YoDB – Supplementary Material

1 Chunk Structure

The chunk metadata structure is given in Algorithm 1. The first field is its *status*. It next holds a pointer to the appropriate funk, and either a munk or a Bloom filter, as well as a pointer to the next chunk in the chunk linked list. Note that multiple *generations* of munks may exist for a chunk throughout its life time. Therefore the chunk metadata keeps the generation number of its latest munk, gen, and a per-generation *sequence number*, seq, which is explained below. The chunk's gen and seq are stored together in one 64-bit word to allow atomic access to both of them. Finally, the chunk includes locks to synchronize concurrent access by multiple threads, as explained below.

Algorithm 1 Chunk metadata structure.

status	▷ baby, child, active, immutable, or aged
ptr funk	▷ funk disk address
ptr munk	▷ munk memory pointer
ptr next	▷ next chunk in linked list
int gen	▷ munk generation
int seq	▷ sequence number in current generation
ptr bloomFilter	▷ summary of set of keys in log
asymmetric lock rebalanceLock	▷ shared/exclusive lock
lock funkChangeLock	▷ acquired with try_lock
PPA[threads]	▷ pending puts

2 Synchronization Data Structures

Using a single pending array to synchronize all operations can cause unnecessary contention. We mitigate this problem in our implementation by maintaining two data structures for coordinating different operations. The first is a per-chunk pending put array (*PPA*) which either indicates the current update's key and version, or indicates that the thread is currently not performing a put. The second is a global pending scan array (*PSA*) which tracks versions used by pending scans for compaction purposes; each entry consists of a version and a sequence number, as well as the scans key range. Each entry in the PPAs and PSA includes, in addition to the operation metadata, an ABA sequence number.

A put operation consists of 5 phases:

1. *pre-process* - locate the target chunk; if a munk exists, prepare a cell to insert the value into;
2. *publish* - obtain a version while synchronizing with concurrent scans and rebalances via the chunk's lock and publish indication in the chunk's PPA;
3. *persist* - write the data into the log, indicate it is persisted in the PPA;
4. *link* - if a munk exists, connect the new cell to the linked list, so it can be found through the list traversal, otherwise, update the row cache to latest version if key is present in the cache; and finally
5. *clean* - clear the entry in the chunk's PPA, and increase the entry's ABA number.

If the put fails to acquire the chunk's lock (since it is being rebalanced), the operation restarts, and re-attempts to find an active chunk. Puts trigger both munk and funk rebalances. The former are handled inline when the munk is close to overflow; the latter are done in the background by helper threads.

A per-chunk linearization counter is used to determine the order of concurrent put operations updating the same key. Therefore, the linearization counter is composed of three parts: (a) the GV value; (b) gen - incremented whenever a munk is cached into memory and when the munk is rebalanced; and (c) seq - incremented upon each put and set to the number of KV-pairs in a munk upon a new generation number. When a new chunk is created as a result of a split, the children chunks inherit their generation number from their parent. The linearization counter is written both to the PPA upon publishing the put operation, and to the log when persisting the data. This ensures all operations see the same order of writes per key.

A scan operation first publishes its intent to obtain a version in the PSA. It determines its scan time t by increasing GV and writing it to its entry in the PSA. The scan operation then starts traversing the chunks in its range. For each chunk, it first waits for all put operations that are either with smaller version than t or still have not acquired a version to clear their entry in the PPA or acquire a larger version. After waiting for all concurrent puts to complete, the scan can read the range from the chunk. If a munk exists, it simply reads the range from the linked list, skipping versions that are not last before t. Otherwise, the scan merges the SSTable and log data and reads the range from the result, again skipping the penultimate versions. When the scan completes, it clears the entry in the PSA, and increases the entry's ABA number. Get operations access neither the PSA nor the PPA.

A munk rebalance iterates through the PSA to collect the maximum version number among the active scans that cannot be reclaimed yet. If a scan published its intent in the PSA but published no version number yet, the rebalance waits until either the version is published or the ABA number in the entry is increased.

3 Supplmental Evaluation Results

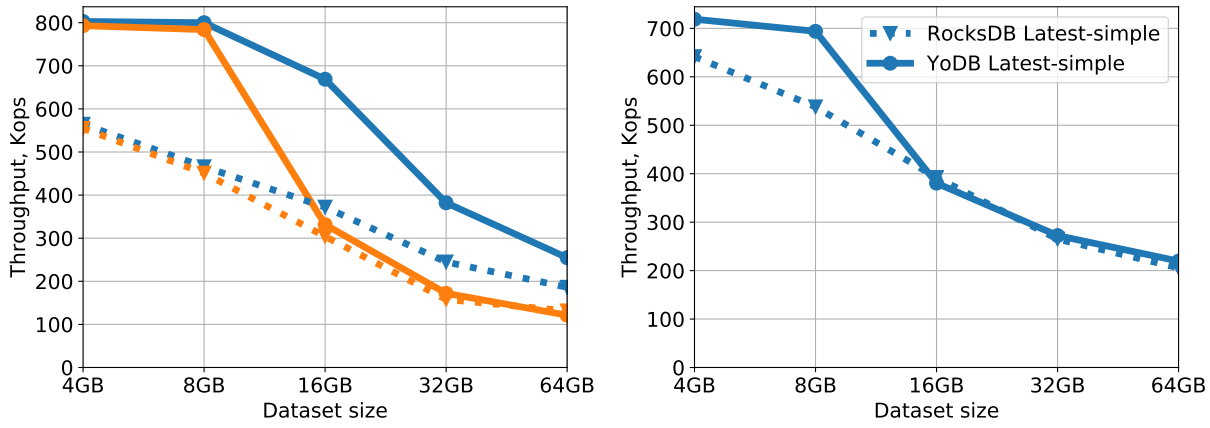
3.1 Get-Dominated Workloads

Figure 1 presents the throughput result for get-dominated workloads. We see that YCSB-B (5% put, 95% get, Figure 1a) achieve results similar to YCSB-C (100% get) that was presented in the main paper. Namely, YoDB achieves up to 1.6x throughput versus RocksDB both with composite keys and with simple ones.

For completeness, we also experimented with the standard YCSB *Latest-simple* distribution. In workload YCSB-D (5% put, 95% get) keys are inserted in sequential order; the read keys' distribution is skewed towards recently added ones. Specifically, the sampled key's position wrt the most recent key is distributed Zipf. This is a workload with medium spatial and temporal locality that represents, e.g., status updates and reads. Throughput results are depicted in Figure 1b.

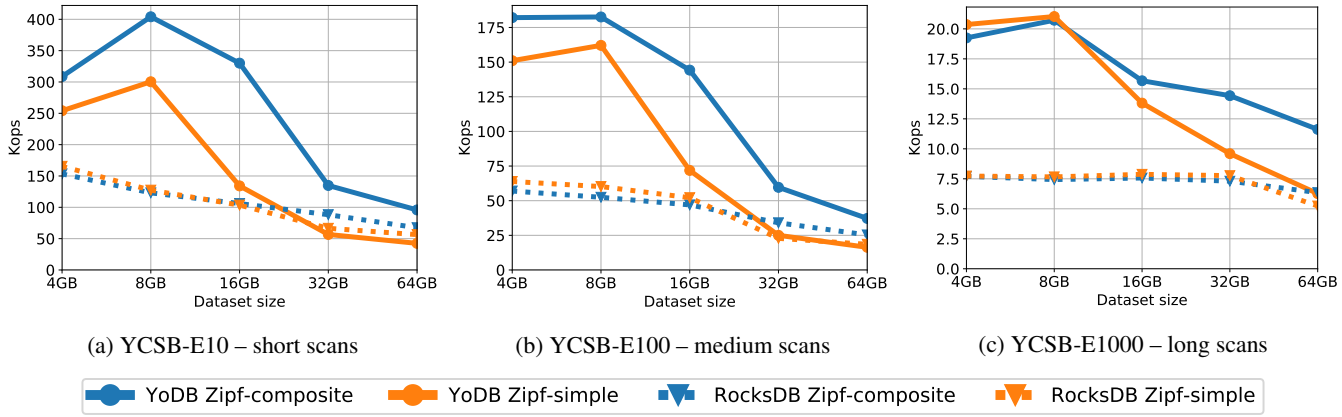
3.2 Scan-Dominated Workloads

In addition to experimenting with the default maximum log size (2MB), we also experimented with smaller logs (256KB maximum size). Figure 2 presents the throughput result for scan-heavy workloads (YCSB-E) with three different scan sizes. It can be seen that with Zipf-simple distribution over large data sets YoDB scans benefit from smaller logs as mergeing them with SSTables is faster.



(a) YCSB-B (b) YCSB-D

Figure 1: YoDB versus RocksDB throughput (Kops), under get-dominated workloads with Zipf distributions over composite (two-component) and simple (non-composite) keys and scaling dataset sizes.



(a) YCSB-E10 – short scans (b) YCSB-E100 – medium scans (c) YCSB-E1000 – long scans

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