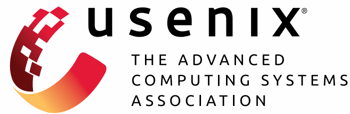



# TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores

**Oana Balmau, Diego Didona, Rachid Guerraoui, and Willy Zwaenepoel, *EPFL;***
**Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka, *Nutanix***

https://www.usenix.org/conference/atc17/technical-sessions/presentation/blamau

**This paper is included in the Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC '17).**

**July 12–14, 2017 • Santa Clara, CA, USA**

ISBN 978-1-931971-38-6

**Open access to the Proceedings of the 2017 USENIX Annual Technical Conference is sponsored by USENIX.**

# TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores

Oana Balmau *EPFL* Diego Didona *EPFL* Rachid Guerraoui *EPFL* Willy Zwaenepoel *EPFL*

Huapeng Yuan *Nutanix* Aashray Arora *Nutanix* Karan Gupta *Nutanix* Pavan Konka *Nutanix*

## Abstract

We present TRIAD, a new persistent key-value (KV) store based on Log-Structured Merge (LSM) trees. TRIAD improves LSM KV throughput by reducing the write amplification arising in the maintenance of the LSM tree structure. Although occurring in the background, write amplification consumes significant CPU and I/O resources. By reducing write amplification, TRIAD allows these resources to be used instead to improve user-facing throughput.

TRIAD uses a holistic combination of three techniques. At the LSM memory component level, TRIAD leverages skew in data popularity to avoid frequent I/O operations on the most popular keys. At the storage level, TRIAD amortizes management costs by deferring and batching multiple I/O operations. At the commit log level, TRIAD avoids duplicate writes to storage.

We implement TRIAD as an extension of Facebook's RocksDB and evaluate it with production and synthetic workloads. With these workloads, TRIAD yields up to 193% improvement in throughput. It reduces write amplification by a factor of up to 4x, and decreases the amount of I/O by an order of magnitude.

## 1 Introduction

Key-value (KV) stores [1, 3, 4, 12, 29, 35, 38, 39, 47] are nowadays a widespread solution for handling large-scale data in cloud-based applications. They have several advantages over traditional DBMSs, including simplicity, scalability, and high throughput. KV store applications include, among others, messaging [12, 18], online shopping [25], search indexing [22] and advertising [12, 22]. At Nutanix, we use KV stores for storing the metadata for our core enterprise platform, which serves thousands of customers with petabytes of storage capacity [21].

KV store systems are available for workloads that fit entirely in memory (e.g., Mica [36], Redis [3], and Memcached [2]), as well as for workloads that require persistent storage (e.g., LevelDB [4], RocksDB [12]). Log-Structured Merge trees (LSMs) [41, 40] are a popular design choice for the latter category. LSMs achieve high write throughput at the expense of a small decrease in read throughput. They are today employed in a wide range of KV stores such as LevelDB [4], RocksDB [12], Cassandra [1], cLSM [29], and bLSM [44].

Broadly speaking, LSMs are organized in two components: a *memory component* and a *disk component*. The memory component seeks to absorb updates. For applications that do not tolerate data loss in the case of failure, the updates may be temporarily backed up in a *commit log* stored on disk. When the memory component becomes full, it is flushed to persistent storage, and a new one is installed. The disk component is organized into levels, each level containing a number of sorted files, called *SSTables*. The levels closer to the memory component hold the fresher information. When level $L_i$ is full, one or more selected files from level $L_i$ are compacted into files at level $L_{i+1}$, discarding stale values. This compaction operation occurs in the background.

Compaction and flushing are key operations, responsible for maintaining the LSM structure and its properties. Unfortunately, they take up a significant amount of the available resources. For instance, for our production workloads at Nutanix, our measurements indicate that, at peak times, compaction can consume up to 45% of the CPU. Moreover, per cluster, an average of 2.5 hours per day is spent on compaction operations for the maps storing the metadata. Clearly, compaction and flushing pose an important performance challenge, even though they occur outside the critical path of user-facing operations.

We propose three new complementary techniques to close this gap. Our techniques reduce both the time and space taken by the compaction and flushing operations, leading to increased throughput. The first technique decreases compaction overhead for skewed workloads. We keep KV pairs that are updated frequently (i.e., hot en-

tries) in the memory component, and we only flush the cold entries. This separation eliminates frequent compactions triggered by different versions of the same hot entry. The main idea of the second technique is to defer file compaction until the overlap between files becomes large enough, so as to merge a high number of duplicate keys. Finally, our third technique avoids flushing the memory component altogether, by changing the role the commit logs play in LSMs and using them in a manner similar to SSTables.

Combined, our three techniques form TRIAD, a new LSM KV store we build on top of RocksDB. We extensively compare TRIAD against the original version of RocksDB on various synthetic workloads, with a focus on skewed workloads, as well as on Nutanix production workloads. TRIAD achieves up to 193% higher throughput than RocksDB. This improvement is the result of an order of magnitude decrease in I/O due to compaction and flushing, up to 4x lower write amplification and 77% less time spent compacting and flushing, on average.

To summarize, this paper makes the following key contributions: (1) the design of TRIAD, a system combining three complementary techniques for reducing compaction work in LSMs, each interesting in its own right, (2) a publicly available implementation of TRIAD as an extension of RocksDB, one of the most popular state-of-the-art LSM KV stores, and (3) an evaluation of its benefits in comparison to RocksDB.

**Roadmap.** The rest of the paper is structured as follows. Section 2 gives an overview of the LSM tree. Section 3 presents the background I/O overheads in LSM KV stores. Section 4 presents our three techniques for reducing the impact of the compaction and flushing operations on performance. Section 5 describes our evaluation results. Section 6 discusses related work. Section 7 concludes the paper.

## 2 Background on LSM

We provide an overview of the LSM structure, its user-facing operations and the flushing and compaction processes that take place in the background.

**LSM Structure.** The high-level view of a typical LSM-based KV store is shown in Figure 1. The system has three main components, which we briefly describe.

▷*Memory Component*. The memory component $C_m$ is a sorted data structure residing in main memory. Its purpose is to temporarily absorb the updates performed on the KV store. The size of the memory component is typically small, ranging from a few MBs to tens of MBs. When the memory component fills up, it is replaced by a new, empty component. The old memory component is then flushed as is to level 0 ($L_0$) of the LSM disk component.

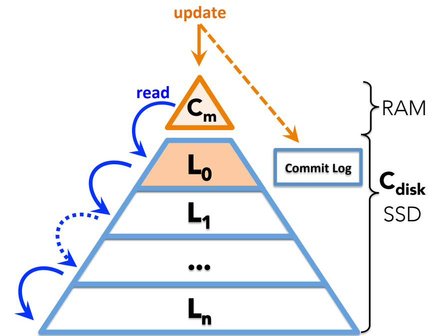



Figure 1: High-level view of a typical LSM KV store.

$L_0$ is a special level of the disk component hierarchy, described below.

▷*Disk Component*. The disk component $C_{disk}$ is structured into multiple levels ($L_0$, $L_1$, ...), with increasing sizes. Each level contains multiple sorted files, called *SSTables*. The memory component $C_m$ is flushed to the first level, $L_0$. Because of this, the SSTables in $L_0$ have overlapping key ranges. SSTables on levels $L_i$ $(i \neq 0)$ have disjoint key ranges. The choice of the number of levels in $C_{disk}$ is an interesting aspect of the LSM structure. From a correctness perspective, it would suffice to have only two levels on disk: one to flush memory components and one in which we compact. However, there is an I/O disadvantage to this approach. When we merge $L_0$ SSTables into $L_1$, we identify all the $L_1$ SSTables that have overlapping key ranges with the $L_0$ SSTable that is being compacted (the SSTables from $L_0$ cover the entire key range, because they are directly flushed from memory). If $L_1$ files are large, then fewer files would have overlapping key ranges, but we would have to re-write large files, leading to overhead in the compaction work and a penalty in terms of of memory use. If $L_1$ files are small, then a large number of files would have overlapping key ranges with the $L_0$ file. These files would need to be opened and rewritten, creating large overhead in the compaction work. The leveled structure allows LSMs to amortize the compaction work, as the updates trickle down the levels.

▷*Commit Log*. The commit log is a file residing on disk. Its purpose is to temporarily log the updates that are made to $C_m$ (in small batches), if the application requires that the data is not lost in case of a failure. All updates performed on $C_m$ are also appended to the commit log. Typically, the size of the commit log is kept small in order to provide fast recovery in case the operations need to be replayed to recover from a failure. A typical value for the size of the commit log is on the order of hundreds of MB.

**User-facing operations.** The main user-facing operations in LSM-based KV stores are reads ($Get(k)$) and

updates ($Update(k,v)$). $Update(k,v)$ associates value $v$ to key $k$. Updates are absorbed in $C_m$ and possibly appended to the commit log. Hence, LSM KV stores achieve high write throughput. $Get(k)$ returns the most recent value of $k$. As illustrated in Figure 1, the read first goes to $C_m$; if $k$ is not found in $C_m$, the read continues to $L_0$, checking all the files. If $k$ is not found in $L_0$, the read goes to $L_1, \ldots$ $L_n$, until $k$ is found. Apart from $L_0$, only one file is checked for the rest of $C_{disk}$'s levels, because of the non-overlapping key ranges property.

**Flushing.** LSM KV stores have two main background operations: flushing and compaction. Flushing is the operation that writes $C_m$ to $L_0$, once $C_m$ becomes full. In case a commit log is used, the flush can also be triggered by the commit log getting full, even if there is still room in the memory component.

**Compaction.** Compaction is the background operation that merges files in disk component $L_i$ into files with overlapping key ranges in disk component $L_{i+1}$, discarding older values in the process. Leveled compaction is a popular strategy for compaction in LSM KV stores [5, 14]. When the size of $L_i$ exceeds its target size, a file $F$ in $L_i$ is picked and merged into the files from $L_{i+1}$ that have overlapping key ranges with $F$, in a way similar to a merge sort. Therefore, in the case of leveled LSM trees, each KV pair might be eventually propagated down to the component on the last level. Hence, some KV pairs could be rewritten once for every level during compaction. RocksDB and TRIAD employ leveled compaction. The techniques proposed by TRIAD could, however, easily be adapted to size-tiered [22] approaches.

## 3 Motivation

Despite I/O operations not being in the critical path of user-facing operations, flushing, logging and compaction still consume computational resources. The amount of CPU cycles spent to coordinate these operations translates into a commensurate amount of processing power that cannot be used to serve the user-generated workload. Hence, the frequency and the length of the I/O operations have a significant impact on the final performance perceived by the user.

We provide experimental evidence of this claim by measuring the extent of the performance reduction due to I/O operations. We consider two workloads that exhibit different levels of skew in the data popularity (skewed/uniform) and two read/write mixes (write dominated and balanced).

We run these workloads on RocksDB and on a version of RocksDB in which we disable background I/O operations (i.e., flushing and compaction; logging was enabled for both experiments). We pin all of the system activity

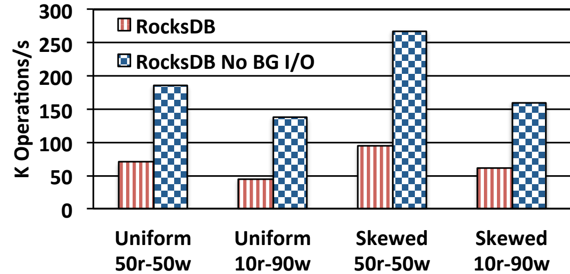



Figure 2: Background I/O impact on throughput.

(i.e., 8 worker threads and all threads created by the KV store) to 8 cores. The LSM structures of the two systems are pre-populated with an identical value for every key accessed during the experiment. This ensures that every read operation can be served, possibly by traversing the on-disk LSM tree. In the RocksdDB version with no background I/O, when a memory component is full, we discard it instead of persisting it, serving requests only from the pre-populated data store. We compare the throughput achieved by the two systems and report it in Figure 2. The plot shows that, for all workloads, background I/O represents a major performance bottleneck, yielding up to a 3x in throughput loss with respect to the ideal case.

Driven by these results, we investigate the causes that trigger frequent and intensive I/O operations. We identify three main sources of expensive I/O operations, one for each of the three main components of the LSM tree architecture, namely (1) *data-skew unawareness*, at the memory component level; (2) *premature and iterative compaction*, at the LSM tree level; (3) *duplicated writes* at the logging level.

**1. Data skew unawareness.** Many KV store workloads exhibit skewed data popularity, in which a few *hot* keys have a much higher probability of being updated than *cold* keys [16]. As we show in Section 5, some Nutanix production workloads also exhibit similar skew.

Data skew causes the commit log to grow more rapidly than $C_m$, because updates to the same keys are appended to the log but absorbed in-place by $C_m$. This triggers frequent flushes of $C_m$ before it reaches its maximum size. Not only does this increase the frequency of flushing, but because the size of the flushed $C_m$ is often smaller than the maximum, the fixed cost of opening and storing a file in $L_0$ is not amortized by the actual writing of data in it.

Data skew also has a negative impact on the extent of the compaction process. In fact, it is highly likely that a copy of a hot key is present in many levels of the LSM tree structure. This results in frequent compaction operations that easily trickle down the LSM tree structure, causing long cascading compaction phases at $L_i$ that likely result into spilling new data to $L_{i+1}$.

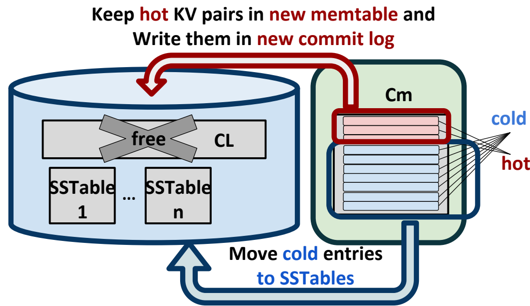



Figure 3: TRIAD-MEM: Before hot-cold key separation.

**2. Premature and iterative compaction.** Existing LSM KV systems exhibit a two-fold limitation in the compaction process. Some LSM KV stores keep only one file in $L_0$ to avoid looking up several SSTables in $L_0$ when reading [5]. As a result, every time the memory component is flushed, a compaction from $L_0$ to the underlying levels is triggered. This choice leads to frequent compactions of the LSM tree.

Other LSM schemes [4, 12, 14] keep several files in $L_0$. This approach leads to the second limitation of existing LSM KV stores. The issue lies in how LSMs compact $L_0$ to $L_1$ when several SSTables are present in $L_0$. In fact, files in $L_0$ are compacted to higher levels one at a time, resulting in several consecutive compaction operations. If two files in $L_0$ share a common key, this key is compacted twice in the underlying LSM tree. Data skew exacerbates this problem, because it increases the probability that multiple $L_0$ files contain the same set of hot keys. Clearly, the higher the load on the system, the higher is the probability of this event happening.

This phenomenon can also arise in systems that keep a single file in $L_0$. Indeed, during the compaction, the system continues serving user operations, thus potentially triggering multiple flushes of the memory component to $L_0$. As a result, when a compaction finishes, it is possible that multiple files reside in $L_0$.

**3. Duplicate writes.** When $C_m$ is flushed to $L_0$, the corresponding commit log is discarded because flushing already ensures the durability of the data. Each KV pair in the new file in $L_0$, however, corresponds to the last version of a key written in the memory component and, hence, appended to the commit log. Therefore, when flushing the memory component to disk in $L_0$, the system is actually replaying I/O that it has already performed when populating the commit log.

## 4 TRIAD

We now provide a detailed description of TRIAD's techniques. The pseudocode for the main parts of TRIAD's

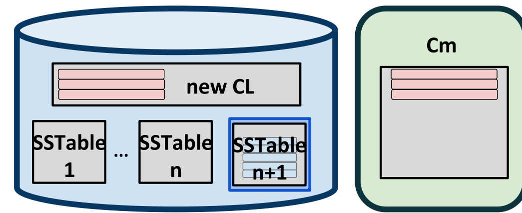



Figure 4: TRIAD-MEM: After hot-cold key separation and flush.

algorithms are shown in Algorithm 1 and Algorithm 2. The approach we use to tackle the I/O overhead is three-fold, each solution addressing one of the challenges highlighted in the previous section:

(1) **TRIAD-MEM** tackles the data skew unawareness issue at the memory component level.
(2) **TRIAD-DISK** tackles the premature and iterative compaction issue by judiciously choosing to defer and batch compaction at the disk component level.
(3) **TRIAD-LOG** tackles the duplicated writes issue, bypassing new file creation during flushes, at the commit log level.

The three techniques complement each other and target the main components of LSM KV stores. Even if they work best together, they are stand-alone techniques and generally applicable to LSM-based KV stores.

### 4.1 TRIAD-MEM

The goal of TRIAD-MEM is to leverage the skew exhibited by many workloads [16] to reduce flushing and, hence, reduce the frequency of compactions. To this end, TRIAD-MEM only flushes *cold* keys to disk, while keeping *hot* keys in memory. This avoids the numerous compactions triggered to ensure non-overlapping key ranges in the LSM disk structure.

TRIAD-MEM separates entries that are updated often (i.e., hot entries) from entries which are rarely updated (i.e., cold entries) upon flushing $C_m$ to $L_0$. The hot entries are kept in the new $C_m$ and only the cold entries are written to disk. This way, the hot entries are updated just in-memory and do not trigger a high number of compactions on disk. The hot-cold key separation during a flush to $L_0$ is shown in Figure 3 and Figure 4.

The separation between hot and cold keys is shown in the *separateKeys* function in Algorithm 2. The top-K entries of the old $C_m$ are selected, where K is a parameter of the system. Ideally, K should be high enough to accommodate all the hot keys, but low enough to avoid a high memory overhead for $C_m$. Thus, properly setting K requires some *a priori* knowledge about the workload. TRIAD, however, is designed to deliver high per-

**Algorithm 1** Update and Flush.

```
 1: function UPDATE(Key k, Val v)
 2:    Entry e =mem.getEntry(k)
 3:    CommitLog log =getCommitLog()
 4:    if (e≠ NULL) then
 5:       e.val = v;  e.updates++
 6:       CLUpdateOffset(log,&e)       ▷ Update CL name and offset in entry e
 7:    else
 8:       e = new Entry(k, v);  e.updates = 1
 9:       CLUpdateOffset(log,&e)       ▷ Update CL name and offset in entry e
10:       mem.add(e)
11:    end if
12: end function

13: function FLUSH(Memtable mem)
14:    if (mem.getSize() < FLUSH_TH) then  ▷ Do not flush if mem too small
15:       CommitLog newLog = new CommitLog()
16:       populateLog(nwLog, mem)
17:       CommitLog log = getCommitLog()
18:       setCurrentCommitLog(newLog)
19:       discardCommigLog(log)
20:       CLUpdateOffset(newLog, mem)
21:    else
22:       Memtable hotMem = new Memtable()
23:       Memtable coldMem = new Memtable()
24:       separateKeys(mem, hotMem, coldMem)
25:       setCurrentMemtable(hotMem)
26:       CommitLog log = getCommitLog()
27:       CommitLog newLog = new CommitLog()
          ▷ Write back hotMem entries to the new log
28:       populateLog(newLog, hotMem)
29:       setCurrentCommitLog(newLog)
          ▷ Update hotMem with offsets from new CL
30:       CLUpdateOffset(newLog, hotMem)
          ▷ Extract index corresponding to cold keys
31:       CLIndex index = getKeysAndOffsets(coldMem)
          ▷ Flush only index and link it to old CL
32:       flushToDisk(index, log)
33:    end if
34: end function
```

**Algorithm 2** Key Separation and Deferred Compaction.

```
 1: function SEPARATEKEYS(Memtables mem, hotMem, coldMem)
 2:    int hotKeyCount = sizeof(Memtable) *
                    PERC_HOT / sizeof(Entry)
 3:    Entry[] hotKeys = getTopKHot(mem, hotKeyCount)
 4:    hotMem.add(hotKeys)
 5:    for k in hotMem do                        ▷ Reset hotness
 6:       k.hotness = 0; k.updates = 0
 7:    end for
 8:    coldMem = mem
 9:    coldMem.remove(hotKeys)
10: end function

11: function DEFERCOMPACTION(())
12:    assert(level == 0)
13:    int totalKeys = 0
14:    HyperLogLog hllVect[]
15:    FileMetaData levelFiles[]
16:    levelFiles = getLevelFiles(0)
17:    for f in levelFiles do
18:       totalKeys+= f.hllKeysCount()
19:       hllVect.pushBack(f.hllGet())
20:    end for
21:    int estimated = hhllMergedEstimate(hllVect)
22:    double overlapRatio = 1 - (estimated / totalKeys)
23:    boolean notEnoughOverlap = overlapRatio < OVERLAP_RATIO_TH
24:    boolean notEnoughFiles = getLevelFiles(0).size()≤ MAX_FILES_L0
25:    if notEnoughOverlap ∧ notEnoughFiles then
26:       return true                            ▷ Defer compaction
27:    end if
28:    return false
29: end function
```

formance with no information about the workload. In our current implementation, K is a constant. We will show in Section 5 that thanks to its holistic multi-level approach, TRIAD is robust against settings of K that correspond to not storing all the hot keys in memory. We are also currently investigating techniques to automatically set K depending on the runtime workload, for example by means of hill climbing [43].

The entries that are preserved in the new memory component and not sent to disk are written to the commit log associated to the new memory component, as shown in Figure 3. This write-back is necessary in order to not lose information. A final optimization when separating the hot and cold keys is not flushing at all if $C_m$'s size is not larger than a certain threshold (denoted $FLUSH_TH$ in Algorithm 1). Indeed, in the case of very skewed workloads, a flush might be triggered not because $C_m$ is full, but because the commit log becomes full. To avoid having a large number of small files, we keep all entries in memory, discard the old commit log, and create a new commit log, with only the freshest values of $C_m$ entries.

We experiment with several methods for hot-cold key detection, including looking at mean and standard deviation of the update frequencies, and selection according to quantiles. Simply selecting as hot keys those keys that are updated with higher frequency than the average one is effective in all workloads.

## 4.2 TRIAD-DISK

TRIAD-DISK acts at $L_0$ of the LSM disk component. In a nutshell, TRIAD-DISK delays compaction until there is enough key overlap in the files that are being compacted. To approximate the key overlap between files, we use the HyperLogLog (HLL) probabilistic cardinality estimator [28, 30]. HLL is an algorithm that approximates the number of distinct elements in a multiset. To compute the overlap between a set of files, we define a metric we call the *overlap ratio*. Assuming we have $n$ files on $L_0$, the overlap ratio is defined as 1 - $(UniqueKeys(file_1, file_2, \dots\ file_n))$ / $sum(Keys(file_i))$, where $Keys(file_i)$ is the number of keys of the i-th SSTable and $UniqueKeys$ is the number of unique keys after merging the $n$ files. $UniqueKeys$ and $Key(file_i)$ are approximated using HLL.

Figure 5 shows an example of how the overlap ratio is used to defer compaction. In the upper part of the figure, there is only one file on $L_0$; the $L_0$ file overlaps with two files on $L_1$. Since the overlap ratio is smaller than the cutoff threshold in this case, compaction is deferred. The lower part of the figure shows the system at a time when $L_0$ contains two files. The overlap ratio is computed between *all the files* in $L_0$ and their respective overlapping files on $L_1$. The overlap ratio is higher than the threshold,

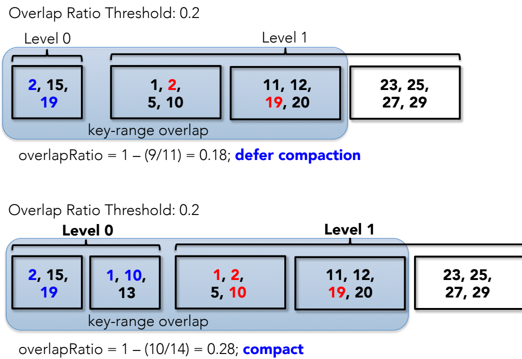



Figure 5: Overlap ratio example.

so compaction can proceed, by doing a multi-way merge between all files in $L_0$ and the overlapping files in $L_1$.

The function *deferCompaction* in Algorithm 2 shows the TRIAD-DISK pseudo-code. We associate an HLL structure to each $L_0$ file in the LSM disk component. Before each compaction in $L_0$, we calculate the overlap ratio of all files in $L_0$. If the overlap ratio is below a threshold, we defer the compaction, unless the number of files in $L_0$ exceeds the maximum allowed number. If the maximum number of files in $L_0$ is reached, we proceed with the $L_0$ to $L_1$ compaction, regardless of the key overlap.

The use of HLL is not new in the context of LSM compaction. So far, however, the way HLL is used is to detect which files have the most key overlap to be compacted (for instance in systems such as Cassandra [1]). This way, the highest number of duplicate keys is discarded during compaction. RocksDB employs a similar idea, where the estimation of the key overlap in files at $L_i$ and $L_{i+1}$ is based on the files' key ranges and sizes. Our use of HLL is different. Instead of employing HLL to decide which files to compact, we are using HLL to decide *whether* to compact $L_0$ into $L_1$ at the current moment, or defer it to a later point in time. If the $L_0$ and $L_1$ SSTables do not have enough key overlap, compaction is delayed until more $L_0$ SSTables are generated.

Current LSMs trigger the compaction of $L_0$ into $L_1$ as soon as the number of files on $L_0$ reaches a certain threshold. The larger the threshold, the more files need to be accessed in $L_0$ by read operations, which increases read latency. However, since the chance of a key being present multiple files on $L_0$ is low (otherwise, the large overlap ratio would trigger compaction), TRIAD-DISK can tolerate more files in $L_0$ without hurting read performance, as we show in Section 5.

## 4.3 TRIAD-LOG

The main insight of TRIAD-LOG is that the data that is written to memory and then persisted into $L_0$ SSTables is already written to disk, in the commit log. The general idea of TRIAD-LOG is to transform CL into a special type of $L_0$ SSTable, a *CL-SSTable*. This way, flushing from memory to $L_0$ is avoided altogether.

TRIAD-LOG enhances the role played by the commit log. As $C_m$ is being written to, the commit log plays its classic role. When flushing is triggered, instead of copying $C_m$ to disk, we convert the commit log into a CL-SSTable. As shown in Figure 6, instead of storing copies of the memory components in $L_0$, we store CL-SSTables. For readability, we only depict the TRIAD-LOG technique, and not the integration with our two other techniques.

The advantage of treating the commit logs as $L_0$ SSTables is that the I/O due to flushing from memory is avoided. However, unlike SSTables, the commit log is not sorted. The sorted structure of the classic SSTables makes it easy to merge SSTables during compaction and to retrieve information from the files. To avoid scanning the entire CL-SSTable in order to find an entry in $L_0$ files, we keep the commit log file offset of the most recent update in $C_m$, for each KV pair. Once the flush operation is triggered, only the small index associated to the offsets in the commit log is written to disk. The index is then grouped with its corresponding commit log file, thus creating the new $L_0$ CL-SSTable format.

For instance, consider a commit log with entries of size 8B, in the format (Key; Value): (1;10), (2;20), (3;30), (4;40), (3;300). Then, in $C_m$, TRIAD-LOG keeps the following entries, in the format (Key; Value; CL offset; CL name): (1; 10; 0; CL-name), (2; 20; 8; CL-name), (3; 300; 32; CL-name), and (4; 40; 24; CL-name). The CL offset is equal to 32 for Key 3, because we keep the offset of the most recent update.

TRIAD-LOG offers the greatest benefits when the workload is more uniform. For such workloads it is relatively rare that the same key appears several times in the log. The corresponding CL-SSTable therefore contains the most recent values of many distinct keys, and relatively few older values. For skewed workloads, in contrast, the log typically contains multiple updates of the same keys, and the corresponding CL-SSTable therefore stores a high number of old values that are no longer relevant.

The flow of the write operation remains unchanged by TRIAD-LOG. The writes are performed in $C_m$ and persisted in the commit log. The only difference is that apart from the value associated to the key, the commit log name and offset entries are updated as well. Similarly, the read path is largely unchanged, except for accessing the files in $L_0$. As before, the reads first look in $C_m$, then in all of the $L_0$ files, and then in one file for each of the lower levels of the disk component. Unlike before, when a file from $L_0$ is read, the index is searched

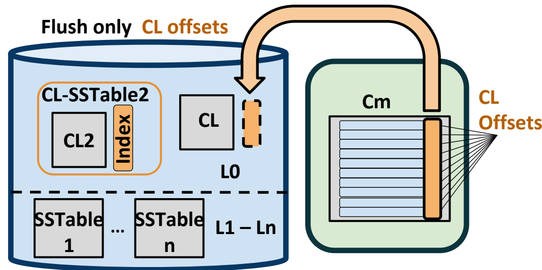



Figure 6: TRIAD-LOG operation flow.

for the key, and, if found, the CL-SSTable is accessed at the corresponding offset.

Compaction from $L_1$ to $L_n$ is unchanged, because no modifications are done to the SSTable format on these levels. Only the compaction between $L_0$ and $L_1$ is affected by our technique. A new compaction operation is needed for merging a CL-SSTable with a regular SSTable. Since the index kept on the CL-SSTable is sorted, it is still possible to proceed in a merge-sort style manner. For clarity and brevity of the presentation, we omit the pseudocode for merging SSTables.

It is straightforward to integrate TRIAD-LOG with TRIAD-DISK, since TRIAD-DISK affects only the decision to call compaction. The integration with TRIAD-MEM is done by flushing only the part of the index corresponding to the cold keys, ignoring the offsets of the hot keys. Then, during compaction, the hot keys are skipped, similarly to the duplicate updates.

**TRIAD Memory Overhead Analysis.** TRIAD reduces I/O by using additional metadata in memory. TRIAD-MEM needs an *update frequency* (4B) field for each memory component entry. For each (CL-)SSTable on $L_0$, TRIAD-DISK tracks the HLL structure (4KB per file). TRIAD-LOG adds two new fields for each memory component entry: the *commit log file ID* (4B) and the *offset* in the commit log (4B). Finally, TRIAD-LOG keeps track of the offsets index (8B per entry) for each CL-SSTable on $L_0$. While the HLLs and offset indexes could be stored on disk, this would incur a performance penalty. Since the number of files on $L_0$ is not large, the memory overhead is not significant. Generally, less than 10 files are kept on $L_0$. Hence, an upper bound on TRIAD's memory overhead is: $12B * EntriesC_m + 10 * (4KB + EntriesC_m * 8B)$. In our tests, the memory overhead is on the order of tens of MB, which is negligible with respect to the tens of GB of I/O saved.

## 5 Evaluation

We implement TRIAD as an extension of Facebook's popular RocksDB LSM-based KV store. The source code of our implementation is available at `https://github.com/epfl-labos/TRIAD`. We evaluate TRIAD with production and synthetic workloads, and we compare it against RocksDB. We show that:

(1) TRIAD achieves up to 193% higher throughput in production workloads.
(2) TRIAD effectively reduces I/O by an order of magnitude and spends, on average, 77% less time performing flushing and compaction.
(3) TRIAD's three techniques work in synergy and enable the system to achieve high throughput without a priori information about the workload (e.g., skew on data popularity or write intensity).

### 5.1 Experimental Setup

We compare TRIAD against RocksDB. Unless stated otherwise, RocksDB is configured to run with its default parameters, and we do not change the corresponding values in the TRIAD implementation. TRIAD uses an overlap threshold of 0.4 and a maximum number of 6 $L_0$ files for TRIAD-DISK. In addition, we configure TRIAD-MEM such that its definition of hot keys corresponds to the top 1 percent of keys in terms of access frequency.

To evaluate TRIAD, we use four production workloads from Nutanix (see Section 5.2). We complement our evaluation with synthetic benchmarks that allow us to control key parameters of the workload, such as skew and write intensity (see Section 5.3). The evaluation is performed on a 20 core Intel Xeon, with two 10-core 2.8 GHz processors, 256 GB of RAM, 960GB SSD Samsung 843T, running Ubuntu 14.04.

Each synthetic benchmark experiment consists of a number of threads concurrently performing operations on the KV store – searching, inserting or deleting keys. Each operation is chosen at random, according to the given workload probability distribution, and performed on a key chosen according to the given workload skew distribution. Before each experiment, the LSM tree is initialized with roughly half of the keys in the key range.

We use as evaluation metrics throughput measured in KOperations/second (KOPS), bytes written to disk, time spent in background operations (i.e., compaction and flushing), write amplification (WA), and read amplification (RA). WA and RA are established metrics for measuring I/O LSM KV store efficiency [4, 12, 32, 34, 35, 38, 48]. WA is the amount of data written to storage compared to the amount of data that the application writes. Intuitively, the lower the WA, the less work is done during compaction. We compute system-wide WA as: $WA = (Bytes_{flushed} + Bytes_{compacted})/Bytes_{flushed}$. RA is the average number of disk accesses per read.

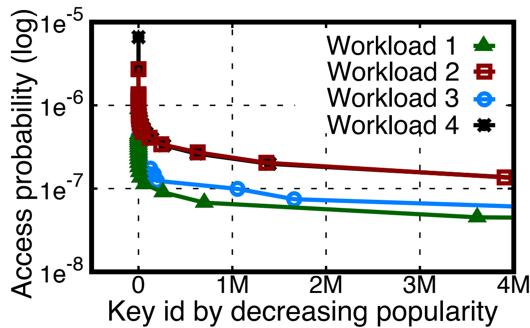



Figure 7: Production workloads key probability distributions (Logarithmic scale on y axis).

| | Wkld. 1 | Wkld. 2 | Wkld. 3 | Wkld. 4 |
|---|---|---|---|---|
| Updates | 250M | 75M | 200M | 75M |
| Keys | 40M | 9M | 30M | 8M |

Figure 8: Production workloads: number of updates and number of keys.

## 5.2 Production Workloads

The production workloads used for the evaluation of TRIAD are internal Nutanix metadata workloads. The key probability distributions of the workloads are shown in Figure 7. The data sizes and number of updates are shown in Figure 8. The production workloads have two different skew profiles: W2 and W4 have more skew in their access patterns, W1 and W3 have less skew.

The left-hand side of Figure 9A presents the throughput comparison between RocksDB and TRIAD, for the four production workloads. TRIAD outperforms RocksDB in the four workloads, with a throughput increase of up to 193%. The right-hand side of Figure 9A shows the corresponding WA for each of the workloads. TRIAD reduces WA by up to 4x.

As expected [34], in RocksDB the WA is higher for the less skewed workloads (W1 and W3) and lower for the more skewed workloads (W2 and W4). There is also a clear correlation between the throughput and the WA: throughput is lower in the workloads with higher WA.

For TRIAD WA is uniform across the four workloads, because TRIAD-MEM converts the skew of the application workload into a disk workload that is closer to uniform. Hence, the workload skew perceived by the disk component is more or less the same across the four workloads, leading, in turn, to more predictable throughput. In contrast with RocksDB, TRIAD's throughput does not exhibit high fluctuation across workloads. We explore this connection between throughput and WA further in the next section.

## 5.3 Synthetic Workloads

We define three workload skew profiles: (WS1) A highly skewed workload where 1% of the data is accessed and updated 99% of the time. This workload reflects the characteristics of Facebook workloads analyzed in [16]. (WS2) A medium skew workload, where 20% of the data is accessed and updated 80% of the time. (WS3) A uniform workload where all keys have the same popularity.

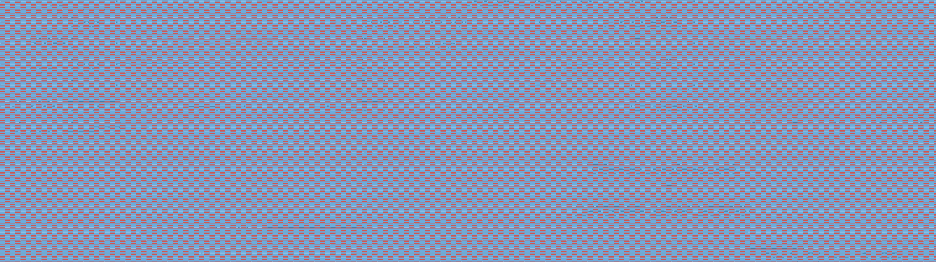
We use two different read-write ratios: one with 10% reads and 90% writes, and one with 50% reads and 50% writes. In all experiments, each key is 8B and each value is 255B. To shorten our experiments with the synthetic workloads, we use a small memory component of 4MB and a dataset of 1M keys, so that compactions happen at shorter time intervals.

Figure 9B shows the throughput comparison between TRIAD and RocksDB for the three workload skews and two read-write ratios. Figure 9C shows the corresponding WA. TRIAD performs up to 2.5x better than RocksDB for the skewed workloads and up to 2.2x better for the uniform workloads.

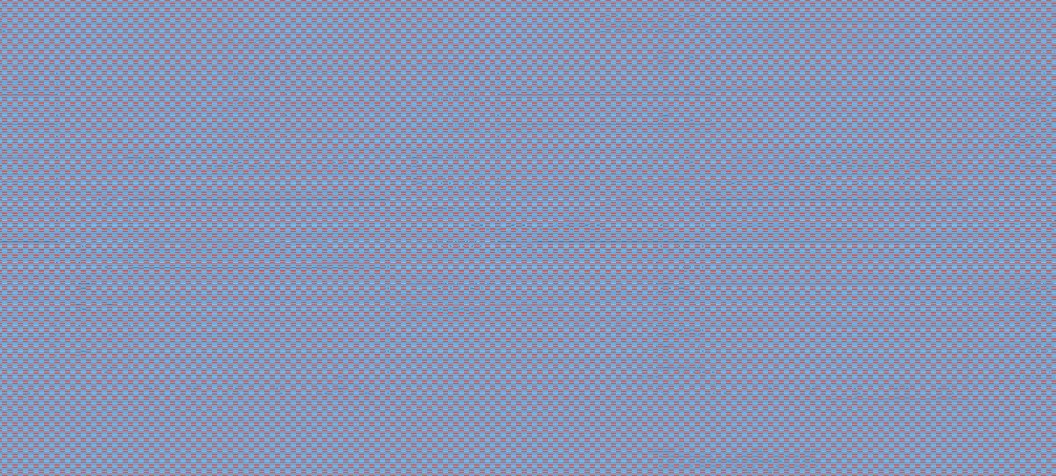
For WS1 all the hot data fits in memory, allowing TRIAD to achieve a throughput increase of 50% for both the write-intensive and the balanced workloads. For WS2, TRIAD-MEM cannot accommodate all the hot keys. Nevertheless, TRIAD still achieves a throughput gain of 51% in the write-intensive workload and 25% in the balanced workload, because TRIAD-DISK and TRIAD-LOG act as a safety net against possible undersizing of the data structure tracking hot keys (Section 4.1), due to lack of detailed knowledge of the workload characteristics. This result showcases the robustness of TRIAD: It consistently delivers high performance, despite not having any prior knowledge of the incoming workload.

WA is decreased by up to 4x in the moderately skewed and uniform workloads. For the highly skewed workload, however, the WA does not change, despite the gain in throughput. This happens because the 1% of the data that is updated 99% of the time fits entirely in memory. As a consequence, $C_m$ is only rarely flushed (as we explain in Section 4), because it takes longer for enough cold entries to be present in $C_m$ to trigger a flush. Therefore, even if the total number of bytes is decreased by an order of magnitude, as Figure 9D shows on the left-hand side, the proportion between the compacted bytes and flushed bytes is similar to RocksDB.

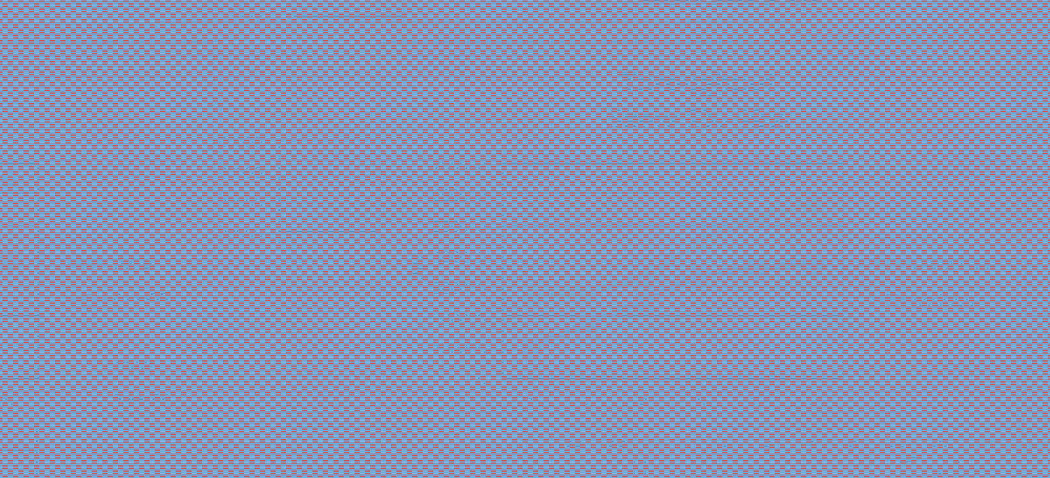
Finally, the right-hand side of Figure 9D shows the time spent in compaction. For the highly skewed workload, the time spent in compaction in TRIAD is an order of magnitude lower than RocksDB, for the same reason as explained above. For the moderately skewed and



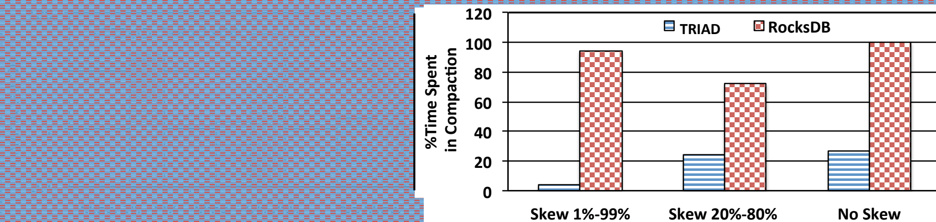
A. Production workload throughput and corresponding WA. 8 threads.



B. Throughput comparison for different workloads and skews (higher is better).



C. Write amplification comparison for different workloads and skews (lower is better).




D. Left: Compacted GB (Logarithmic scale). Right: Percentage of time spent in compaction. 8 threads, 10%reads – 90%writes.

Figure 9: TRIAD in production and synthetic workloads.

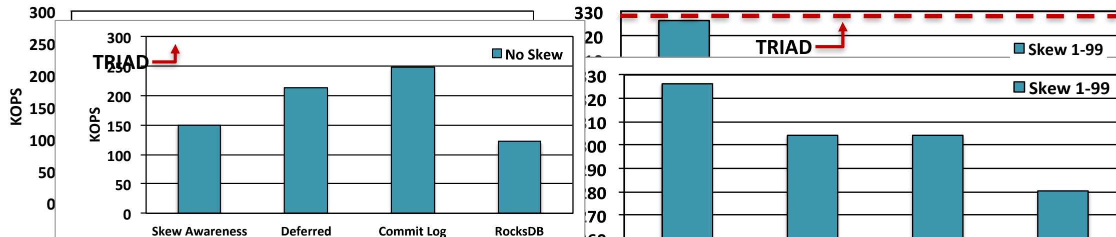



Figure 10: Throughput breakdown for uniform and skewed workloads. 16 threads.

uniform workloads, the time spent in compaction is decreased by 48% and 72%, respectively.

## 5.4 Breakdown of TRIAD's Benefits

We discuss the contribution of each of TRIAD's techniques, for different types of workloads, reporting the throughput achieved by versions of TRIAD where we only implement one out of the three techniques.

Figure 10 shows the throughput breakdown for each of the techniques, for synthetic workloads WS3 (left-hand side) and WS1 (right-hand side), with a 10%–90% read–write ratio. While all three techniques outperform RocksDB individually, TRIAD-MEM brings more benefits than TRIAD-DISK and TRIAD-LOG for the highly skewed workload, and vice-versa for the uniform workload. Indeed, TRIAD-MEM alone obtains 97% of the throughput that TRIAD achieves for the skewed workload. For the uniform workload, TRIAD-DISK and TRIAD-LOG obtain 85% and 97%, respectively.

A similar trend can be noticed in the WA breakdown in Figure 11. TRIAD-MEM performs best for WS1, but does not decrease WA as the workload is closer to uniform, having close to no effect compared to RocksDB for the workload with no skew (right-most column). TRIAD-DISK and TRIAD-LOG are complementary to TRIAD-MEM, decreasing WA by up to 60% and 40%, respectively, for the uniform workload.

The lower-right plot in Figure 11 shows the RA breakdown for a uniform workload, with 10% reads. As expected, TRIAD-MEM lowers RA, because more requests can be served from memory. TRIAD-DISK, however, increases RA compared to the baseline, as it keeps more files in $L_0$, and all these files may have to be accessed on a read. TRIAD-LOG does not have an impact on read amplification. Overall, TRIAD has a low overhead over the baseline, increasing RA by at most 5%.

The breakdown shows tat the three techniques are complementary: no one alone gives 100% of the benefits across all workload types. Their combination allows TRIAD to achieve high performance for any workload, automatically adapting to its characteristics without a priori knowledge.

# 6 Related Work

Our work is related to previous designs of LSM-based KV stores and to various systems that employ optimization techniques similar to the ones integrated in TRIAD.

**Related LSM-based KV stores.** LevelDB [4] is one of the earliest LSM-based KV stores and employs level-style compaction. Its single-threaded compaction, along with the use of a global lock for synchronization at the memory component level are two of its main bottlenecks. RocksDB [12] introduces multi-threaded compaction and tackles other concurrency issues. LevelDB and RocksDB expose several tuning knobs, such as the number and the sizing of levels, and policies for compaction [13, 14, 27]. Recent studies, simulations and analytical models show that the efficiency of LSM-based KV stores is highly dependent on their proper setting, as well as workload parameters [34, 26] and requirements like memory budget [24]. In contrast, TRIAD presents techniques that cover a large spectrum of workloads. Thanks to its holistic approach, TRIAD is able to deliver high performance without relying on *a priori* information about the workload.

bLSM [44] proposes carefully scheduling compaction to bound write latency. VT-tree [45] uses an extra layer of indirection to avoid sorting any previously sorted KV pairs during compaction. HyperlevelDB [9] also addresses the write and compaction issues in LevelDB, through improved parallelism and an alternative compaction algorithm [7, 8]. HyperLevelDB's compaction chooses a set of SSTables which result in the lowest WA between two levels. TRIAD takes a different approach to prevent the occurrence of high WA, by using HLL to decide whether to compact or not at the first level of the disk component.

LSM-trie [47] proposes a compaction scheme based on the use of cryptographic functions. This scheme gives up the sorted order of the entries in the LSM tree to favor compaction efficiency over performance in range queries. TRIAD instead preserves the sorted order of the keys, facilitating support for efficient range queries.

WiscKey [37] separates keys from values and only stores keys in a sorted LSM tree, allowing it to reduce

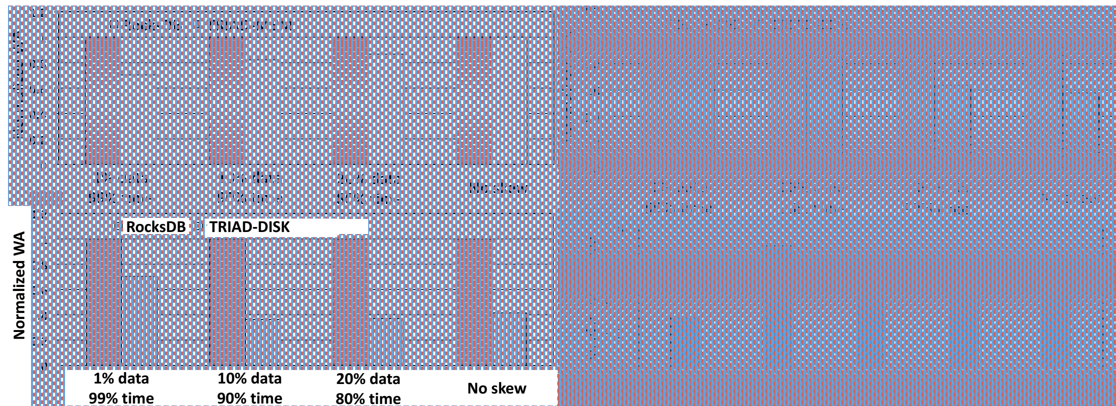



Figure 11: Write Amplification and Read Amplification Breakdown. 8 threads.

data movement, and consequently reduce write amplification. The techniques proposed in TRIAD are orthogonal to this approach, and can be leveraged in synergy to it to further enhance I/O efficiency.

Cassandra [1], HBase [11] and BigTable [22] are distributed LSM KV stores, employing size-tiered compaction. In addition, Cassandra also supports the leveled compaction strategy, based on LevelDB's compaction scheme [5, 6]. For both compaction strategies, Cassandra introduced HyperLogLog (HLL) to estimate the overlap between SSTables, before starting the merge [10]. TRIAD also makes use of HLL in its deferred compaction scheme. However, instead of using HLL to determine which files to compact, the overlap between files computed with HLL is used only at $L_0$, to determine whether we should compact or wait.

Ahmad and Kemme [15] also target a distributed KV store and propose to offload the compaction phase to dedicated servers. In contrast, the techniques proposed in TRIAD are applied within each single KV store instance and do not need dedicated resources to be implemented.

Tucana [42], LOCS [46] and FloDB [17] act on other aspects of the KV store design to improve performance. Tucana uses an internal structure similar to a $B-\varepsilon$ tree [20] and uses copy-on-write instead of write-ahead logging. LOCS exploits the knowledge of the underlying SSD multi-channel architecture to improve performance, e.g., by load balancing I/O. FloDB inserts an additional fast in-memory buffer on top of the existing in-memory component of the LSM tree to achieve better scalability. TRIAD can integrate some of these features to further improve its performance.

**Systems with similar optimization techniques.** The hot-cold separation technique is employed in SSDs to improve the efficiency of the garbage collection needed by the Flash Translation Layer [23, 33]. In TRIAD, instead, it is used at the KV store level to reduce the amount of data written to disk.

Delaying and batching the execution of updates is used in $B-\varepsilon$ trees and in systems, e.g., file-systems [31], which use $B-\varepsilon$ trees as main building block. This technique is employed to amortize the cost of updates [19] and to reduce the cases in which the effect of an update is immediately undone by a following update [49]. By contrast, TRIAD defers the compaction of the $L_0$ level of the LSM tree and batches the compaction of multiple keys to increase the efficiency of the compaction process.

## 7 Conclusion

TRIAD is a new LSM KV store aiming to reduce background I/O operations to disk. TRIAD embraces a holistic approach that operates at different levels of the LSM KV store architecture. TRIAD increases I/O efficiency by incorporating data skew awareness, by improving the compaction process of the LSM tree data structure and by performing more efficient logging.

We compared TRIAD with Facebook's RocksDB and we showed, using production and synthetic workloads, that TRIAD achieves up to an order of magnitude lower I/O overhead and up to 193% higher throughput.

**Acknowledgements.** We would like to thank our shepherd, Liuba Shrira, the anonymous reviewers and Dmitri Bronnikov, Rishi Bhardwaj, Ashvin Goel, and Amitabha Roy for their feedback that helped us to improve the paper. This work was supported in part by the Swiss National Science Foundation through grant No. 166306 and by a gift from Nutanix, Inc. Part of the work has been done while Oana Balmau was an intern at Nutanix.

## References

[1] Apache Cassandra. `http://cassandra.apache.org`.

[2] Memcached, an open source, high-performance, distributed memory object caching system. `https://memcached.org/`.

[3] Redis, an open source, in-memory data structure store. https://redis.io/.

[4] LevelDB, a fast and lightweight key/value database library by Google, 2005. https://github.com/google/leveldb.

[5] Leveled Compaction in Apache Cassandra, 2011. http://www.datastax.com/dev/blog/leveled-compaction-in-apache-cassandra.

[6] When to Use Leveled Compaction, 2012. http://www.datastax.com/dev/blog/when-to-use-leveled-compaction.

[7] Hyperleveldb performance benchmarks., 2013. http://hyperdex.org/performance/leveldb/.

[8] Inside hyperleveldb., 2013. http://hackingdistributed.com/2013/06/17/hyperleveldb/.

[9] HyperLevelDB, a fork of LevelDB intended to meet the needs of HyperDex while remaining compatible with LevelDB., 2014. https://github.com/rescrv/HyperLevelDB.

[10] Improving compaction in Cassandra with cardinality estimation, 2014. http://www.datastax.com/dev/blog/improving-compaction-in-cassandra-with-cardinality-estimation.

[11] Apache HBase, a distributed, scalable, big data store, 2016. http://hbase.apache.org/.

[12] RocksDB, a persistent key-value store for fast storage environments, 2016. http://rocksdb.org/.

[13] RocksDB options of compaction priority, 2016. http://rocksdb.org/blog/2016/01/29/compaction_pri.html.

[14] RocksDB tuning guide, 2016. https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide.

[15] Ahmad, M. Y., and Kemme, B. Compaction management in distributed key-value datastores. *Proc. VLDB Endow. 8*, 8 (Apr. 2015), 850–861.

[16] Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., and Paleczny, M. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review* (2012), vol. 40.

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

[18] Băsescu, C., Cachin, C., Eyal, I., Haas, R., Sorniotti, A., Vukolić, M., and Zachevsky, I. Robust data sharing with key-value stores. DSN 2012.

[19] Bender, M. A., Farach-Colton, M., Jannen, W., Johnson, R., Kuszmaul, C., B., Porter, D. E., Yuan, J., and Zhan, Y. An introduction to b$\varepsilon$-trees and write-optimization. *login 40*, 5 (Oct. 2015).

[20] Brodal, G. S., and Fagerberg, R. Lower bounds for external memory dictionaries. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (Philadelphia, PA, USA, 2003), SODA '03, Society for Industrial and Applied Mathematics, pp. 546–554.

[21] Cano, I., Aiyar, S., and Krishnamurthy, A. Characterizing private clouds: A large-scale empirical analysis of enterprise clusters. SOCC 2016.

[22] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems 26*, 2 (2008).

[23] Chang, L.-P. On efficient wear leveling for large-scale flash-memory storage systems. In *Proceedings of the 2007 ACM Symposium on Applied Computing* (New York, NY, USA, 2007), SAC '07, ACM, pp. 1126–1130.

[24] Dayan, N., Athanassoulis, M., and Idreos, S. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data* (New York, NY, USA, 2017), SIGMOD '17, ACM, pp. 79–94.

[25] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: amazon's highly available key-value store. SOSP 2007.

[26] Dong, S., Callaghan, M., Galanis, L., Borthakur, D., Savor, T., and Strum, M. Optimizing space amplification in rocksdb. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings* (2017).

[27] Dong, S., Callaghan, M., Galanis, L., Borthakur, D., Savor, T., and Stumm, M. Optimizing space amplification in rocksdb.

[28] Fusy, É., Olivier, G., and Meunier, F. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. AofA 2007.

[29] Golan-Gueta, G., Bortnikov, E., Hillel, E., and Keidar, I. Scaling concurrent log-structured data stores. Eurosys 2015.

[30] Heule, S., Nunkesser, M., and Hall, A. Hyperloglog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm. ICDT 2013.

[31] Jannen, W., Yuan, J., Zhan, Y., Akshintala, A., Esmet, J., Jiao, Y., Mittal, A., Pandey, P., Reddy, P., Walsh, L., Bender, M. A., Farach-Colton, M., Johnson, R., Kuszmaul, B. C., and Porter, D. E. Betrfs: Write-optimization in a kernel file system. *Trans. Storage 11*, 4 (Nov. 2015), 18:1–18:29.

[32] Kuszmaul, B. A comparison of fractal trees to log-structured merge (lsm) trees. *White Paper* (2014).

[33] Lee, S., Liu, M., Jun, S., Xu, S., Kim, J., and Arvind, A. Application-managed flash. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies* (Berkeley, CA, USA, 2016), FAST'16, USENIX Association, pp. 339–353.

[34] Lim, H., Andersen, D. G., and Kaminsky, M. Towards accurate and fast evaluation of multi-stage log-structured designs. FAST 2016.

[35] Lim, H., Fan, B., Andersen, D. G., and Kaminsky, M. Silt: A memory-efficient, high-performance key-value store. SOSP 2011.

[36] Lim, H., Han, D., Andersen, D. G., and Kaminsky, M. MICA: A holistic approach to fast in-memory key-value storage. *management 15*, 32 (2014).

[37] Lu, L., Pillai, T. S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Wisckey: Separating keys from values in ssd-conscious storage. FAST 2016.

[38] Marmol, L., Sundararaman, S., Talagala, N., and Rangaswami, R. Nvmkv: A scalable, lightweight, ftl-aware key-value store. USENIX ATC 2015.

[39] Marmol, L., Sundararaman, S., Talagala, N., Rangaswami, R., Devendrappa, S., Ramsundar, B., and Ganesan, S. Nvmkv: A scalable and lightweight flash aware key-value store. HotStorage 2014.

[40] O'Neil, P., Cheng, E., Gawlick, D., and O'Neil, E. The log-structured merge-tree (LSM-tree). *Acta Informatica 33*, 4 (1996).

[41] Ousterhout, J., and Douglis, F. Beating the I/O bottleneck: A case for log-structured file systems. *ACM SIGOPS Operating Systems Review 23*, 1 (1989).

[42] PAPAGIANNIS, A., SALOUSTROS, G., GONZÁLEZ-FÉREZ, P., AND BILAS, A. Tucana: Design and implementation of a fast and efficient scale-up key-value store. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference* (Berkeley, CA, USA, 2016), USENIX ATC '16, USENIX Association, pp. 537–550.

[43] RUSSELL, S., AND NORVIG, P. Artificial intelligence: a modern approach (2nd edition).

[44] SEARS, R., AND RAMAKRISHNAN, R. bLSM: a general purpose log structured merge tree. SIGMOD/PODS 2012, ACM.

[45] SHETTY, P., SPILLANE, R., MALPANI, R., ANDREWS, B., SEYSTER, J., AND ZADOK, E. Building workload-independent storage with vt-trees. FAST 2013.

[46] WANG, P., SUN, G., JIANG, S., OUYANG, J., LIN, S., ZHANG, C., AND CONG, J. An efficient design and implementation of lsm-tree based key-value store on open-channel ssd. In *Proceedings of the Ninth European Conference on Computer Systems* (New York, NY, USA, 2014), EuroSys '14, ACM, pp. 16:1–16:14.

[47] WU, X., XU, Y., SHAO, Z., AND JIANG, S. LSM-trie: An LSM-tree-based ultra-large key-value store for small data items. USENIX ATC 2015.

[48] YANG, J., PLASSON, N., GILLIS, G., TALAGALA, N., AND SUNDARARAMAN, S. Don't stack your log on my log. INFLOW 2014.

[49] YUAN, J., ZHAN, Y., JANNEN, W., PANDEY, P., AKSHINTALA, A., CHANDNANI, K., DEO, P., KASHEFF, Z., WALSH, L., BENDER, M. A., FARACH-COLTON, M., JOHNSON, R., KUSZMAUL, B. C., AND PORTER, D. E. Optimizing every operation in a write-optimized file system. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies* (Berkeley, CA, USA, 2016), FAST'16, USENIX Association, pp. 1–14.