. It writes 13% less bytes than *NoCompaction* on SSD, and 8% on HDD. In contrast, *Adaptive* slashes the number of flushes and compactions by 58% and 61%, respectively, and the number of bytes written by almost 30%. As expected, the lower the redundancy estimate threshold R, the more aggressive the algorithm, and consequently, the higher the savings. Figure 6 and Table 2 summarize the results.

Note that the amount of data written to WAL (approximately 45% of the write volume) remains constant across all experiments. HBase operators often choose placing log files on inexpensive hard disks, since they are only needed for

Table 1: **Write throughput (operations per second) of the best Accordion policy (*Basic*) vs *NoCompaction*, across multiple key distributions and disk hardware types.**

	Zipf, SSD	Zipf, HDD	Uniform, SSD	Uniform, HDD
NoCompaction	75,861	60,457	74,971	35,342
Basic	115,730	78,392	92,816	38,488



(a) Zipf distribution

(b) Uniform distribution

Figure 5: **Write throughput speedup vs *NoCompaction* (HBase legacy MemStore) achieved by Accordion with *Basic* and *Adaptive* policies. Measured on the Zipf and uniform key distributions, on SSD and HDD hardware. In SSD systems, *Basic* increases the write throughput by close to 48%.**

recovery, and in this case disk wear is not a major concern for WAL data.

All-in-all, there is a tradeoff between the write throughput and storage utilization provided by the different Accordion policies. *Adaptive* provides multiple operating points that trade major storage savings for minor performance losses.

Garbage collection We saw above that redundancy elimination proved to be less impactful than memory management overhead, especially in systems with fast SSD hardware. Figure 7, which studies GC overhead for a range of policies and tuning parameters, further corroborates this observation. It shows a clear negative correlation between the GC time and the write throughput. Note that this does not necessarily mean that GC time is the only reason for performance reduction. Rather, GC cycles may be correlated with other costs, e.g., memory fragmentation, which reduces locality.

Serialized and off-heap segments We also experiment with the write-only workload using serialized segments, whose indices are allocated on MSLAB chunks (as described in Section 3.4). We compare them to un-serialized flat segments, to which we refer here shortly as *flat*. We concentrate on the Zipf key distribution and SSD hardware. In these experiments we run with synchronous WAL. We run two sets of experiments: in the first all MSLAB chunks are allocated on-heap, and in the second – off-heap. For the serialized segments we set parameter A to 0.1, for less frequent in-memory flushes. Figure 8 presents the throughput speedup using flat and serialized segments with the *Basic* policy over *NoCompaction*. For the flat *Basic*, the speedup is 27% on-heap and 29% off-heap. For the serialized *Basic*, the speedup is 33% on-heap and 44% off-heap. The gain from the serialized *Basic* is larger for the off-heap case, because when the index is taken off-heap the JVM GC has less work to do.

4.2.2 Read-Write Workload

Our second benchmark studies read latencies under heavy write traffic. We run batched puts from 10 client threads and single-key gets from two other threads. The keys of all operations are distributed Zipf. The workload is measured after loading 30GB of data (300M write operations) to make sure most if not all keys have some version on disk. We measure the 50th, 75th, 90th, 95th and 99th get latency percentiles. Figure 9 depicts the relative latency slowdown of *Basic* and *Adaptive* ($R = 0.2$) versus *NoCompaction*.

The HDD systems enjoy a dramatic reduction in tail latencies (15% to 40% for *Adaptive*). We explain this as follows. The tail latencies are cache misses that are served from disk. The latter are most painful with HDDs. Thanks to in-memory compaction, *Adaptive* prolongs the lifetime of data in MemStore. Therefore, more results are served from MemStore, i.e., there are less attempts to read from cache, and consequently also from disk.

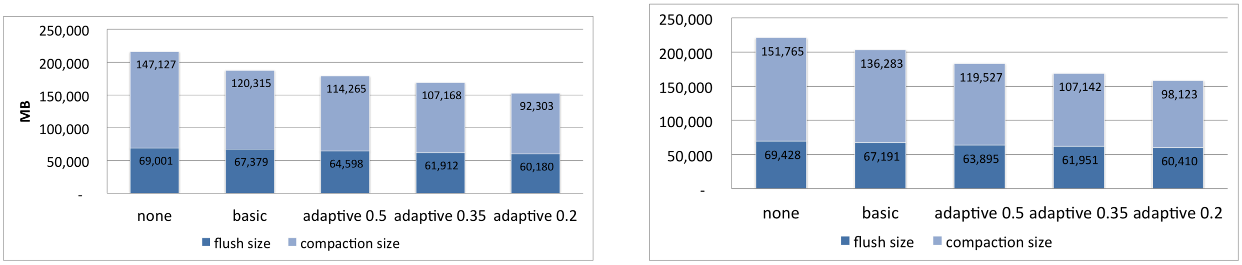
In SSD systems, the get latencies are marginally slower in all percentiles (e.g., the 95th percentile is 2.13ms in *NoCompaction* versus 2.26ms in *Adaptive*). This happens because most reads are dominated by in-memory search speed, which depends on the number of segments in the MemStore pipeline, which was set to 5 in this experiment. The other factor is increased GC overhead in *Adaptive* that stems from redundancy elimination.

4.2.3 Scan-Write Workload

Finally, we study scans on SSD under Zipf key distribution. We exercise the standard YCSB scans workload (workload *e*), where 95% of $500,000$ operations are short range queries (50 keys in expectation), and the remaining 5% are put operations. The workload is measured after loading 50GB of data (500M write operations). The latency slowdown of *Basic* over *NoCompaction* is presented in Figure 10. Here we also experiment with different pipeline sizes (parameter S). With only two segments in the pipeline ($S = 2$) the search for each key is a bit faster than with a longer pipeline ($S = 5$). We can see that the slowdown for the short pipeline is negligible – up to 3%. When the pipeline may grow to five segments, the performance gap slightly increases to $2\% - 8\%$. These results resemble the SSD reads slowdown.

Table 2: **Number of flushes and compactions, measured for *NoCompaction* vs Accordion policies, for the Zipf key distribution.**

Policy	#flushes, SSD	#compactions, SSD	#flushes, HDD	#compactions, HDD
NoCompaction	1468	524	1504	548
Basic	1224	355	1210	443
Adaptive (R=0.5)	922	309	879	316
Adaptive (R=0.35)	754	261	711	248
Adaptive (R=0.2)	631	209	630	216



(a) SSD

(b) HDD

Figure 6: **Bytes written by flushes and compactions, measured for *NoCompaction* vs Accordion policies, for the Zipf key distribution. The *Adaptive* policy provides double-digit savings for disk I/O.**

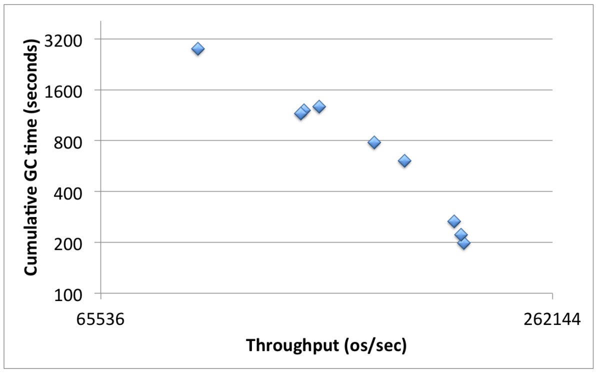


Figure 7: **Cumulative GC time vs write throughput, measured on SSD machines for multiple Accordion policies and configurations, for the Zipf key distribution. Each data point depicts the throughput vs GC time of one experiment. Both axes are in log-log scale. The graph shows a clear negative correlation between the two metrics.**

4.2.4 Parameter Tuning

One of the main sources of memory management overhead is the skiplist data structure used to index the active memory segment. Not only is it bigger in size compared to a flat index, it is also fragmented whereas a static index is stored in a consecutive block of memory. Therefore, flat storage incurs smaller overhead in terms of allocation, GC, and cache misses. We first tune the size of this data structure.

We evaluate the *Basic* policy with different bounds on the active segment fraction: $A = 0.25$, 0.1, 0.05, and 0.02. *NoCompaction* has no static memory segments, hence its throughput is designated with a single point $A = 1$. We measure throughput in a write-only Zipf workload, on SSD, for the four values of A, and with a pipeline size $S = 2$. The throughput of all five runs for each segment size are depicted in Figure 11. The store scales as the active segment memory fraction decreases.

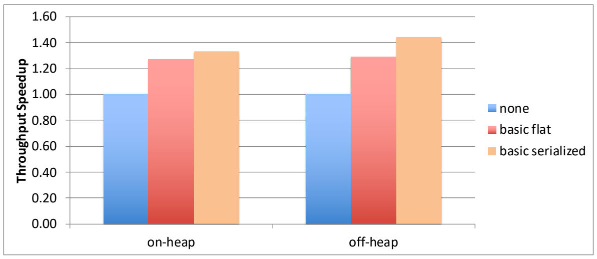
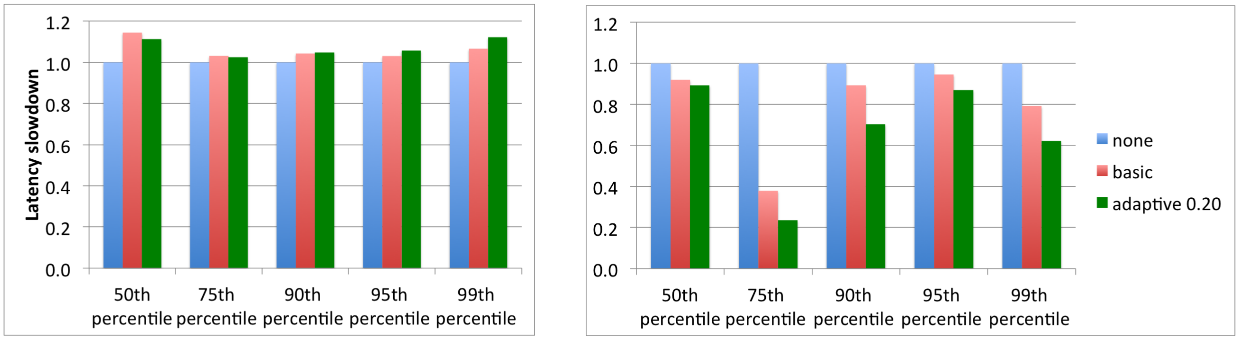


Figure 8: **Throughput speedup when using flat (unserialized) and serialized segments allocated on MSLAB, off-heap and on-heap.**

The next parameter is the pipeline size. When A is less than 10%, it is clear that merging the active segment into the much bigger static data structure over and over again can be inefficient, as it creates a new index and releases the old one. The alternative is to batch multiple segments ($S > 1$) in the pipeline before merging them into a single segment. This throttles the index creation rate. On the flip side, the read APIs must scan all the segments, which degrades their performance. Figure 12 depicts the write-only throughput results as a function of the number of segments in the pipeline. The peak throughput is achieved with $S = 5$ on SSD and $S = 4$ on HDD. In Figure 10 above, we observed the effect of the parameter S on read performance – we saw that the gap in scan latency between $S = 5$ and $S = 1$ is at most 5%.

Parameter Tuning for *Eager* With the preferred active segment size chosen for *Basic* ($A = 0.02$), *Eager* performs poorly since compacting the data so frequently is inefficient. Figure 13 shows that when the active segment is much bigger ($A = 0.25$), *Eager* performs better, and outperforms the



(a) SSD

(b) HDD

Figure 9: **Read latency speedup (respectively, slowdown) of *Basic* and *Adaptive* versus *NoCompaction*, under high write contention. Latencies are broken down by percentile. In HDD systems, *Adaptive* delivers up to 40% tail latency reduction.**

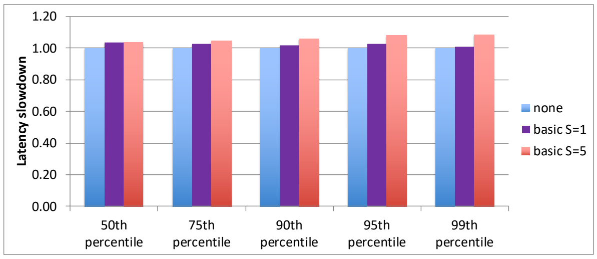


Figure 10: **Latency speedup for scans of *Basic*.**

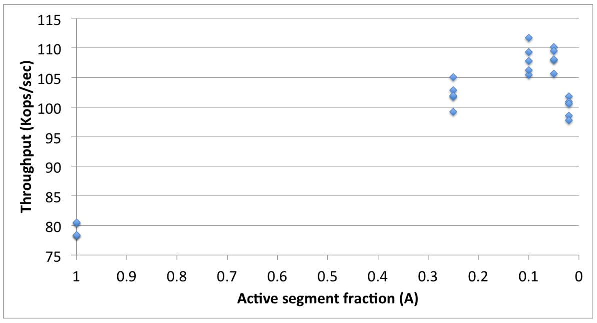


Figure 11: **Tuning of the active segment memory fraction, A, for the *Basic* policy, with $S = 2$, Zipf distribution on SSD. Five experiments are conducted for each A. The write throughput is higher for small values of A.**

baseline by 20%. We also discovered that the write volume for *Eager* is roughly the same in these two settings, whereas *Basic*'s write volume increases with A; (these results are not shown). Since *Adaptive* is always superior to *Eager*, we excluded *Eager* from the experiments reported above.

5. RELATED WORK

The basis for LSM data structures is the *logarithmic method* [21]. It was initially proposed as a way to efficiently transform static search structures into dynamic ones. A *binomial list* structure stored a sequence of sorted arrays, called *runs* each of size of power of two. Inserting an element triggers a cascaded series of merge-sorting of adjacent runs. Searching an element is done by applying a binary search on the runs starting with the smallest run until the element is found. AlphaSort [30] further optimizes a sort algorithm by taking advantage of the entire memory hierarchy (from registers to disk); each run fits into the size of one level in the hierarchy.

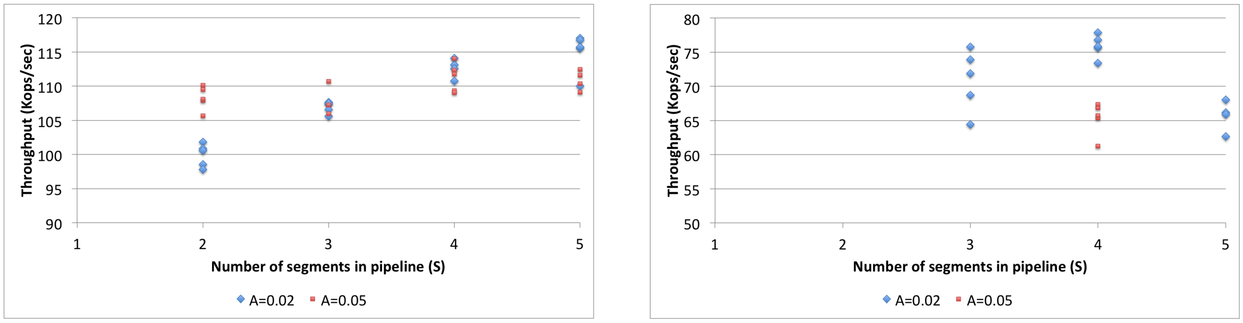
These methods inspired the original work on *LSM-trees* [31] and its variant for multi-versioned data stores [29].

Because compaction frequency and efficiency has a significant impact on LSM store performance and disk wear, many previous works have focused on tuning compaction parameters and scheduling [6, 14, 17, 18, 32] or reducing their cost via more efficient on-disk organization [28]. However, these works focus on *disk compactions*, which deal with data *after* it has been flushed to disk. We are not aware of any previous work that added *in-memory* compactions to LSM stores.

Some previous works have focused on other in-memory optimizations. cLSM [25] focused on scaling LSM stores on multi-core hardware by adding lightweight synchronization to LevelDB. Accordion, on the other hand, is based on HBase, which already uses a concurrent in-memory skiplist; improving concurrency in the line of cLSM is orthogonal to our contribution.

Facebook's RocksDB [14] is the state-of-the-art implementation of a key-value store. All writes to RocksDB are first inserted into an in-memory active segment (called memtable). Once the active segment is full, a new one is created and the old one becomes immutable. At any point in time there is exactly one active segment and zero or more immutable segments [15]. Unlike Accordion immutable segments take the same form of mutable segments. Namely, there is no effort done to flatten the layout of the indices of immutable segments or to eliminate redundant data versions in any way while it is in-memory.

Similarly to Accordion, the authors of FloDB [20] also observed that using large skiplists is detrimental to LSM store performance, and so FloDB also partitions the memory component. However, rather than using a small skiplist and large flat segments as Accordion does, FloDB uses a small hash table and a large skiplist. Put operations inserting data to the small hash table are much faster than insertions to the large skiplist, and the latter are batched and inserted via multi-put operations that amortize the cost of the slow skiplist access. Unlike Accordion, FloDB does not reduce the memory footprint by either flattening or compaction.



(a) SSD

(b) HDD

Figure 12: **Tuning the pipeline size bound, S, for the *Basic* policy. Five experiments are conducted for each S.**

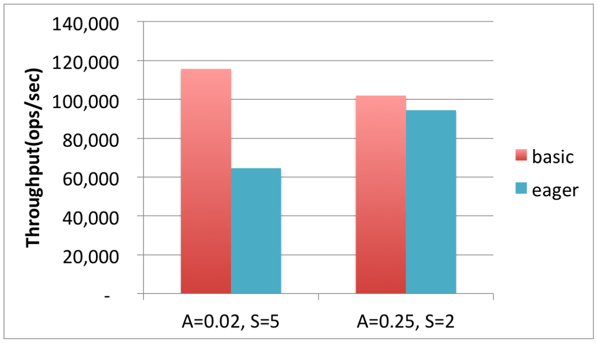


Figure 13: **Throughput (op/sec): *Eager* versus *Basic* in different settings.**

LSM-trie [35] is motivated by the prevalence of small objects in industrial NoSQL workloads. Like us, they observed that skiplists induce high overhead when used to manage many small objects. They suggest to replace the LSM store architecture with an LSM trie, which is based on hashing instead of ordering and thus eliminates the memory for indexing. This data structure is much faster, but only for small values, and, more importantly, does not support range queries.

6. DISCUSSION

While disk compactions in LSM stores have gotten a lot of attention, their in-memory organization was, by and large, ignored. In this work, we showed that applying the LSM principles also to RAM can significantly improve performance and reduce disk wear. We presented Accordion, a new memory organization for LSM stores, which employs *in-memory* flushes and compaction to reduce the LSM store's memory footprint and improve the efficiency of memory management and access. We integrated Accordion in Apache HBase following extensive testing. It is available in HBase 2.0 and up.

In this paper, we evaluated Accordion with different storage technologies (HDD and SSD) and various compaction policies, leading to some interesting insights and surprises:

Flattening and active component size We showed that flattening the in-memory index reduces the memory management overhead as well as the frequency of disk writes, thus improving performance and also reducing disk wear-out. Originally, we expected an active segment (skiplist) comprising 10–20% of the memory store to be effective, and anticipated that smaller active components would result in excessive flush and compaction overhead. Surprisingly, we found that performance is improved by using a much smaller active component, taking up only 2% of the memory store. This is due in part to the fact that our workload is comprised of small objects – which are common in production workloads [35] – where the indexing overhead is substantial.

Impact of memory management on write throughput We showed that the write volume can be further reduced, particularly in production-like self-similar key access distributions, by merging memory-resident data, namely, eliminating redundant objects. We expected this reduction in write volume to also improve performance. Surprisingly, we found that disk I/O was not a principal bottleneck, even on HDD. In fact, write throughput is strongly correlated with GC time, regardless of the write volume.

Cost of redundant data elimination We expected redundant data elimination to favorably affect performance, since it frees up memory and reduces the write volume. This has led us to develop *Eager*, an aggressive policy that frequently performs in-memory data merges. Surprisingly, this proved detrimental for performance. The cost of the *Eager* policy was exacerbated by the fact that we benefitted from a significantly smaller active component than originally expected (as noted above), where it literally could not keep up with the in-memory flush rate. This led us to develop a new policy, *Adaptive*, which heuristically decides when to perform data merges based on an estimate of the expected yield. While *Adaptive* does not improve performance compared to the *Basic* policy, which merges only indices and not data, it does reduce the write volume, which is particularly important for longevity of SSD drives.

Deployment recommendations Following the set of experiments we ran, we can glean some recommendations for selecting policies and parameter values. The recommended compaction policy is *Adaptive*, which offers the best performance vs disk-wear tradeoff. When running without MSLAB (i.e., with a flat index), an active segment size of $A = 0.02$ offered the best performance in all settings we experimented with, though bigger values of A might be most appropriate in settings with big values (and hence lower meta-data-to-data ratios). When using MSLAB allocation, it is wasteful to use such a small active segment, because an entire chunk is allocated to it, and so we recommend using $A = 0.1$. Assuming the workload is skewed (as production workloads usually are), we recommend using a fairly aggressive compaction threshold $R = 0.2$. If the workload is more uniformly distributed or if write throughput is more crucial than write amplification, then one can increase the value R up to 0.5 or even higher to match the throughput of *Basic*. The pipeline size S induces a tradeoff between read and write performance. If the workload is write-intensive then we recommend using $S = 5$, and if the workload is read-oriented $S = 2$ is better. We note that most of the recommendations here are set as default values in the coming HBase 2.0 release.

Acknowledgments

We are thankful to the members of the HBase community for many helpful discussions during the development of the Accordion project. In particular, we would like to thank Michael Stack, Ramakrishna Vasudevan, Anoop Sam John, Duo Zhang, Chia-Ping Tsai, and Ted Yu.

7. REFERENCES

[1] Apache HBase. http://hbase.apache.org.

[2] Apache hbase at airbnb. https://www.slideshare.net/HBaseCon/apache-hbase-at-airbnb.

[3] Avoiding full gcs with memstore-local allocation buffers. http://blog.cloudera.com/blog/2011/02/.

[4] Cassandra. http://cassandra.apache.org.

[5] Dragon: A distributed graph query engine. https://code.facebook.com/posts/1737605303120405.

[6] Hbase compaction tuning tips. https://community.hortonworks.com/articles/52616/hbase-compaction-tuning-tips.html.

[7] Hbase operations in a flurry. http://bit.ly/2yaTfoP.

[8] Java concurrent skiplist map. https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentSkipListSet.html.

[9] LevelDB. https://github.com/google/leveldb.

[10] Mongodb. https://mongorocks.org.

[11] Mysql. http://myrocks.io.

[12] Off-heap memtables in cassandra 2.1. https://www.datastax.com/dev/blog/off-heap-memtables-in-cassandra-2-1.

[13] Offheap read-path in production the alibaba story. https://blog.cloudera.com/blog/2017/03/.

[14] RocksDB. http://rocksdb.org/.

[15] Rocksdb tuning guide.

[16] Scylladb. https://github.com/scylladb/scylla.

[17] Sstable compaction and compaction strategies. http://bit.ly/2wXc7ah.

[18] Universal compaction.

[19] Zen: Pinterest's graph storage service. http://bit.ly/2ft4YDx.

[20] O. Balmau, R. Guerraoui, V. Trigonakis, and I. Zablotchi. Flodb: Unlocking memory in persistent key-value stores. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 80–94, New York, NY, USA, 2017. ACM.

[21] J. L. Bentley and J. B. Saxe. Decomposable searching problems i. static-to-dynamic transformation. *Journal of Algorithms*, 1(4):301–358, 1980.

[22] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[23] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[24] P. Devineni, D. Koutra, M. Faloutsos, and C. Faloutsos. If walls could talk: Patterns and anomalies in facebook wallposts. In *Proceedings of the 2015 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining 2015*, ASONAM '15, pages 367–374, New York, NY, USA, 2015. ACM.

[25] G. Golan-Gueta, E. Bortnikov, E. Hillel, and I. Keidar. Scaling concurrent log-structured data stores. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 32:1–32:14, New York, NY, USA, 2015. ACM.

[26] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly generating billion-record synthetic databases. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, SIGMOD '94, pages 243–252, New York, NY, USA, 1994. ACM.

[27] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka. Write amplification analysis in flash-based solid state drives. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, SYSTOR '09, pages 10:1–10:9, New York, NY, USA, 2009. ACM.

[28] L. Lu, T. S. Pillai, H. Gopalakrishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. *TOS*, 13(1):5:1–5:28, 2017.

[29] P. Muth, P. E. O'Neil, A. Pick, and G. Weikum. Design, implementation, and performance of the lham log-structured history data access method. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB '98, pages 452–463, 1998.

[30] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet. AlphaSort: A cache-sensitive parallel external sort. *The VLDB Journal*, 4(4):603–628, Oct. 1995.

[31] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Inf.*, 33(4):351–385, June 1996.

[32] R. Sears and R. Ramakrishnan. blsm: A general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 217–228, New York, NY, USA, 2012. ACM.

[33] A. S. Tanenbaum and H. Bos. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 4th edition, 2014.

[34] G. Wu, X. He, and B. Eckart. An adaptive write buffer management scheme for flash-based ssds. *Trans. Storage*, 8(1):1:1–1:24, Feb. 2012.

[35] X. Wu, Y. Xu, Z. Shao, and S. Jiang. Lsm-trie: An lsm-tree-based ultra-large key-value store for small data items. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 71–82, Santa Clara, CA, 2015. USENIX Association.