YoDB: Rethinking Design Principles for Key-Value Storage

Abstract

Many applications of key-value (KV-)store technologies are characterized by high *spatial locality* of access, e.g., data items with identical composite-key prefixes are created or scanned together. This prevalent access pattern is underused by the ubiquitous LSM-tree design employed by KV-stores today.

We present YoDB, a persistent KV-store designed for spatially-local workloads, in particular through better use of RAM that reduces disk I/O. It outperforms RocksDB – an industry-leading LSM database – in the majority of standard YCSB benchmarks, and more notably on workloads with spatial locality. For example, YoDB exceeds RocksDB by 1.25x to 3x in range scan throughput in these scenarios. It further reduces write amplification by up to 2.6x, thereby contributing to slower device wear. YoDB provides strong (atomic) guarantees for random updates, random lookups, and scans. Finally, it provides consistent crash recovery semantics, with near-instant recovery time.

1 Introduction

Key-value stores (KV-stores) are widely used nowadays by a broad range of applications, and are projected to continue to increase in popularity in years to come; market research identifies them as the "driving factors" of the NoSQL market, which is expected to garner $4.2B by 2020 [3].

KV-stores provide a simple programming model. The data is an ordered collection of key-values pairs. The application can perform random writes, random reads, and range queries. A common design pattern is the use of *composite* keys that represent an agglomerate of attributes, where the attribute that is most important for range query performance is the key prefix. For instance, in messaging and email applications, keys are typically a concatenation of user id with additional fields such as thread id, time, and post id. Example queries in Facebook Messenger retrieve, e.g., the last 100 messages for a given user [20]. In social networks such as Facebook, associations representing graph edges are indexed by a key consisting of a pair of object ids and an association type [17]. An example queries retrieves all friendship relations for a given user id. Mobile analytics services such as Flurry Analtyics [1] aggregate statistics on data identified by composite keys consisting of attributes such as app id, device, time, location, event id, etc. The query API [13] is app id-centric, hence this dimension comes first in the key.

Composite keys induce *spatial locality* in workloads with high temporal locality, as popular entities (for example, users) result in popular *key ranges*. Spatial locality may also arise with simple (non-composite) keys, for example in reverse URLs, which are often used as keys for web search indexing [25]. While the prevalence of temporal locality (i.e., frequent access to popular entities) in real-world workloads is widely-recognized, and indeed standard benchmarks (e.g., YCSB [26]) feature skewed key-access distributions such as Zipf, these benchmarks fail to capture the spatial aspect of locality. This, in turn, leads to storage systems being optimized for a skewed distribution on individual keys with no spatial locality. In this paper, we make spatial locality a first class consideration, which leads us to rethink the design principles underlying today's popular KV-stores.

The de facto standard approach to building KV-stores today is *LSM* – log-structured merge trees [36]. The LSM approach optimizes write performance by absorbing random writes in memory and periodically flushing them as sequential files to disk. While sequential disk access dramatically improves I/O throughput, it is important to notice that the LSM design initially groups writes into files *temporally*, and not by key-range. A background *compaction* process later merge-sorts any number of files, grouping data by keys. This approach is not ideal for workloads with high spatial locality for two reasons. First, a popular key range will be fragmented across many files during long periods (between compactions). Second, the compaction process is costly both in terms of performance (as it consumes high disk bandwidth) and in terms of *write amplification*, namely the number of physical writes associated with a single application write. The latter is significant particularly in SSD as it increases disk wear. The temporal grouping of data means that compaction is indiscriminate with respect to key popularity: Since new (lower level) files are always merged with old (higher level) ones, a "cold" key range that has not been accessed since the beginning of time continues to be repeatedly re-located by compactions.

Because LSM's in-memory component consists only of recently-written keys, it does not contain keys that are frequently read without being modified. This causes *read amplification*, where a read has to search for the requested key in multiple locations. In order to optimize the read-path, LSMs use a cache of popular file blocks and Bloom filters that reduce unnecessary file access. But a key range is typically dispersed over multiple cache blocks and Bloom filters. This memory organization does not leverage spatial locality, resulting in sub-optimal use of memory resources in case such locality is present. Furthermore, it does not naturally lend itself to range scans, which are common with composite keys.

Finally, we note that LSM's temporal file organization optimizes disk I/O but induces a penalty on in-memory operation. For example, all keys – including popular ones – are flushed to disk periodically, even though persistence is assured via a separate *write-ahead-log (WAL)*. This increases write amplification and also makes the flushed keys unavailable for fast read from memory. This is particularly wasteful if the system incorporates sufficient DRAM to hold almost the entire working set. The drop in DRAM prices (more than 6.6x since 2010 [12]) and significant performance benefit DRAM offers make this scenario increasingly common.

We present YoDB, a persistent KV-store whose design diverges from the ubiquitous LSM approach. Like LSMs, we optimize I/O by absorbing updates in memory and performing bulk writes to disk. And yet unlike LSMs, we partition data according to key ranges and not temporally. Data is organized (both on disk and in memory) in large *chunks* holding contiguous key ranges. Popular chunks are cached in RAM for the benefit of both the write-path and the read-path. Chunks reduce the fragmentation of key ranges, resulting in (1) better read and write performance for workloads with spatial locality, and (2) faster range scans. Moreover, since chunks are compacted in memory, writes are flushed to disk less frequently than LSM writes (relying on a per-chunk WAL for persistence) yielding (3) reduced write amplification, and (4) better performance with memory-resident working sets.

All YoDB APIs (put, get, and scan) provide strong (atomic) consistency guarantees on multiprocessor CPU hardware. The system employs fast concurrent data structures for in-memory processing, to scale with the number of cores. Finally, YoDB provides near-zero failure recovery time since it does not need to replay WAL on recovery.

We have implemented YoDB in C++. We compare it to the recent release of RocksDB [9], an industry-leading KV-store based on LSM design. Our experiments, based on the popular YCSB benchmark suite [26], show that YoDB significantly outperforms RocksDB, especially when spatial locality is high. For example, with composite keys and a memory-resident working set, YoDB accelerates scans by up to 3x, puts by up to 1.6x, and gets by up to 2.2x. With larger working sets than the available RAM, YoDB still exceeds the RocksDB throughput by $25\% - 100\%$ on gets and scans, and 30% on puts.

We do not claim, however, that YoDB is better than a mature LSM-tree implementation under all circumstances – there are scenarios in which RocksDB performs better (e.g., under low locality). The primary goal of our work is to draw attention to the importance of spatial locality in today's workloads, and to propose a design alternative to LSM that is better suited for such locality. As with all new approaches, there is ample room for future improvements.

This paper proceeds as follows: We present our design principles in §2 and the YoDB algorithm in §3. We then discuss implementation details in §4 and evaluate YoDB in §5. Finally, §6 surveys related work and §7 concludes.

2 Design Principles

YoDB is a persistent key-value store that provides, similarly to other KV-stores [7, 8, 9], strong consistency guarantees:

Atomic API – put, get, and *range scan* (or scan) operations. Scans are atomic in the sense that all key-value pairs returned by a single scan belong to a consistent snapshot reflecting the state of the data store at a unique point in time.

Consistent recovery. Following a crash, the system recovers to a well defined execution point some time before the crash. The exact recovery point depends on the put persistency model. *Asynchronous* persistence, where puts are buffered and persisted to disk in the background, allows applications to trade durability for speed. Data consistency is preserved following recovery, in the sense that if some put is lost, then all ensuing (and thus possibly dependent) puts are lost as well.

Our key design goals are the following:

1. *Focus on spatial locality and range scans.* Multiple NoSQL applications embed multi-dimensional data in a single-dimension composite key. This design provides high spatial locality on the primary dimension (key prefix). We strive to express this locality in physical data organization in order to exploit it efficiently for scans by the primary dimension.
2. *High performance with memory-resident working sets.* To sustain high speed, key-value stores nowadays leverage increasing DRAM sizes where they can hold most of the active data set. We strive for maximum performance in this "hyper-local" case.

3. *Low write amplification.* We seek to minimize disk writes in order to boost performance and reduce disk wear, especially for SSD devices.

4. *Fast recovery.* Because crashes are inevitable, the mean-time-to-recover should be kept very short.

Given the aforementioned requirements, we make the following design choices in YoDB:

1. *Chunk-based organization.* We organize data, both on-disk and in-memory, in large chunks pertaining to key ranges. Each chunk has a file representation called *funk* (file chunk), and may be cached in a memory data structure called *munk* (memory chunk). This organization exploits spatial locality and is friendly to range scans.

   We use a number of techniques to optimize in-memory access, including partially sorting keys in each chunk and indexing munks. To expedite access to keys whose chunks are only on-disk (i.e., have no munks), individual popular keys are cached in a *row cache*, and *Bloom filters* are used to limit excessive access to disk.

2. *Infrequent disk compactions.* As long as a chunk is cached (has a munk), its funk's organization does not have to be optimized since queries do not access it. Therefore, YoDB infrequently performs reorganization (compaction) on such funks. Conversely, when a funk holds cold data, its organization hardly deteriorates, and therefore compaction is not necessary. Note that this is unlike LSM-trees, where all disk components are compacted, regardless of which keys reside in memory and whether keys are hot or cold.

3. *Multi-versioning for atomic scans.* YoDB employs multi-versioning along with copy-on-write to keep data versions required by atomic scans. In other words, if a put attempts to overwrite a key required by an active scan, then a new version is created alongside the existing one, whereas versions that are not needed by any scan are not retained. Thus, version management incurs a low overhead (as it occurs only on scans).

4. *In-funk WALs.* YoDB logs writes within funks and avoids duplicating the updates in a separate WAL. This reduces write amplification and expedites recovery.

3 YoDB's Design

§3.1 discusses YoDB's data organization. We discuss atomic scans in §3.2, and describe YoDB's normal (maintenance-free) operation flow in §3.3. The data structure's maintenance is discussed in §3.4. Finally, §3.5 discusses flushing data to disk and failure recovery.

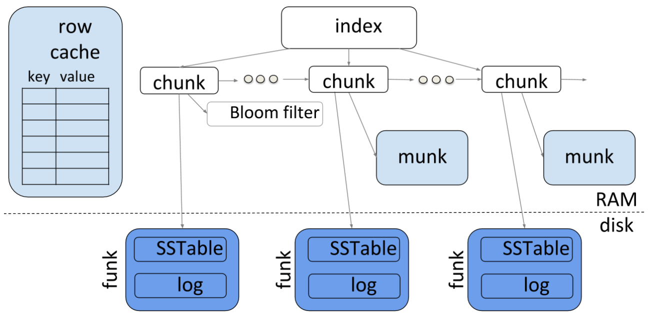


Figure 1: YoDB data layout.

3.1 Data organization

Chunks, funks, and munks. YoDB's data layout is depicted in Figure 1. Data resides in fixed-size chunks, each holding a contiguous key range. This improves the efficiency of both disk access and memory access, in particular, for range scans. At run-time, a list of all chunks' metadata is kept in RAM, where each chunk's data (consisting of keys in the corresponding range and values associated with them) is kept on disk (for persistence), and possibly in memory (for fast access).

On disk, each chunk is associated with a funk, consisting of two files: a compacted and sorted key-value map *SSTable* (Sorted String Table [24]) and a write *log*. When a funk is created, the SSTable holds all the chunk's keys with corresponding values, and the log is empty. New key-value pairs are subsequently appended to the unsorted log. If a key is over-written, it remains in the SSTable associated with the old value, and is included in the log with the new one. That is, the log is more up-to-date.

This structure allows us to benefit from sorted searches on the SSTable, and at the same time allows for updating chunks without re-writing existing data, thus minimizing write amplification. As a funk's log grows, however, searching becomes inefficient and the funk is no longer compact, i.e., it may contain redundant (over-written) values. Therefore, once the log exceeds a certain threshold, we reorganize the funk via a process we call *rebalance*, as explained below.

A subset of the chunks is also cached in memory to allow fast access, where each cached chunk is associated with a munk data structure. Munks are volatile and can be removed and recreated from funks at any time.

At run-time, YoDB holds in memory a list of chunk metadata objects as well as a *chunk index* – a sorted map from keys to chunks. Since metadata objects do not hold actual keys and values, they are significantly smaller than munks and funks (typically, less than 1 KB instead of tens of MBs). The detailes of the chunk's metadata structure are deferred to the the supplemental material.

A munk holds key-value pairs in an array-based linked list. When a munk is created, it is sorted by key, so each cell's successor in the linked list is the ensuing cell in the array. As new keys are added, they create bypasses in the linked list, and consecutive keys in the list are no longer necessarily adjacent in the array. Nevertheless, as long as a sizeable prefix of the array is sorted, bypasses are short in expectation. Keys can thus be searched efficiently via binary search on the sorted prefix and a short traversal of one "bypass".

As key-value pairs are added, overwritten, and removed munks and funks need to undergo reorganization. This includes (1) *compaction* to deallocate removed and overwritten data, (2) *sorting* keys to make searches more efficient, and (3) *splitting* overflowing chunks. All reorganizations are performed by YoDB's *rebalance* operation. If the chunk has a munk, then rebalance compacts and sorts the munk in-memory by creating a new (compacted and sorted) munk instead of the existing one. Funks of munk-less chunks are also compacted by replacing them with new funks, albeit less frequently. Splits create new chunks as well as new funks (and possibly munks) associated with them.

Expediting reads. As long as a chunk is memory-resident, the munk data structure serves both the read-path and the write-path for keys in this chunk's range. In this case, the chunk metadata is quickly located using the index, and its munk's sorted prefix allows for fast binary search. Thus, YoDB is particularly fast when almost the entire working set is memory-resident. We take two measures to mitigate the performance penalty of accessing keys in munk-less chunks.

First, we keep a *row cache* holding popular key-value pairs from munk-less chunks. Note that unlike munks, which cache key ranges, the row cache holds individual keys, and is thus more effective in dealing with point queries (gets as opposed to scans) with no spatial locality. Thus, popular key ranges can be scanned quickly from munks, while isolated popular keys can be quickly found using the row cache.

For working sets that are larger than the available DRAM, these two caches might not suffice, and so a certain portion of reads are served from disk. Here, the slowest step is the sequential search of the log. To reduce log searches to a minimum, each munk-less chunk holds a *Bloom filter* for the corresponding funk's log. Using Bloom filters, we eliminate most of the redundant log searches. To reduce the log search time in case it does happen, we further partition the Bloom filter into a handful of filters, each summarizing the content of part of the log; this allows us to know not only whether or not to search the log, but also which part of it to search.

Thread synchronization. The replacement of a chunk (due to a split) or rebalance of a funk or munk must be executed atomically, and moreover, must be synchronized with concurrent puts. This is controlled by the chunk's *rebalanceLock*, which is held for short time periods during chunk, funk, and munk replacements. It is a shared/exclusive lock, acquired in shared mode by puts and in exclusive mode by rebalance. Gets and scans do not acquire the lock.

To minimize I/O, we allow at most one thread to rebalance a funk at a given time; this is controlled by the funkChangeLock. This lock is held throughout the creation of the new funk. It is acquired using a try_lock call, and threads that fail to acquire it do not retry, but instead wait for the winning thread to complete the funk's creation.

In addition to the chunk list and chunk index, YoDB keeps a *global version (GV)* for supporting atomic scans (described in the next section) and tracks active threads' activities in the *Pending Operations (PO)* array, which has one entry per active thread. The PO is used to synchronize puts with scans, as well as for garbage collection purposes (old versions not needed by any active scan can be reclaimed). We note that using a single pending array to synchronize all operations can cause contention, which we mitigate in our implementation by tracking the pending puts in per-chunk arrays as done in [19] (see the supplemental material for details).

3.2 Multi-versioning for atomic scans

We support atomic scans via multi-versioning using the system-wide global version, GV. A scan operation creates a *snapshot* associated with GV's current value by incrementing GV, which signals to ensuing put operations that they must not overwrite values associated with smaller versions than the new GV value. This resembles a *copy-on-write* approach, which virtually creates a snapshot by indicating that data pertaining to the snapshot should not be modified in place.

To allow garbage collection of old versions, YoDB tracks snapshot times of active scans in the pending operations array, PO. The compaction process that runs as part of rebalance removes old versions that are no longer required for any scan listed in PO. Specifically, for each key, it removes all but the last version that is smaller than the minimal scan entry in PO and also smaller than the value of GV when the rebalance begins.

A put obtains a version number from GV without incrementing it. Thus, multiple puts may write values with the same version, each over-writing the previous one.

If a put obtains its version before a scan increments GV, then the new value must be included in the scan's snapshot. However, because the put's access to the GV and the insertion of the new value to the chunk do not occur atomically, a subtle race may arise. Consider a put operation that obtains version 7 from GV and then stalls before inserting the value to the chunk, while a scan obtains version 7 and increments GV to 8. The scan then proceeds to read the appropriate chunk and does not find the new value although it should be included in its snapshot.

To remedy this, we have puts announce (in PO) the key they intend to change when beginning their operations, and have scans wait for relevant pending puts to complete. That is, a put operation first registers itself in PO with the key it intends to put. It thens reads GV and sets the version field in its PO entry to the read version. After completing the actual put (in the appropriate funk and munk), it unregisters itself in PO (i.e., indicates that the put is complete).

A scan, in turn, waits for the completion of all pending puts that might affect it – these are puts whose updates are in the scanned key range, and either do not have a version yet or have versions lower than the scan time. This is done by busy waiting on the PO entry until it changes; monotonically increasing counters are used in order to avoid ABA races.

3.3 Normal operation flow

Get and put. Algorithm 1 presents pseudocode for the normal operation flow, without rebalance. Both operations begin by locating the target chunk using the lookup function. In principle, this can be done by traversing the linked list, but that would result in a linear search time. To expedite the search, lookup searches the index first. However, index updates are lazy – they occur after the new chunk is already in the linked list – and therefore the index may return a stale chunk that has already been replaced by rebalance or miss newly added chunks. To this end, the index search is repeated with a smaller key in case the index returns a stale chunk, and the index search is supplemented by a linked-list traversal. A similar approach was used in earlier works [19, 39].

A get proceeds without synchronization. If the chunk has a munk, get locates the key in it by first using binary search in its sorted prefix, and then traversing the linked list edges. Otherwise, the get searches the key in the row cache. If it is not found, it queries the Bloom filter to learn whether the key might be present in the target chunk's log, and if so, searches for it there. Finally, if the key is not in any of these places, it searches the SSTable on disk.

Upon locating the chunk, a put grabs its rebalanceLock in shared mode to ensure that the chunk is not being rebalanced during its operation. It then publishes itself in the PO as described in §3.2 above. It proceeds to write the new key-value pair to the funk's log and to the munk (if one exists). It also updates the row cache in case the key is present there. The munk and funk are multi-versioned to support atomic scans, whereas the row cache is not used by scans and holds only the latest version of each key. In case the put's version is the same as the current latest one, it over-writes the current one in the munk but appends a new one in the funk's log.

We note that in case multiple puts update the same key concurrently, subtle races may arise. For example, two concurrent puts may update the funk and munk (or the funk and row cache) in different orders, and so the latest update to one will not coincide with the latest update to the other. This can be addressed, for example, by locking keys. In our implementation we address this issue using monotonically increasing counters that determine the order of puts within a version (a similar approach was taken in [19]); we enforce updates to occur in order of version-counter pairs everywhere (including the row cache).

Finally, a put releases the chunk's lock and unregisters itself from PO.

Algorithm 1 YoDB normal operation flow for thread i.

```
1:  procedure GET(key)
2:      $C \leftarrow$ lookup(key)
3:      if $C$.munk then
4:          search key in $C$.munk linked list; return
5:      search key in row cache; return if found
6:      if key$\in C$.bloomFilter then
7:          search key in funk.log; return if found
8:      search key in funk.SSTable; return if found
9:      return NULL

10: procedure PUT(key, val)
11:     $C \leftarrow$ lookup(key)
12:     lockShared($C$.rebalanceLock)
13:     PO[i] $\leftarrow \langle$put, key, $\perp\rangle$        ▷ publish thread's presence
14:     gv $\leftarrow$ GV                                ▷ read global version
15:     PO[i] $\leftarrow \langle$put, key, gv$\rangle$                 ▷ and write in PO
        ▷ write to funk log, munk (if exists), and row cache
16:     append $\langle$key, val, gv$\rangle$ to funk.log
17:     if $C$.munk then
18:         add $\langle$key, val, gv$\rangle$ to $C$.munk's linked list
19:     update $\langle$key, val$\rangle$ in row cache (if key is present)
20:     unlock($C$.rebalanceLock)
21:     PO[i] $\leftarrow \perp$

22: procedure SCAN(key1, key2)
23:     gv $\leftarrow$ F&I(GV)                        ▷ read global version
24:     PO[i] $\leftarrow \langle$scan, key1, key2, gv$\rangle$          ▷ publish in PO
25:     $T \leftarrow$ PO entries updating keys in range [key1, key2]
26:     wait until $\forall t \in T, t$ completes or has a version $>$ gv
27:     $C \leftarrow$ lookup(key1)
28:     repeat
29:         collect from $C$.munk or $C$.funk (log and kstore)
             max version $\leq$gv for all keys in [key1, key2]
30:         $C \leftarrow C$.next
31:     until reached key2
```

Scan. A scan first fetches-and-increments the GV to obtain its snapshot time gv. It then publishes itself in PO with its key-range and gv, to signal to concurrent rebalances not to remove versions it needs. Next, it waits for pending puts that affect its key range to complete or obtain larger versions. Finally, it collects the relevant values from all chunks in the scanned range. Specifically, if the chunk has a munk, the scan reads from it, for each key in its range, the latest version of the value that precedes its snapshot time. Otherwise, the scan collects all the relevant versions for keys in its range from both the SSTable and the log and merges the results.

3.4 Reorganization

Rebalance is used to improve data organization in a chunk's funk or munk by removing old versions that are no longer needed for scans, removing deleted items, and sorting all the keys in the chunk. It can be invoked by a thread attempting to access the chunk or a dedicated background thread. Funk rebalance is important for two main reasons: (1) to reduce the time spent searching for a key in the log, and (2) to reduce disk space occupancy. In case a chunk has a munk, rebalance usually reorganizes only the munk, since all searches are served by it. The respective funk is reorganized much less frequently, only in order to bound disk space occupancy.

Reorganization involves creating a new funk or munk to replace the current one. In some cases, the chunk itself is split, creating new funks (and munks if applicable).

Because the new funk or munk contains the same relevant data as the replaced one, gets and scans can proceed uninterrupted while rebalance is taking place. However, in order to avoid data loss, puts need to wait. To this end, a munk rebalance begins by obtaining the chunk's rebalanceLock in exclusive mode, blocking puts, which acquire the lock in shared mode. When the lock is held, the chunk is *immutable*, and otherwise it is *active*. When the new munk is ready, the rebalance process replaces the munk pointer in the chunk and releases rebalanceLock, thus re-activating the chunk.

Since funk reorganization may take a long time, we allow the chunk to be active while the new funk is created, and then make it immutable for a short time. In order to avoid redundant I/O, we use the funkChangeLock to ensure only one thread works to create a new funk. Once that thread completes, it acquires the rebalanceLock in exclusive mode. It then copies to the new chunk any new items added to log in the old chunk before it became immutable. When this is done, it replaces the funk pointer in the chunk and releases the lock, re-activating the chunk.

In case of a split, the chunk is immutable when we create two new chunks to replace it. If the chunk has a munk, we split the munk (by creating two new munks) and update the appropriate pointers in the new chunks. Since creating new funks again involves I/O, we do not wish to keep the new chunks immutable for the duration of this process, and allow funk creation to proceed in the background while the two new chunks still point to the same old funk.

After we replace the old chunk in the list with the two new ones, the old chunk is still accessible via the chunk index (even though it is no longer in the list). The new chunks are therefore created in *baby* status, indicating that they are still immutable. Once the new chunks are indexed, the old chunk is *aged*, and the new chunks can become mutable. At this point, we change their status to *child*, indicating that they are no longer immutable, but share a funk with another chunk, and so should not be rebalanced. Once the funk split completes, we make the chunk immutable in order to complete the funk switch, and then change their status to active.

Merges can be handled the same way by making the two merged munks immutable for the duration of the switch; we do not implement this feature in our prototype.

3.5 Disk flushes and recovery

Like most popular KV-stores, YoDB supports two modes of operation – *synchronous* and *asynchronous*. With the former, updates are persisted to disk before returning to the user, and so a user is ensured when its operation completes that the written data will survive failures. The drawback of this approach is that it is roughly an order-of-magnitude slower than the asynchronous mode in existing KV-stores like RocksDB [9] as well as in YoDB.

The asynchronous mode expedites updates by performing them in RAM only and periodically *flushing* them to disk. This reduces write latency and increases throughput, but may lead to loss of data that was written shortly before the crash. The tradeoffs between the two approaches are well known and the choice is typically left to the user.

Recovery semantics. In the synchronous mode, the funks always reflect all completed updates. In this case, recovery is straightforward: we simply construct the chunks linked list and chunk index from the funks on disk, and then the database is immediately ready to serve new requests, populating munks and Bloom filters on-demand.

In the asynchronous mode, some suffix of the data written before a crash may be lost, but we ensure that the data store *consistently* reflects a *prefix* of the values written. For example, if put(k1, v1) completes before put(k2, v2) is invoked and then the system crashes, then following the recovery, if k2 appears in the data store, then k1 must appear in it as well. Such recovery to a *consistent snapshot* of the data store is important, since later updates may depend on earlier ones.

Checkpointing for consistent recovery. To support recovery to a consistent snapshot in asynchronous mode, we use a background process to periodically create and persist *checkpoints* of the data store. The checkpointing process creates a snapshot similarly to the atomic scan mechanism. That is, it begins by fetching-and-incrementing GV to obtain a snapshot version gv. Next, it synchronizes with pending puts via PO to ensure that all puts with smaller versions are complete. It then flushes all the pending writes to disk (using an fsync call). Once the flush is complete, it writes gv to a dedicated *checkpoint file* on disk, indicating that all updates pertaining to versions smaller than or equal to this version have been persisted. Note that some puts with higher versions than gv might be reflected on disk while others are not.

Recovery in YoDB is lazy, keeping all data on disk, and allowing it to be fetched from disk into munks on demand in the course of the normal operation mode. But to ensure consistency, following a recovery, retrievals from funks should ignore newer versions that were not included in the latest completed checkpoint before the crash. This must be done by every operation that reads data from a funk – get or scan from a munk-less chunk, funk rebalance, or a munk load.

To facilitate this check, we distinguish between pre-crash versions and new ones created after recovery using *incarnation numbers*. Specifically, a version is split into an incarnation number (in our implementation, the four most-significant bits of the version) and a per-incarnation version number. The normal mode operation incrementing the GV in effect increases the latter. The recovery procedures increments the former and resets the latter, so versions in the new incarnation begin from zero.

We maintain a recoveryTable mapping each recovered incarnation to its last checkpointed version number. For example, Table 1 shows a possible state of the recovery table after two recoveries, i.e., during incarnation 2.

incarnation	last checkpointed version
0	1375
1	956

Table 1: Example recovery table during incarnation 2.

Every read from a funk (during get, scan, funk rebalance, and munk load) then refers to the recovery table in order to identify versions that should be ignored – these are versions from old incarnations that exceed the checkpoint number for their incarnation.

The recovery procedure is very simple: it reads the checkpoint time from disk, loads the recoveryTable into memory, adds a new row to it with the last incarnation and latest checkpoint time, and persists it again. It then increments the incarnation number, and resumes normal operation with version number 0 in the new incarnation.

4 Implementation

We implement YoDB in C++. We borrow the SSTable implementation from the RocksDB open source [9]. Similarly to RocksDB, we use jemalloc for memory allocation.

We now describe some of our implementation choices.

Chunk index is implemented as a sorted array holding the minimal keys of all chunks. Whenever a new chunk is created (upon split), the index is rebuilt and the reference to it is atomically flipped. We found this simple implementation to be fastest since splits are infrequent.

Munk's cache applies simple LFU eviction policy, in which the score is a weighted average of the number of accesses per operation type. We use exponential decay to maintain the recent access counts, similar to [27]: periodically, all counters are sliced by a factor of two.

The row cache, on the other hand, implements a coarse-grained LRU policy by maintaining a fixed-size queue of hash tables. New entries are inserted into the head table. Once it overflows, a new empty table is added to the head, and the tail is discarded. Consequently, lookups for recently cached keys are usually served by the head table, and unpopular keys are removed from the cache in a bulk, once the tail table is dropped.

The row cache must never serve stale values. Therefore, a put updates the cache if a previous version of that key is already in the cache. If the key is not present in the cache, the put does not update it, to avoid overpopulating in write-dominated workloads. After a get, the up-to-date version is added to in the head table unless it is already there. If the key's version also exists in another table, its value is shared by the two versions, to avoid duplication.

5 Evaluation

We compare the YoDB prototype to RocksDB – a mature industry-leading KV-store implementation – under a variety of scenarios. We use the most recent RocksDB release 5.17.2, available Oct 24, 2018. It is worth noting that RocksDB's performance improved significantly during the last two years, primarily through optimized LSM compaction algorithms [23].

The experiment setup is described in §5.1. Performance results for YoDB and RocksDB in different workloads are presented in §5.2. §5.3 analyzes YoDB's sensitivity to different configuration settings and application parallelism.

5.1 Setup

Testbed. We employ a C++ implementation [10] of YCSB [26], the de facto standard benchmarking platform for KV-stores. YCSB provides a standard set of APIs and a workload suite. The platform decouples data access from workload generation, thereby providing common ground for backend comparison.

A workload is defined by (1) the ratio of get, put, and scan accesses, and (2) access pattern defined as a synthetic distribution of keys. YCSB provides a basic set of benchmarks inspired by real-life applications, and allows developing new ones through workload generation APIs. A typical YCSB instance stress-tests the backend KV-store through a pool of concurrent worker threads that drive identical workloads.

Our hardware is a 12-core Intel Xeon 5 machine with 4TB SSD disk. Unless otherwise stated, the driver application exercises 12 workers in all experiments. In order to guarantee a fair memory allocation to all KV-stores, we run each experiment within a Linux container with 16GB RAM.

Metrics. Our primary performance metric is *throughput* and *latency percentiles*, as produced by YCSB. In addition, we measure *write amplification*, namely, bytes written to storage over bytes passed from the application. In order to explain the results, we also explore *read amplification* (in terms of bytes as well as number of system calls per application read). The OS performance counters are retrieved from the Linux proc filesystem.

Workloads. We scale the dataset size from 4GB to 64GB, in order to exercise multiple locality scenarios with respect to the available 16GB of RAM. Similarly to the published RocksDB benchmarks [5], we use 10-byte keys that YCSB pads with a fixed 4-byte prefix (effectively, 14-byte keys), and 800-byte values. The data is stored uncompressed.

We study two different key-access distributions:

1. *Zipf-simple* – the standard YCSB Zipfian distribution over simple (non-composite) keys. Key access frequencies are sampled from the heavy-tailed Zipf distribution, following the description in [29], with $\theta = 0.8$. The ranking is over a random permutation of the entire key range, so popular keys are uniformly dispersed. This workload exhibits medium temporal locality (e.g., the most popular key's frequency is approximately 0.7%), and no spatial locality.

2. *Zipf-composite* – a Zipfian distribution over composite keys. The key's 14 most significant bits comprise the primary attribute. Both the primary attribute and the remainder of the key are drawn from Zipf ($\theta = 0.8$) distributions over their ranges. Zipf-composite exhibits high spatial locality, representing workloads with composite keys, such as message threads [20], social network associations [17], and analytics engines [1].

The workloads exercise different mixes of puts, gets, and scans. We use standard YCSB scenarios (A to E) that range from write-savvy (50% puts) to read-savvy (95% – 100% gets or scans) settings. In order to stress the system even more on the write side, we introduce a new workload, named YCSB-P, comprised of 100% puts. It captures a heavy-duty data load scenario (e.g., from an external data pipeline).

Evaluation methodology and configuration. Each experiment consists of two stages. The first stage builds the dataset, by filling an initially empty store with a sequence of KV-pairs, ordered by key. The second phase exercises the specific scenario; all worker threads follow the same access pattern. Most experiments perform 80 million data accesses. The experiments that run scans perform 4 to 16 million accesses, depending on the scan size. We run 5 experiments for each data point, and present the median metric measurement.

We only present the performance metrics in asynchronous logging mode, since synchronous logging is approximately 10 times slower, thereby trivializing the results of every scenario that includes puts.

All experiments in §5.2 use the same YoDB configuration, to avoid over-tuning; §5.3 provides insights on parameter choices. We allocate 8GB to munks and 4GB to the row cache, so together they consume 12GB out of the 16GB container. The row cache uses three hash tables. The Bloom filters for munk-less funks are partitioned 16-way. We set the YoDB chunk size limit to 10MB, and the rebalance threshold factor to 0.7 – i.e., chunks can grow up to 7MB before being compacted. The funk log size limit is 2MB for munk-less chunks, and 20MB for chunks with munks.

We use RocksDB with the default configuration, which is also used by its public performance benchmarks [5].

5.2 YoDB versus RocksDB

Figure 2 presents throughput measurements under the Zipf-simple and Zipf-composite key distributions in the put-only workload YCSB-P, the mixed YCSB-A, the get-only YCSB-C, and the scan-heavy YCSB-E with three different scan sizes. YCSB-B is omitted because the results are very close to those for YCSB-C. For completeness, we also experiment with YCSB-D that exercises the Latest workload – frequent reads of the recently added simple keys. The results are also close to YCSB-B, and omitted as well. See the YCSB-B and YCSB-D experiments in the supplementary material.

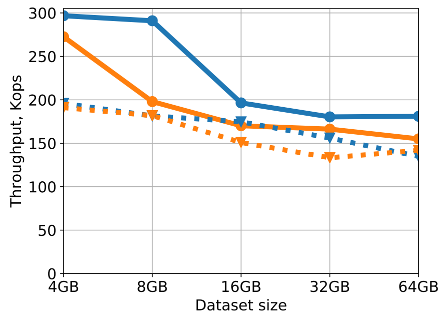
We now discuss the results for the different scenarios.

Put-only. In YCSB-P (100% put, Figure 2a), YoDB's throughput is 1.3x to 1.6x versus RocksDB's with composite keys, and 1.1x to 1.5x with simple keys. YoDB benefits from spatial locality whereas RocksDB's write performance is relatively insensitive to it, as is typical for LSM stores.

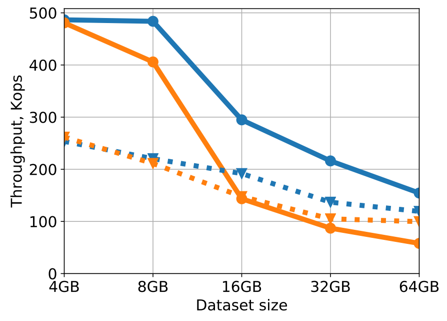
This workload's bottleneck is the reorganization of persistent data (funk rebalances in YoDB, compactions in RocksDB), which causes write amplification and hampers performance. For small datasets (8GB or less), YoDB accommodates all puts in munks, and so funk rebalances are rare. In big datasets, funk rebalances do occur, but mostly in munk-less chunks, which are accessed less frequently than popular ones. Hence, in both cases, funk rebalances incur less I/O than RocksDB's compactions, which do not distinguish between hot and cold data.

YoDB's high log size limit for chunks with munks also reduces I/O by decelerating munk flushes. This does not impact recovery time since YoDB does not replay its logs.

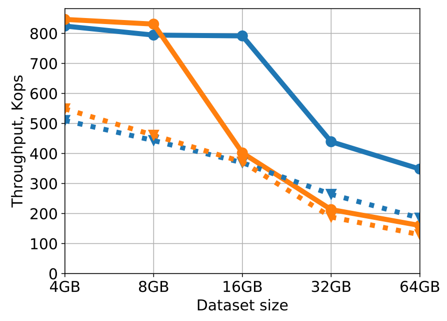
The write amplification in this workload is summarized in Table 2. We see that YoDB reduces the disk write rate dramatically, with the largest gain observed for big datasets



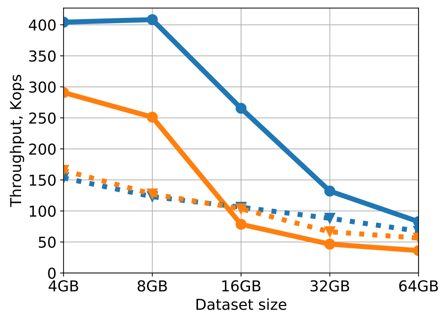
(a) YCSB-P – put-only



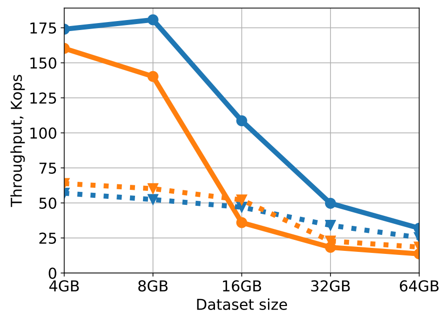
(b) YCSB-A – mixed get-put



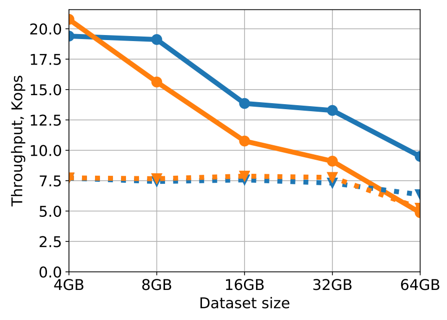
(c) YCSB-C – get-only



(d) YCSB-E10 – short scans



(e) YCSB-E100 – medium scans



(f) YCSB-E1000 – long scans

Figure 2: YoDB versus RocksDB throughput (Kops), under multiple workloads with Zipf distributions over composite (two-component) and simple (non-composite) keys and scaling dataset sizes.

(e.g., 1.29 versus 2.92 for the 64GB database under Zipf-composite). Thus, YoDB also reduces disk wear.

	4GB	8GB	16GB	32GB	64GB
Zipf-composite:					
RocksDB	1.93	2.15	2.29	2.54	**2.92**
YoDB	1.42	1.38	1.27	1.30	**1.29**
Zipf-simple:					
RocksDB	1.94	2.20	2.69	3.03	2.77
YoDB	1.41	1.37	1.15	1.13	1.31

Table 2: YoDB versus RocksDB write amplification under the put-only workload (YCSB-P) and scaling dataset sizes.

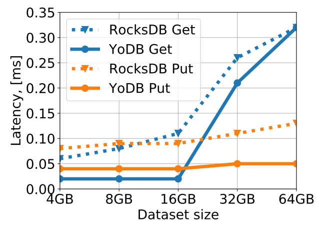
Mixed put-get. YCSB-A (50% put, 50% get, Figure 2b) is particularly challenging for YoDB because it exercises a high contention between gets and puts. Here, the bottleneck is disk search in gets, primarily the linear search in funk logs that keep filling up due to concurrent puts. (In-memory searches are three orders of magnitude faster.)

YoDB achieves 1.3x to 2.2x throughput versus RocksDB with composite keys, thanks to better exploitation of spatial locality. Figure 3a shows that YoDB is also faster wrt the tail (95%) put and get latency in this scenario. With simple keys, YoDB is faster than RocksDB for small datasets that fit in memory, and slower for big datasets. Figure 3b zooms in on this setting: while YoDB is on par with RocksDB wrt the put latency in this setting, it falls short wrt the get latency.

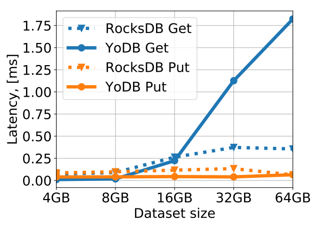
When the working set does not fit in RAM, disk access dominates the tail latencies. Figure 4a depicts the distribution of gets by the storage component that fulfils the request, and Figure 4b presents the disk search latencies by component. For example, for the 64GB dataset, 1.9% of gets are served from logs under Zipf-composite, versus 4.0% under Zipf-simple, and the respective log search latencies are 1.7 ms vs 4.4 ms. This happens because in the latter, puts are more dispersed, hence the funks are (1) cached less effectively by the OS, and (2) rebalanced less frequently so their logs grow longer.

Naturally, the efficiency of in-memory caching is paramount. In the same example, the RAM hit rate (munk and row cache combined) is 90.5% with composite keys versus 79% without them. The row cache becomes instrumental as spatial locality drops – it serves 32.8% of gets for Zipf-simple versus 12.4% for Zipf-composite.

Scan-dominated. In YCSB-E (5% put, 95% scan, Figures 2d- 2f). the number of items to iterate through is sampled uniformly in the range [1,S], where S is either 10, 100, or 1000. This workload (except with very short scans on very large datasets) is the best for YoDB, since it exhibits the spatial locality the system has been designed for. YoDB

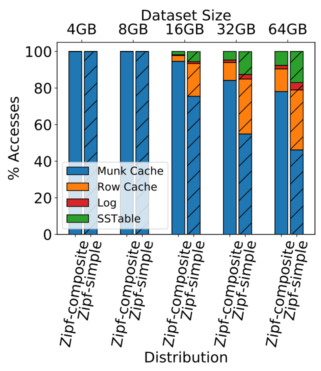


(a) Zipf-composite

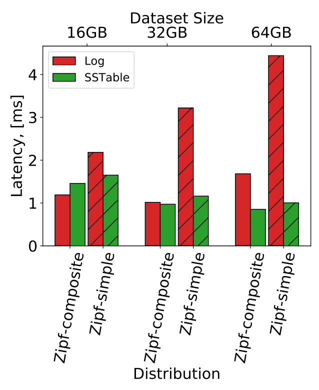


(b) Zipf-simple

Figure 3: YoDB versus RocksDB 95% latency (ms), under a mixed get-put workload (YCSB-A).



(a) Fraction of get accesses



(b) On-disk get access latency

Figure 4: YoDB 95% tail latency analysis by the serving component, under a mixed get-put workload (YCSB-A).

achieves 1.25x to 3x throughput versus RocksDB under Zipf-composite, and in Zipf-simple, improves over RocksDB in small datasets. §5.3 discusses how the scan performance on big datasets could be improved through dynamic adaptation of the funk log size limit.

Get-only. We see that in YCSB-C (100% get, Figure 2c), YoDB achieves 1.6x to 2.1x throughput versus RocksDB with composite keys, and up to 1.6x with simple ones.

Since YCSB-C only exercises the read path, performance hinges on caching efficiency. Both YoDB and RocksDB benefit from OS (filesystem) caching in addition to application-level caches. Table 3 compares the two systems' read amplification (as a proxy for cache hit ratio) with composite keys, in terms of (1) actual disk bytes read and (2) read system calls. Under the first, RocksDB is slightly better in large datasets. However, it relies extensively on the OS cache – in the 64GB dataset, it performs almost 12 times as many system calls as YoDB, wasting the CPU resources on kernel-to-user data copy. RocksDB developers explain that the block cache scaling potential is limited in their database, due to tension between its read-path and write-path RAM resources [11]. YoDB, in contrast, exploits its munk cache for both reads and writes, which leads to better RAM utilization.

	4GB	8GB	16GB	32GB	64GB
RocksDB, bytes	0.05	0.11	0.20	0.47	0.95
YoDB, bytes	0.0	0.0	0.11	0.80	1.07
RocksDB, syscall	3.84	4.01	4.14	4.28	**4.37**
YoDB, syscall	0.0	0.0	0.10	0.24	**0.41**

Table 3: YoDB versus RocksDB read amplification, in terms of bytes and system calls, under a read-only workload (YCSB-C) with Zipf-composite distribution.

5.3 YoDB Insights

Vertical scalabiliy. Figure 5 illustrates YoDB's throughput scaling for the 64GB dataset under both Zipf distributions. We exercise the YCSB-A, YCSB-C and YCSB-P scenarios, with 1 to 12 worker threads. As expected, YCSB-C (read-only scenario) scales nearly perfectly (9.4x for composite keys, 7.7x for simple ones). The other workloads scale slower, due to read-write and write-write contention as well as background munk and funk rebalances.

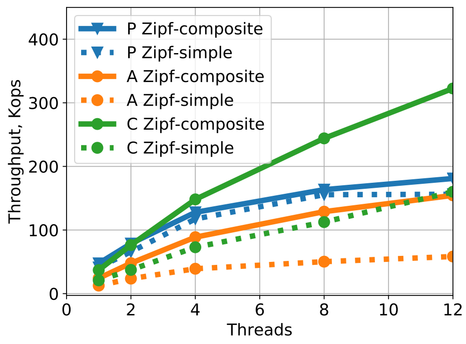
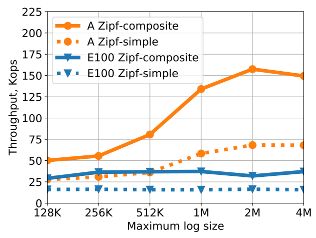
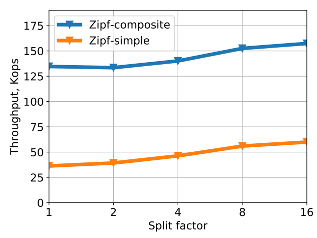


Figure 5: YoDB scalability with the number of threads for the 64GB dataset and different workloads.



(a) Maximum log size, YCSB-A and YCSB-E100



(b) Bloom filter split factor, YCSB-A

Figure 6: YoDB throughput sensitivity to configuration parameters, on the 64GB dataset under the YCSB-A (mixed put-get) and YCSB-E100 (scan-dominant, 1 to 100 items).

Configuration parameters. We explore the system's sensitivity to funk-log configuration parameters, for the most challenging 64GB dataset, and explain the choice of the default values.

Figure 6a depicts the throughput's dependency on the log size limit for munk-less funks, under the YCSB-A and YCSB-E100 workloads with Zipf-composite key distribution. The fraction of puts in YCSB-A is 50% (versus 5% in YCSB-E), which makes it more sensitive to the log size. A low threshold (e.g., 128KB) causes frequent funk rebalances, which degrades the performance more than 3-fold. On the other hand, too high a threshold (4MB) lets the logs grow bigger, and slows down gets. Our experiments are tuned to use 2MB logs, which favors write-intensive workloads. YCSB-E favors smaller logs, since the write rate is low, and more funk rebalances can be accommodated. Its throughput could grow by up to 20% by tuning to use 512KB logs.

Figure 6b depicts the throughput dependency on the Bloom filter split factor, under YCSB-A. Partitioning to 16 mini-filters gives the best result; beyond this point the benefit levels off. The impact of Bloom filter partitioning on the end-to-end get latency as well as the memory footprint is negligible.

6 Related Work

The vast majority of industrial mainstream NoSQL KV-stores are implemented as LSM trees [7, 9, 16, 24, 32], building on the foundations set by O'Neil et al. [36, 35].

Due to LSM design popularity, much effort has been invested into working around its bottlenecks. A variety of compaction strategies has been implemented in production systems [23, 41] and research prototypes [18, 38]. Other suggestions include storage optimizations [34, 38], boost of in-memory parallelism [16, 28], or leveraging workload redundancies to defer disk flushes [18, 21].

For example, PebblesDB [38] introduces fragmented LSM trees, in which level files are sliced into *guards* of increasing granularity and organized in a skiplist-like layout. This structure reduces write amplification. In contrast, YoDB eliminates the concept of levels altogether, and employs a flat storage layout instead. WiscKey [34] separates key and value storage in SSTables, also in order to reduce amplification. This optimization is orthogonal to YoDB's concepts, and could benefit our work as well. Accordion [21] splits the LSM in-memory buffer into mutable and immutable levels, which are periodically merged in the background similarly to traditional compaction, in order to reduce disk flushes. This mechanism is similar to munk rebalance in YoDB, but the latter rebalances data at the chunk level and so benefits from spatial locality.

YoDB's design resembles classic B-trees [31] by supporting direct random updates to leaf blocks (chunks). It resolves the B-tree write throughput issue through munks and in-memory compaction. B-trees, which have been designed prior to multi-versioning concurrency control (MVCC) methods, heavily rely on locks for consistency of operation; this method is inferior to MVCC in terms of performance.

In recent years, B^{ε}-trees [22] have emerged as promising way to organize data. They are used in production KV-stores [15] and filesystems [30]. B^{ε}-tree is a B-tree variant that uses overflow buffers in internal nodes as well as leaves, trading buffer compactions for random I/O. YoDB applies similar ideas to leaf-level storage. TokuDB's [15] performance is on par with RocksDB in many use cases, but its disk image tends to be bigger [6].

Tucana [37] is a B^{ε}-tree optimization that uses three techniques to reduce overheads: copy-on-write, private allocation, and memory-mapped I/O. YoDB could incorporate some of these to further improve performance. However, Tucana paper does not provide consistent semantics for scans that span multiple leaf segments.

In-memory KV-stores [2, 4, 14, 40] have originally emerged as fast volatile data storage, e.g., web and application caches. Over time, most of them evolved to support durability, albeit as a second-class citizen in most cases. These systems resemble YoDB in their memory-centric approach. However, we are unaware of their consistency guarantees or performance optimizations with respect to disk-resident data in such systems.

7 Conclusions

We presented YoDB – a novel persistent KV-store designed for workloads with high spatial locality, as prevalent in modern data-driven applications. YoDB provides strong (atomic) consistency guarantees for random updates, random lookups, and range queries. YoDB outperforms the state-of-the-art RocksDB LSM store in the majority of YCSB benchmarks, with both standard and spatially-local key distributions, in which it excels in particular. YoDB further reduces write amplification to near-optimal under write-intensive workloads. Finally, it provides near-instant recovery from failures, in contrast to traditional data stores based on centralized write-ahead logs. YoDB is presented as a conceptual prototype, which can be improved in multiple ways through ideas borrowed from other designs.

References

[1] Flurry analytics. `https://developer.yahoo.com/flurry/docs/analytics/`.

[2] Ignite database and caching platform. https://ignite.apache.org/.

[3] NoSQL market is expected to reach $4.2 billion, globally, by 2020. https://www.alliedmarketresearch.com/press-release/NoSQL-market-is-expected-to-reach-4-2-billion-globally-by-2020-allied-market-research.html.

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

[5] RocksDB tuning guide. https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guidel.

[6] TokuDB vs RocksDB. https://www.percona.com/live/17/sessions/tokudb-vs-rocksdb.

[7] Apache hbase, a distributed, scalable, big data store. `http://hbase.apache.org/`, Apr. 2014.

[8] A fast and lightweight key/value database library by google. `http://code.google.com/p/leveldb`, Jan. 2014.

[9] A persistent key-value store for fast storage environments. `http://rocksdb.org/`, June 2014.

[10] Yahoo! Cloud Serving Benchmark in C++, a C++ version of YCSB. `https://github.com/basicthinker/YCSB-C`, 2014.

[11] Increase default size of RocksDB block cache. `https://github.com/facebook/mysql-5.6/issues/441`, 2016.

[12] Average selling price of DRAM 1Gb equivalent units from 2009 to 2017 (in U.S. dollars). `https://www.statista.com/statistics/298821/dram-average-unit-price/`, 2018.

[13] Flurry Analytics Reporting API. `https://developer.yahoo.com/flurry/docs/api/code/analyticsapi/`, 2018.

[14] Memcached, an open source, high-performance, distributed memory object caching system. `https://memcached.org/`, Dec. 2018.

[15] Percona TokuDB. `https://www.percona.com/software/mysql-database/percona-tokudb`, 2018.

[16] Scylla the real-time big data database. `https://www.scylladb.com/`, 2018.

[17] Armstrong, T. G., Ponnekanti, V., Borthakur, D., and Callaghan, M. Linkbench: A database benchmark based on the facebook social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2013), SIGMOD '13, ACM, pp. 1185–1196.

[18] Balmau, O., Didona, D., Guerraoui, R., Zwaenepoel, W., Yuan, H., Arora, A., Gupta, K., and Konka, P. Triad: Creating synergies between memory, disk and log in log structured key-value stores. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference* (2017), USENIX ATC '17, pp. 363–375.

[19] Basin, D., Bortnikov, E., Braginsky, A., Golan-Gueta, G., Hillel, E., Keidar, I., and Sulamy, M. Kiwi: A key-value map for scalable real-time analytics. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2017), PPoPP '17, ACM, pp. 357–369.

[20] Borthakur, D., Gray, J., Sarma, J. S., Muthukkaruppan, K., Spiegelberg, N., Kuang, H., Ranganathan, K., Molkov, D., Menon, A., Rash, S., Schmidt, R., and Aiyer, A. Apache hadoop goes realtime at facebook. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data* (2011), SIGMOD '11, pp. 1071–1080.

[21] Bortnikov, E., Braginsky, A., Hillel, E., Keidar, I., and Sheffi, G. Accordion: Better memory organization for lsm key-value stores. *Proc. VLDB Endow. 11*, 12 (Aug. 2018), 1863–1875.

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

[23] Callaghan, M. Name that compaction algorithm. `https://smalldatum.blogspot.com/2018/08/name-that-compaction-algorithm.html`, 2018.

[24] 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 Trans. Comput. Syst. 26*, 2 (June 2008), 4:1–4:26.

[25] Cho, J., Garcia-Molina, H., and Page, L. Efficient crawling through url ordering. In *Proceedings of the Seventh International Conference on World Wide Web 7* (1998), WWW7, pp. 161–172.

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

[27] Einziger, G., Friedman, R., and Manes, B. Tinylfu: A highly efficient cache admission policy. *ACM Trans. Storage 13*, 4 (Nov. 2017), 35:1–35:31.

[28] Golan-Gueta, G., Bortnikov, E., Hillel, E., and Keidar, I. Scaling concurrent log-structured data stores. In *EuroSys* (2015), pp. 32:1–32:14.

[29] Gray, J., Sundaresan, P., Englert, S., Baclawski, K., and Weinberger, P. J. Quickly generating billion-record synthetic databases. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data* (1994), SIGMOD '94, pp. 243–252.

[30] Jannen, W., Yuan, J., Zhan, Y., Akshintala, A., Esmet, J., Jiao, Y., Mittal, A., Pandey, P., Reddy, P., Walsh, L., Bender, M., Farach-Colton, M., Johnson, R., Kuszmaul, B. C., and Porter, D. E. BetrFS: A right-optimized write-optimized file system. In *13th USENIX Conference on File and Storage Technologies (FAST 15)* (2015), pp. 301–315.

[31] Knuth, D. E. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[32] Lakshman, A., and Malik, P. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev. 44*, 2 (Apr. 2010), 35–40.

[33] Lehman, P. L., and Yao, s. B. Efficient locking for concurrent operations on b-trees. *ACM Trans. Database Syst. 6*, 4 (Dec. 1981), 650–670.

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

[35] Muth, P., O'Neil, P. E., Pick, A., and Weikum, G. 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* (1998), VLDB '98, pp. 452–463.

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

[37] 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* (2016), USENIX ATC '16, pp. 537–550.

[38] Raju, P., Kadekodi, R., Chidambaram, V., and Abraham, I. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), SOSP '17, pp. 497–514.

[39] Spiegelman, A., Golan-Gueta, G., and Keidar, I. Transactional data structure libraries. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2016), PLDI '16, pp. 682–696.

[40] Srinivasan, V., Bulkowski, B., Chu, W.-L., Sayyaparaju, S., Gooding, A., Iyer, R., Shinde, A., and Lopatic, T. Aerospike: Architecture of a real-time operational dbms. *Proc. VLDB Endow. 9*, 13 (Sept. 2016), 1389–1400.

[41] Tribble, P. How to Ruin Your Performance by Choosing the Wrong Compaction Strategy. `https://www.scylladb.com/2017/12/28/compaction-strategy-scylla/`, 2017.