TreeLine: An Update-In-Place Key-Value Store for Modern Storage

Geoffrey X. Yu*
Massachusetts Institute of Technology
geoffxy@mit.edu

Markos Markakis*
Massachusetts Institute of Technology
markakis@mit.edu

Andreas Kipf *
Massachusetts Institute of Technology
kipf@mit.edu

Per-Åke Larson
University of Waterloo
gpalarson@outlook.com

Umar Farooq Minhas†
Apple
ufminhas@apple.com

Tim Kraska
Massachusetts Institute of Technology
kraska@mit.edu

ABSTRACT

Many modern key-value stores, such as RocksDB, rely on log-structured merge trees (LSMs). Originally designed for spinning disks, LSMs optimize for write performance by only making sequential writes. But this optimization comes at the cost of reads: LSMs must rely on expensive compaction jobs and Bloom filters—all to maintain reasonable read performance. For NVMe SSDs, we argue that trading off read performance for write performance is no longer always needed. With enough parallelism, NVMe SSDs have comparable random and sequential access performance. This change makes update-in-place designs, which traditionally provide excellent read performance, a viable alternative to LSMs.

In this paper, we close the gap between log-structured and update-in-place designs on modern SSDs with the help of new components that take advantage of data and workload patterns. Specifically, we explore three key ideas: (A) *record caching* for efficient point operations, (B) *page grouping* for high-performance range scans, and (C) *insert forecasting* to reduce the reorganization costs of accommodating new records. We evaluate these ideas by implementing them in a prototype update-in-place key-value store called *TreeLine*. On YCSB, we find that TreeLine outperforms RocksDB and LeanStore by 2.20× and 2.07× respectively on average across the point workloads, and by up to 10.95× and 7.52× overall.

PVLDB Reference Format:
Geoffrey X. Yu, Markos Markakis, Andreas Kipf, Per-Åke Larson, Umar Farooq Minhas, and Tim Kraska. TreeLine: An Update-In-Place Key-Value Store for Modern Storage. PVLDB, 16(1): 99 - 112, 2022.
doi:10.14778/3561261.3561270

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/mitdbg/treeline.

1 INTRODUCTION

Modern persistent key-value stores, such as RocksDB [48] and LevelDB [27], are typically built using log-structured merge trees (LSMs) [53]. The key idea behind LSMs is *buffered log structuring*. Writes are first buffered in memory, and then eventually flushed to immutable files on disk. These files are then periodically compacted (i.e., merged) in the background to remove overwritten and deleted records. LSMs are popular because they provide stellar write performance. They ensure that all disk writes are sequential, which exploits the high sequential write bandwidth of traditional disks.

Yet despite these benefits, LSMs are not a silver bullet in key-value store design. While their design makes writes efficient, it comes at the cost of reads, since records can be present in multiple locations on disk. This is why systems like RocksDB and LevelDB employ block caches, Bloom filters [7, 22], and various compaction strategies [9, 19, 50]—complex and hard-to-tune [47] techniques all aimed at reducing the I/O overhead of reads. For traditional disks (e.g., HDDs and SATA SSDs), this write versus read performance trade-off has been the preferred choice. Random I/O on traditional disks is prohibitively expensive, and so any design that minimizes the amount of random I/O outshines the competition. But is this trade-off still the right one for modern storage devices?

We make the observation that modern NVMe SSDs no longer suffer the same significant random write drawback as traditional disks [29]. With enough request parallelism, NVMe SSDs can achieve their peak sequential write throughput through random writes [29, 39, 56]. This naturally leads us to a research question: how should a persistent key-value store's design change for NVMe SSDs where random writes are comparable to sequential writes in performance?

Our hypothesis is that an *update-in-place* design is the answer for larger-than-memory workloads that are (i) read-heavy, or (ii) skewed write-heavy. Update-in-place designs, such as a classical disk-based B+ tree [16, 51, 52], can offer excellent read performance because each record is stored in a single location on disk—requiring only one I/O to read, if inner nodes are cached in memory. High read performance is desirable because read-heavy workloads such as caching [8, 45] or analytics [4, 11, 40] are common in practice [10].

While disk-based B+ trees do have these read benefits, they are also known to suffer from their own challenges. First, updating a single record on a page requires reading and writing the entire page, which leads to write amplification. Second, scans can lead to random reads because logically consecutive leaf pages are not necessarily stored sequentially on disk; on NVMe SSDs, we observe that random reads still underperform sequential reads. Third, inserts also cause write amplification because of the need to "make space" in the on-disk structure to hold the new records.

Thus, in order to validate our hypothesis, we need to develop a new design for NVMe SSDs that has the read benefits of a classical update-in-place design while also mitigating its traditional write

*The first three authors contributed equally to this paper.
†Work done while at Microsoft Research.

Proceedings of the VLDB Endowment, Vol. 16, No. 1 ISSN 2150-8097.
doi:10.14778/3561261.3561270

drawbacks. In other words, observing that the fast random I/O of NVMe SSDs would be favorable for an update-in-place design, we propose techniques to make such a design competitive across the board. Our new design leverages three complementary techniques: (A) *record caching* to reduce read/write amplification in skewed workloads, (B) *page grouping* to translate scans into sequential reads, and (C) *insert forecasting* to reduce the I/O needed to "make space" for new records. We implement these techniques in *TreeLine*, a new update-in-place key-value store designed for NVMe SSDs.

TreeLine buffers all writes in its record cache (key idea A), allowing it to (i) keep hot records in memory for as long as possible, and (ii) batch writes that go to the same on-disk page to amortize the I/O costs for updating a page. Caching records instead of pages efficiently uses the capacity of the cache even when hot records are not clustered on a few pages. This is true in many real applications: popular items on an e-commerce platform have scattered item IDs, while the activity of users of a social media website is similarly unrelated to their name. In both cases, hot items are updated frequently, illustrating the importance of skewed update-heavy workloads.

Instead of laying out pages randomly on disk, TreeLine *groups* pages storing adjacent key ranges so that they are stored contiguously on disk (key idea B). Doing so lets TreeLine make long physical reads, which benefits scans, while still allowing it to access data at page granularity for point reads. Moreover, TreeLine uses linear models to map records to specific pages within page groups. This helps TreeLine keep its in-memory index small, as it only needs to index the page *group* boundaries (instead of every page boundary).

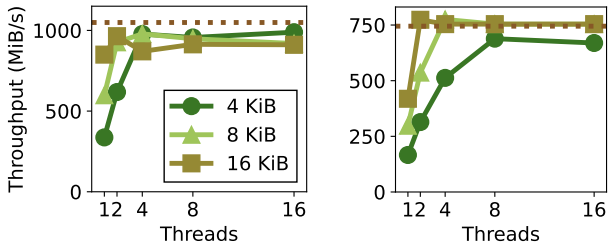
Finally, to address the expensiveness of inserts in an update-in-place design, TreeLine exploits the repetitiveness in skewed insert workloads to *forecast* the location and volume of inserts it expects to receive (key idea C). It uses the forecast to *leave appropriate space* in its on-disk pages, thereby reducing how frequently it needs to reorganize its on-disk pages to accommodate the new records. TreeLine tracks the inserts observed across different parts of the key space and extrapolates these trends forward—a simple but effective technique as we show in our evaluation (see Section 5).

Our proposed techniques provide benefits over other recent designs that revisit update-in-place for NVMe SSDs. LeanStore [36] does not optimize for the out-of-memory case, even though it is efficient when the working set fits into memory. KVell [39] indexes every key, leading to a higher memory overhead than TreeLine, which only indexes segment boundaries using page grouping.

We evaluate TreeLine on YCSB [17] using synthetic and real-world datasets. We compare TreeLine against (i) RocksDB [48], a widely-used LSM key-value store, and (ii) LeanStore [36], a state-of-the-art update-in-place key-value store. Across our point YCSB workloads with 1024 byte records, TreeLine outperforms RocksDB and LeanStore by 2.20× and 2.07× respectively on average. Although TreeLine makes some random reads from disk, we find that it still outperforms RocksDB and LeanStore by 2.50× and 2.80× respectively with 16 threads on uniform scan-heavy workloads.

Contributions. In summary, we make the following contributions:

- We analyze the key performance challenges that arise when employing an update-in-place design on NVMe SSDs and make the case for addressing them with our key ideas.



(a) Intel DC P4510 **(b) AWS c5d.9xlarge**

Figure 1: Random write throughput as we increase the number of concurrent writing threads on two distinct NVMe SSDs. The dotted line is each SSD's peak sequential write throughput. The setup for Figure 1a is as described in Section 5.1.

- We propose *page grouping*: a technique that boosts scan performance by writing logically adjacent pages together onto disk.
- We introduce *insert forecasting*: a technique that predicts the location and volume of inserts to reduce I/O overhead.
- We implement these key ideas, along with *record caching*, into *TreeLine*: a new update-in-place persistent key-value store for NVMe SSDs. We find that it outperforms RocksDB and LeanStore by up to 10.95× and 7.52× respectively overall on YCSB.

We have open-sourced TreeLine [60].

2 WHY REVISIT UPDATE-IN-PLACE DESIGNS?

TreeLine is a new *update-in-place* key-value store for NVMe SSDs. Before diving into its design, we first make the case for why we believe that now is the time to revisit update-in-place designs over LSMs—the current popular design choice.

Random Writes ≈ Sequential Writes on NVMe SSDs. Recently, NVMe SSDs have become readily available. NVMe SSDs follow a completely different storage paradigm compared to traditional hard disk drives. Free from mechanical elements like a spinning disk and a moving head, they instead utilize solid state components and employ an address translation layer to reduce device wear. This shift not only reduces overall access times, but provides two crucial benefits: (i) accessing data sequentially is no longer mechanically superior to accessing them randomly, and (ii) highly parallel requests are now feasible [15], which can be used to hide most of the access latency [56]. To empirically confirm these characteristics, we run a series of experiments on an Intel DC P4510 NVMe SSD [31] and the NVMe SSD on AWS c5d.9xlarge instances [3]. As shown in Figure 1, we use fio [5] to measure the write throughput while varying the (i) request size, and (ii) number of concurrent writing threads. As we increase the number of concurrent writing threads, both devices' *random* write throughputs approach their peak *sequential* write throughputs (1050 MiB/s [31] and 750 MiB/s).

LSMs Leave Read Performance on the Table. As we describe in Section 1, LSMs optimize for writes; but their design also complicates reads. To see how much read performance LSMs "leave on the table", we compare RocksDB (a widely-used LSM key-value store) against our own naïve disk-based B+ tree (an update-in-place system) on an Intel DC P4510 NVMe SSD. We also run TreeLine,

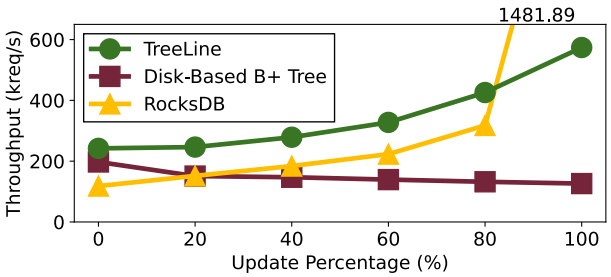


Figure 2: TreeLine, a disk-based B+ tree, and RocksDB compared on Zipfian read/scan/update workloads. We vary the proportion of updates in the workload from 0% to 100%.

our new system. We compare the systems on a Zipfian-distributed ($\theta = 0.79$) workload consisting of reads, updates, and scans on 64 byte records. We vary the proportion of update requests from 0% to 100%. Of the requests that are not updates, 10% are range scans and the rest are point reads. Figure 2 shows each system's throughput in thousands of requests per second. For read-heavy workloads (i.e., less than 40% updates), the limitations of the LSM design lead to RocksDB performing similarly to or worse than our naïve disk-based B+ tree (see Section 5.1 for our experimental setup). TreeLine, using an update-in-place design, takes this performance difference a step further and outperforms RocksDB all the way up to 80% updates. We outline the key ideas behind TreeLine in Section 3 and we describe why and when it outperforms RocksDB in Section 5.

Summary. The advent of modern NVMe SSDs means that random writes are no longer significantly more expensive than sequential writes. We believe that this change affords us an opportunity to reconsider how to design persistent key-value stores. Free from needing to ensure sequential writes, we believe that adopting an update-in-place design is a good choice because they excel at read-heavy workloads—an important and common class of workloads. Although writes are more challenging in an update-in-place design, we present techniques in Section 3 that mitigate these challenges.

3 TREELINE: KEY IDEAS

3.1 Record Caching (Key Idea A)

Workload Skew Does Not Care About Your Layout.

Even in cases where the total size of our data is large enough to warrant using modern storage media instead of a purely in-memory solution, it is usually the case that our working set is much smaller than that, as well as relatively stable in the short term. Given the performance discrepancy between main memory and storage access times, some form of caching would be beneficial. The question then becomes, what granularity this cache should be at.

We argue that record caching is the better option for providing good cache utilization when pursuing an update-in-place design. Certainly, this choice increases the metadata:data ratio for the cache memory usage, since the fixed metadata overhead for each cache entry is now incurred per record (possibly a few bytes) instead of per page (4 KiB in our design). However, the alternative of page caching has the potential for even worse memory consumption.

Consider a working set that is uncorrelated with the key sort order - one example would be a database of user metadata, where the activity of users is not correlated to the value of their `user_id`. An LSM design, like RocksDB [48], would consolidate updates from hot users in the write buffer and later in level 0 files. Caching the blocks in these files and continuing to employ the write buffer would be sufficient to keep the working set in memory.

An update-in-place design, on the contrary, cannot perform this consolidation efficiently. Having a single "true" copy of the record means that frequent and costly reorganizations would be required to consolidate hot records in "hot pages". A more sustainable option is to organize records in some workload-independent way, such as having each page responsible for a specific key range at any time. Then, page caching could lead to significant cache under-utilization. In the worst case, there could only be a single few-byte hot record in each of several different pages. As such, caching at record granularity is preferable in an update-in-place design.

3.2 Page Grouping (Key Idea B)

Small Pages, Large Pages: Why Not Both?

From the perspectives of space amplification and I/O reduction, the ideal page is *as small as possible* (often 4 KiB in practice). Indeed, pages can be seen as simply an unfortunate artifact of current storage technology, with novel media like Optane persistent memory [33] pushing the boundary towards byte addressability.

However, small pages hurt range scans, which benefit from reading large amounts of contiguous data from storage. This is because small random reads still do not perform as well as sequential reads on NVMe SSDs; the best way to utilize the full read bandwidth is through large requests [29, 56], making large pages attractive.

To bridge this gap, we propose *page grouping*. The idea is that pages themselves are small but contiguous on the storage device, providing the potential for larger reads when needed. As a thought experiment, one could implement this idea for the entire key space, laying out the data consecutively in pages and keeping the page boundaries in memory. But this design will suffer in the face of inserts, requiring us to re-write (on average) half of the entire database whenever we insert a record.

Instead, we implement page grouping locally, by co-locating pages produced during each reorganization—creating *data segments*. Grouping also lets us reduce the number of entries we have to index in memory to only one entry per segment. We can achieve this by (i) fitting a piece-wise linear model over the keys and their positions in the data segment, and then (ii) using the linear components of the model to define and find the page boundaries. Each linear model indexes records belonging to one data segment and we place the records into the pages using the model to ensure that there is no indexing error (model-based inserts [1, 23]). This approach both (i) allows for long reads when needed, and (ii) leads to a compact in-memory index without sacrificing page-level access for point operations. The in-memory index only stores each segment's lower boundary, linear model, and location on disk; this information is enough to compute the correct page for any key.

Page grouping is governed by two parameters, *goal* and *epsilon*, which represent (i) the target fill rate of each page, and (ii) the maximum deviation that a model's prediction can have from a

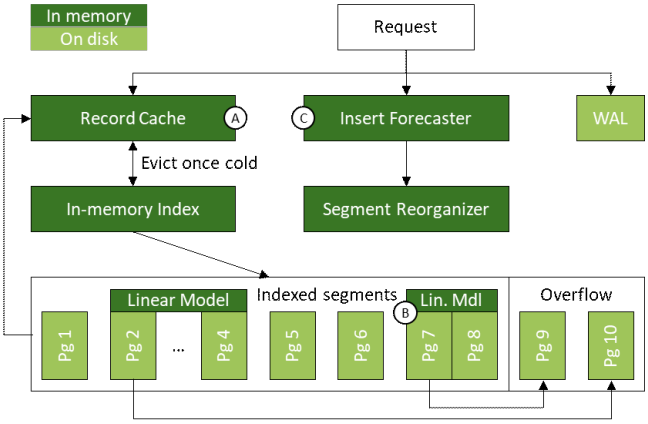


Figure 3: The design of TreeLine, highlighting our key ideas.

record's true position in the dataset. A user would typically set these parameters by first selecting goal (which impacts TreeLine's space amplification) and then maximizing epsilon, within the constraints of the maximum number of records that can fit in a page. Space for at least epsilon records must be left at each of the front and back of the page, to deal with model error near page boundaries.

For example, when using 64 byte records, each page fits up to 54 records after accounting for a 10-byte slot per record and the 89-byte page header (see Section 4.4). In our experiments, we find that setting goal to 44 and epsilon to 5 provided good performance. We study how these parameters affect the segments in Section 5.4.

3.3 Insert Forecasting (Key Idea C)

Half-Full Pages Are Usually Half-Empty.

Even with page grouping, requiring reorganization for each insert is impractical; data pages must contain empty space to absorb *some* inserts. This is common in tree data structures, which split a full node into two half-full ones. However, in a disk-based update-in-place design, empty space creates amplification during I/O.

Thus, we must leave empty space intelligently, by leveraging patterns in the inserts to generate forecasts about the future insert load. During reorganization, we then leave empty space according to this forecast, decreasing space amplification while still absorbing inserts in a high-performance manner. For this paper, we maintain an in-memory histogram to track the distribution of inserts and use it to forecast inserts for each part of the key space, but TreeLine is also compatible with more elaborate forecasting schemes.

4 TREELINE: IMPLEMENTATION DETAILS

In this section we describe TreeLine, which implements our three key ideas. As shown in Figure 3, TreeLine responds to the performance characteristics of modern storage media by using an update-in-place design, consisting of two main parts: (i) a *Tree*, an in-memory index to map reads and writes to pages, and (ii) a *Line* of variable-sized on-disk data segments, all on a single logical level. Section 4.1 describes each supported operation, while Sections 4.2-4.6 then introduce the individual components of TreeLine. Sections 4.7 and 4.8 deal with crash consistency and recovery, as well as thread synchronization, respectively.

4.1 Supported Operations

Lookups. Lookups first check the record cache (Section 4.2), which contains the most recent version of a record. If the key is not cached, the in-memory index (Section 4.3) is used to find the correct data segment, while the associated linear model (if any) locates the data page (Section 4.4). That page is brought into memory and searched. If the key is still not found, the page's overflow page (if any) is also brought into memory and searched. If there is no match in the overflow page either, TreeLine reports that the key was not found.

Data Modifications. Inserts, updates, and deletes initially create or update an entry in the record cache. Once the entry is later selected for eviction, TreeLine finds the corresponding data page using the in-memory index and segment linear model (if any), and brings it into memory to perform the operation, together with its overflow page (if any). If the base and overflow page are both full, reorganization is triggered (Sections 4.5 and 4.6).

Range Scans. Range scans proceed like lookups, but both the record cache and appropriate data page(s) (base and overflow) are always checked for keys within the specified range, which are then merged in key order. Records encountered in the record cache override records with the same key that might exist on a data page.

4.2 Record Cache (Key Idea A)

Cache Admittance. Whenever TreeLine admits a record into the record cache, it sets a priority level for the entry, from 0 to p_{max}. The priority is incremented whenever a cache entry is accessed.

We admit records into the record cache on three different occasions. First, any data modification request (insert, update, delete) by the user is cached with priority $p_{mid} = p_{max}/2$. If no entry with the same key is already present, TreeLine possibly evicts an entry to make space. Second, any lookup of a non-cached record will cache the record with priority p_{mid} after retrieving it from the appropriate data page. Third, we can choose to optimistically also cache additional entries from the same page with priority 1 whenever we cache a record through the lookup path.

Cache Eviction. TreeLine uses the clock algorithm to evict entries from a full cache: it cycles through cache entries until it finds an entry with priority 0, the *eviction candidate*, decrementing the priority level of each entry it encounters. If the eviction candidate is dirty, we continue advancing the "clock hand" up to 32 entries to find a clean candidate instead. We do this to prefer evicting entries that do not require I/O. If the final eviction candidate is dirty, we also write out all dirty cache entries that would go to the same page, but do not evict them. This helps amortize the eviction I/O costs. We have chosen this eviction algorithm because of its minimal synchronization overhead, which is crucial under high parallelism: just a single pointer needs to be atomically updated, compared to e.g., moving an arbitrary element to the back of an LRU queue.

4.3 In-Memory Index

The in-memory index is our map to the on-disk portion of the data. We employ the TLX `btree_map` [6]: a fast but traditional B+ tree.

Consulting the In-Memory Index. For both evictions of dirty cache entries and lookups/scans of non-cached keys, TreeLine needs

to retrieve the correct segment and page for keys. This is achieved through the in-memory index, which maps the lexicographically smallest key of each data segment to the appropriate (physical) segment identifier. This is sufficient because data segments cover mutually exclusive and collectively exhaustive key ranges. For multi-page segments, a compact linear model is also stored in the in-memory index in order to select the correct data page within the segment without needing additional index nodes.

Updating the In-Memory Index. TreeLine updates the in-memory index during reorganization (see Section 4.5). A read/write latch protects against concurrent in-memory index lookups. Although we have not seen a noticeable impact of this latch, one could use an opportunistic concurrency scheme [38] instead in future work.

4.4 Pages and Segments

Data Pages. TreeLine stores data in data pages. We adapt the physical page design from the implementation of `BTreeNode` in LeanStore [2, 36]. The page size is currently set to 4 KiB to match the page size of the underlying SSD. Each data page uses common prefix compression based on the lowest (inclusive) and highest (exclusive) key that it is responsible for, as defined at page creation time. It stores records in insertion order in the back of the page, while a sorted array of *slots* grows from the front of the page, with each slot pointing to a record; this maintains sorted access while reducing the copying overhead during data modification operations.

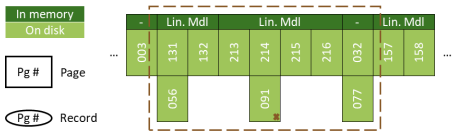
Data Segments. As introduced in Section 3.2, TreeLine employs page grouping to co-locate some data pages on disk, creating data segments ("ungrouped" data pages are "single-page data segments"). Data segments consist of pages with the layout described above, together with any possible overflow pages (see Section 4.5). The non-overflow (*base*) pages comprising a data segment are laid out logically contiguously on disk. For multi-page data segments, the in-memory index stores a linear model together with the segment identifier, letting TreeLine find the correct page within the segment for a given key. More details are covered in Section 4.5.

4.5 Supporting a Growing Database (Key Idea B)

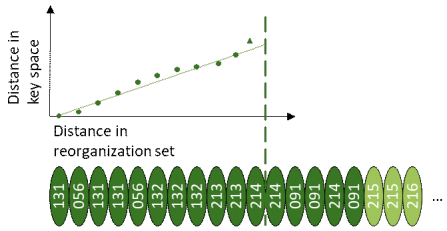
Overflow Pages. When a data page becomes full, TreeLine allocates an *overflow* page, which is not added to the in-memory index; it is accessed through the base page that overflowed. Overflow pages are laid out like base pages, and each inherit responsibility for the same key range as their base page. Each of the base and the overflow page remains sorted (using slots), but no guarantees are provided for the ordering of keys between these two pages.

Overflow pages help us collect more records for a given segment, to amortize the reorganization cost. Once an overflow page also fills up, the entire segment involved, called the *full segment*, will be reorganized. We do not allow for more than one overflow page per base page to bound the cost of lookups and data modifications.

Page Orderings. For clarity, we distinguish three types of page orderings. Our data pages match the page size of the underlying SSD (4 KiB), so these orderings apply to pages in both the "TreeLine" and the "SSD" sense. The *physical order* refers to the actual page locations on the SSD. It is not exposed, since the drive itself abstracts it away to balance device wear through out-of-place writes. Instead,



(a) Page 091 - the overflow for 214 - is full, causing reorganization. Phase I detects more segments with reorganization potential.



(b) Phase II fits a linear model to the reorganization set. Once the error threshold is reached, records are forwarded to Phase III.

(c) Phase III writes a new segment out. Once all new segments have been written, Phase IV updates the in-memory index.

Figure 4: An example reorganization.

the drive exposes a *logical order* of pages, implemented in the device controller. In TreeLine, each base data page contains keys for a disjoint region of the key space. Based on the lower boundaries of these regions, we also get the *key order* of pages.

Reorganization. We now move on to the reorganization process, which is divided into four phases. *Range detection* is followed by one or more iterations of *model building* and *segment write-out*, followed by the *index update*. We provide an example in Figure 4.

Phase I: Range Detection. Only considering records from the full segment can lead to sub-optimal data layouts. Instead, TreeLine also looks "around" the full segment S for segments with *reorganization potential*: segments with at least one base page with a (not necessarily full) overflow page. It examines neighboring segments in key order, up to a distance of r (currently set to 5) segments from S in each direction. This process results in a set of up to $2r + 1$ segments to be reorganized, which we call *pre* segments, which are contiguous in key order and include S. In Figure 4a, Phase I finds two neighboring segments with reorganization potential around S.

Phase II: Model Building. This phase operates on the sorted set of all records in the pre segments, called the *reorganization set*, if it is non-empty. It is visualized as the collection of ovals in Figure 4b.

TreeLine considers records from the reorganization set in key order and tries to place them into as-large-as-possible newly-created segments, which we call *post* segments. Each page, together with its associated overflow page, is brought into memory as needed and kept there until all of its records have been written into new segments. In Figure 4b, Pages 131, 056, 132, 213, 214 and 091 have been brought into memory. We call the smallest key (in key order) of the reorganization set the *reference key*, which will be the smallest key in the first page of the new segment being built.

While considering each record, TreeLine builds a linear model (light green line in Figure 4b) using the PGM index's [24] optimal piecewise linear regression algorithm [54]. We chose this algorithm because it produces the optimal (fewest) segments in one pass over the data [24, 54]. This model will determine the correct page for a record within the segment, based on the record's key and the segment's reference key. It relates the distance between the record's key and the reference key **in key space** to their distance **within the reorganization set**. For example, if the keys in the reorganization set are {"a", "c", "f"}, the reference key is "a" and the distance of "f" from "a" is 5 in key space, but 2 in the set.

A linear model is space-efficient and reasonably accurate at the granularity of the reorganization set, but it will still accumulate error. We continue processing successive records, until we hit a configurable error threshold *epsilon*, indicated by the dashed dark green line in Figure 4b. Setting epsilon reflects a trade-off. A large epsilon will let us process more records, but the resulting model will be less accurate. To compensate, we will need to place fewer records per page, increasing space amplification. On the other hand, a small epsilon will only let us process fewer records at a time, leading to smaller segments and eroding the benefits of page grouping.

Once we reach the error threshold, the current iteration of Phase II ends. Any processed records are sent to Phase III as a *write-out set* and removed from the reorganization set, before repeating Phase II.

An iteration of Phase II is also terminated early in three cases. One is whenever the contents of the reorganization set fit on a single page. No model is needed for a single-page segment, so we directly forward all the records to Phase III. The second case is whenever we exceed the memory budget set aside for reorganization; remember that each page with processed records is kept in-memory during model building. In this case, we stop and forward the records and model to Phase III, even if the model error was still below epsilon. The third case is when we have processed enough records (16 * goal) to fit a 16 page segment without exceeding epsilon.

Phase III: Segment Write-out. Given a write-out set and a model from Phase II, Phase III materializes the post segment. As shown in Figure 4c, the model is used to place the records in a number of logically contiguous pages on the SSD, either freshly allocated or retrieved from a free list (see Phase IV). The number of pages is determined by dividing the size of the write-out set by the goal parameter. However, to simplify file management, TreeLine only creates fixed-size segments containing 1, 2, 4, 8, or 16 pages. So, if the proposed post segment comes out to e.g., 10 pages, TreeLine will only create an 8 page post segment with *goal* records per page, returning the leftover records to the reorganization set for processing in the following iteration of Phase II. Phase III also generates a unique non-zero *production ID* associated with this segment and includes it in the first page of the post segment.

Phase IV: Index Update. TreeLine concludes the reorganization process by updating the in-memory index. The entries for the pre segments are deleted, while entries with the reference key, model (if any) and segment identifiers of the post segments are inserted. The pre segments are then invalidated by overwriting their production ID with the value zero, which aids recovery (see Section 4.7). They are then added to a free list, from where they can be retrieved by a future reorganization to be overwritten with its post segments.

4.6 Insert Forecasting (Key Idea C)

Tracking Inserts. We base our insert forecasting on statistics we collect from tracking inserts. We divide the workload into *epochs*, which represent a certain number of workload inserts (e.g., 100,000). Over the course of an epoch, we build an equi-depth histogram with b partitions that captures the distribution of inserts. For each insert, we increment the counter of the histogram bin corresponding to the key (an $O(\log b)$ operation). To reduce tracking overhead, we could sample inserts, but we find that the overhead is less than 100 ns per insert, which is negligible for disk-based systems. Once an epoch ends, we "freeze" its histogram and use it for forecasting the distribution of future inserts, discarding any older histograms. This way, there are always two histograms: one being built based on the current epoch and one "frozen" from the last epoch. Besides the epoch size, we need to configure the parameter b. In the extreme case, each partition would correspond to one data page. However, we find a more coarse-grained partitioning to be sufficient for our workloads. Since we use an equi-depth histogram (as opposed to equi-width), we need to set the partition boundaries. In theory, we could use the full data for this purpose, but this would cause I/O. Instead, we maintain an in-memory *reservoir sample* [41] of the inserts. At every point in time, that sample represents a uniform random sample of the base data. When creating a new histogram, we sort the in-memory sample and use it to determine *approximate* partition boundaries. By default, we use a sample size of $10 * b$.

Generating Forecasts. To forecast inserts for a segment, we use its boundaries to query the "frozen" histogram and sum the counters of intersecting partitions. For partitions that partially overlap the query range, we use linear interpolation. We then forecast inserts for the f epochs following the epoch of the "frozen" histogram.

Utilizing Forecasts. During reorganization, we use insert forecasting to determine the empty space in each page, by setting the goal parameter. Using a pessimistic estimate of the number of keys in the current segment, together with the insert forecast, we can determine how many pages will be needed. We then set the goal so as to distribute the current records evenly across these pages.

Interaction with Page Grouping. Both insert forecasting and page grouping affect the physical page layout during reorganizations. The forecasted inserts for a particular segment influence the goal parameter, which in turn affects the epsilon parameter, as per Section 3.2, since the maximum number of records per page must not be exceeded. Finally, both goal and epsilon impact the linear model construction, by influencing the transition to Phase III and the post segment size (see Section 4.5).

4.7 Crash Consistency & Recovery

Crash Consistency. The on-disk information should enable TreeLine to recover to a consistent state after a crash. Data modification operations only touch a single SSD page, for which the SSD guarantees atomic writes. However, extra care is needed for the two processes that edit multiple pages at a time: overflow page allocation and reorganization. When allocating an overflow page, we first write it to disk, before providing the corresponding base page with the overflow page identifier and writing out the base page as well. We use **Check I** below to recover from crashes between

Table 1: Segment (left) and page (right) lock compatibility.

	IR	IW	O	OX
IR	**Y**	**Y**	**Y**	N
IW	**Y**	**Y**	N	N
O	**Y**	N	N	N
OX	N	N	N	N

	S	X
S	**Y**	N
X	N	N

these two writes. For reorganization, we detect and repair crashes when only some of the post segments have been written out using **Check II** below. In Phase I, a START REORG record is added to the write-ahead log, including a new globally unique production ID and a list of the pre segments and their current production IDs. The new production ID will also be stored on the post segments resulting from this reorganization. In Phase IV, an END REORG record is appended to the write-ahead log, with the same production ID.

Recovery. We now present a sketch of recovery based on the above scheme. We leave the implementation of this sketch for future work.

Recovery aims to rebuild the in-memory index based on the on-disk data pages. It scans data segments in logical order and inserts the reference key of each into the in-memory index. Any overflow pages are deferred to be examined again in the end of the scan, by which point the corresponding base page should be already indexed. We then perform **Check I**: we use the lower bound of each overflow page to find the correct base page through the in-memory index, and ensure the base page points to the overflow page.

Before the data page scan, TreeLine also performs **Check II**: it checks the write-ahead log for any START REORG records without an END REORG record with the same production ID. Among such in-flight reorganizations, there are two cases. If there is at least one pre segment on disk with a production ID not matching the one logged in the START REORG message, Phase IV must have been reached before crashing. We then treat the reorganization as successful, invalidate any pre segments that still have the production IDs logged in the START REORG message, and treat the post segments as valid when we encounter them during the page scan.

However, all pre segments might still match the production IDs in the START REORG record. In this case, we deem the reorganization failed and treat the pre segments as valid during the page scan, include them in the in-memory index and re-attempt the reorganization after recovery. At the same time, we identify any post segments we encounter during the page scan using the production ID of the START REORG record, and treat them as invalid.

Note that, for each given crash, there could be at most as many concurrent reorganizations as the number of user threads, providing an upper bound on the amount of work needed during recovery.

4.8 Thread Synchronization

TreeLine uses two types of locks: segment locks and page locks. Table 1 provides their compatibility matrices. Segment locks can be acquired in **I**ntention **R**ead, **I**ntention **W**rite, re**O**rganization or re**O**rganization e**X**clusive mode. Page locks can be acquired in **S**hared or e**X**clusive mode. One page lock protects both a base page and its overflow page. We will now explain our locking strategy.

Lookups. TreeLine first locks the appropriate segment in IR mode. It then locks the appropriate page in S mode and performs the lookup. The page lock is then released before the segment lock.

Data Modifications. TreeLine first locks the appropriate segment in IW mode. It then locks the appropriate page in X mode and performs the operation. The page lock is then released before the segment lock. If reorganization is triggered, locks are released and TreeLine follows the reorganization path below.

Range Scans. TreeLine first locks the appropriate segment in IR mode. It then locks each page in the segment in S mode as the scan proceeds, releasing each page lock as soon as the page has been scanned. If the segment is exhausted and the scan is not finished, TreeLine uses lock coupling to avoid a reorganization from intervening: it firsts acquires an IR lock on the next segment before releasing the IR lock on the previous one.

Reorganization. After Phase I, reorganization locks all the pre segments in key order in O mode. This mode lets reads and scans still use the pre segments while the post segments are being created. At the start of Phase IV, TreeLine upgrades to an OX lock, implicitly also waiting for anyone accessing the pre segments to finish their reads. Once the in-memory index has been updated and at least one pre segment has been invalidated, the OX lock is released.

5 EVALUATION

In this work, we present three techniques that mitigate the traditional drawbacks of update-in-place designs, which we implement in TreeLine. As a result, the goal of our evaluation is to examine the effectiveness of these techniques in comparison to (i) LSM-based systems, and (ii) other update-in-place systems. To that end, we aim to answer the following questions:

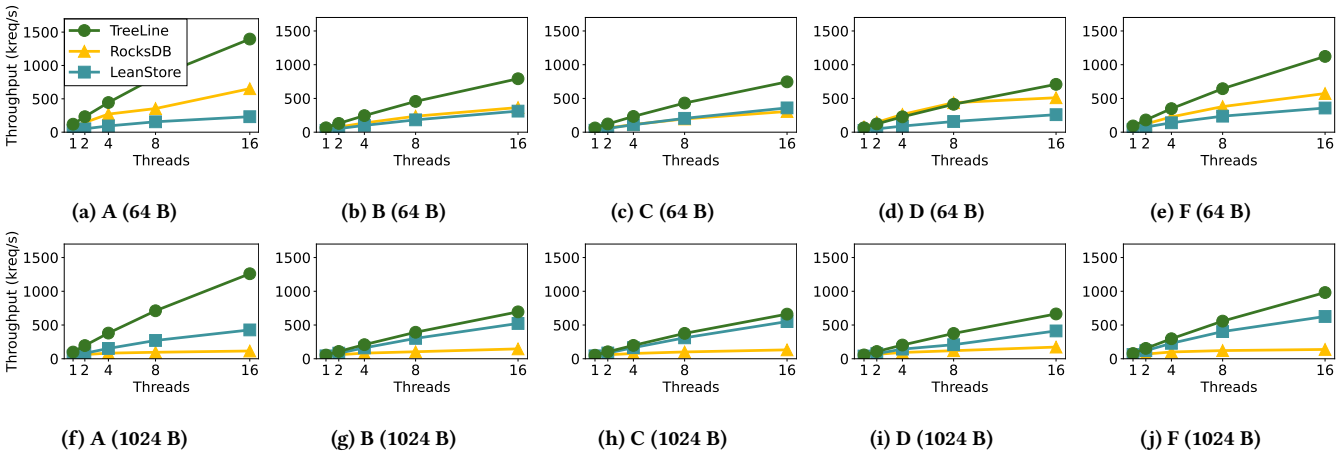
- How does TreeLine compare against RocksDB [48] (an LSM-based key-value store) and LeanStore [36] (a state-of-the-art update-in-place key-value store) on throughput and the amount of physical I/O performed? (Section 5.2)
- How do the record cache and page grouping contribute to TreeLine's overall performance? (Section 5.3)
- How does the choice of the page grouping parameters (goal and epsilon) affect the grouping "effectiveness"? (Section 5.4)
- How effective is insert forecasting? (Section 5.5)

We find that TreeLine outperforms RocksDB by 2.20× and LeanStore by 2.07× on average across our 1024 byte YCSB point workloads. With 16 request threads, TreeLine outperforms RocksDB and LeanStore by 2.50× and 2.80× respectively on uniformly distributed scan-heavy workloads (YCSB E), averaged across three datasets.

5.1 Experimental Setup

Hardware and Environment. We use a machine equipped with a 20-core 2.10 GHz Intel Xeon Gold 6230 CPU [32] and 128 GiB of memory, running Linux 5.12.5. We use a 1 TB Intel DC P4510 NVMe SSD [31] and the ext4 file system for all our experiments.

Baselines. We compare TreeLine against RocksDB [48] (an LSM key-value store) and LeanStore [36] (a state-of-the-art update-in-place key-value store). We use RocksDB version 6.14.6 [46] and LeanStore at commit `d3d8314` [2]. For a fair comparison, we disable block compression and SSTable checksums in RocksDB, since these features are not present in TreeLine. We configure RocksDB and LeanStore to use 4 KiB blocks to match TreeLine's pages. On RocksDB, we enable both Bloom filters and prefix Bloom filters [49]



(a) A (64 B) (b) B (64 B) (c) C (64 B) (d) D (64 B) (e) F (64 B)

(f) A (1024 B) (g) B (1024 B) (h) C (1024 B) (i) D (1024 B) (j) F (1024 B)

Figure 5: Zipfian YCSB point workloads on the Amazon dataset. The first (second) row shows results for 64 (1024) byte records.

Table 2: A description of the YCSB workloads.

Workload	Description
A	50% Read, 50% Update
B	95% Read, 5% Update
C	100% Read
D	95% Read Latest, 5% Insert
E	95% Range Scan (average length 50, maximum length 100), 5% Insert
F	50% Read-Modify-Write, 50% Read

Table 3: Our Amazon dataset system configurations.

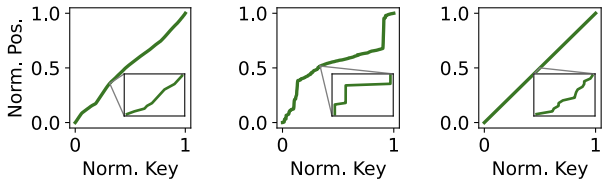
Config.	System	Setup Details
64 B	TreeLine	44/5 pg. grp. goal/epsilon, 683 MiB rec. cache
	RocksDB	107 MiB memtables (×2), 469 MiB block cache
	LeanStore	683 MiB buffer pool
1024 B	TreeLine	2/0.5 pg. grp. goal/epsilon, 10903 MiB rec. cache
	RocksDB	1715 MiB memtables (×2), 7473 MiB block cache
	LeanStore	10903 MiB buffer pool

with 10 bits and a prefix length of 3 respectively (the recommended defaults). We also disable write-ahead logging on all three systems, to distinguish the performance of TreeLine, RocksDB, and LeanStore on write-heavy workloads from the logging overhead.

Workloads. We use our own C++ implementation of the Yahoo! Cloud Serving Benchmark (YCSB) [17] to perform our evaluation. We provide a description of the YCSB workloads in Table 2.

System Configurations. We use 64 byte records *(64 B)* (8 byte key, 56 byte value) and 1024 byte records *(1024 B)* (8 byte key, 1016 byte value). Depending on the dataset, we give each system enough memory to store up to 33% of the dataset in memory. Table 3 describes how this memory is used for the Amazon dataset; it also lists the page grouping parameters used. We give TreeLine, RocksDB, and LeanStore access to 4 background threads.

Datasets. We evaluate against three datasets (one synthetic, two real-world): (i) a synthetic dataset of uniformly distributed keys (20 million keys), (ii) an Amazon reviews dataset (33 million keys), and (iii) an Open Street Maps (OSM) dataset (23 million keys) [18]. We plot their key cumulative distribution functions (CDFs) in Figure 6.



(a) Amazon (b) OSM (c) Synthetic

Figure 6: Our datasets' key CDFs.

Checkpoints. To ensure consistent results, we start each experiment from the same database checkpoint, created for each system as follows. We first load each database with the dataset being tested. Then, we run a 40 million uniform update workload. The purpose of this update workload is to add levels to RocksDB's LSM tree to more closely resemble an LSM that has "been used". Finally, we persist the databases; their on-disk images become the "checkpoints" we use. We run each experiment against a copy of these checkpoints.

Metrics. We primarily measure throughput during the workload and report it as thousands of requests processed per second (kreq/s). Using iostat [26], we also measure the amount of physical I/O and the physical I/O throughput observed by the operating system. We report TreeLine's throughput relative to RocksDB and LeanStore using speedups. We compute averages by taking a geometric mean.

5.2 End-to-End Performance

5.2.1 Skewed Point Workloads. We first run the YCSB point workloads with Zipfian-distributed requests ($\theta = 0.99$). Figure 5 shows TreeLine's, RocksDB's, and LeanStore's throughputs. Since our conclusions are similar across datasets, we only show and discuss our results for the Amazon dataset as it is the largest (33 million records). TreeLine outperforms RocksDB (LeanStore) by 1.62× (2.81×) and 2.99× (1.53×) on average for the 64 B and 1024 B configurations respectively. From these results, we draw three conclusions.

TreeLine outperforms RocksDB and LeanStore on skewed update-heavy workloads because it (i) leverages its record

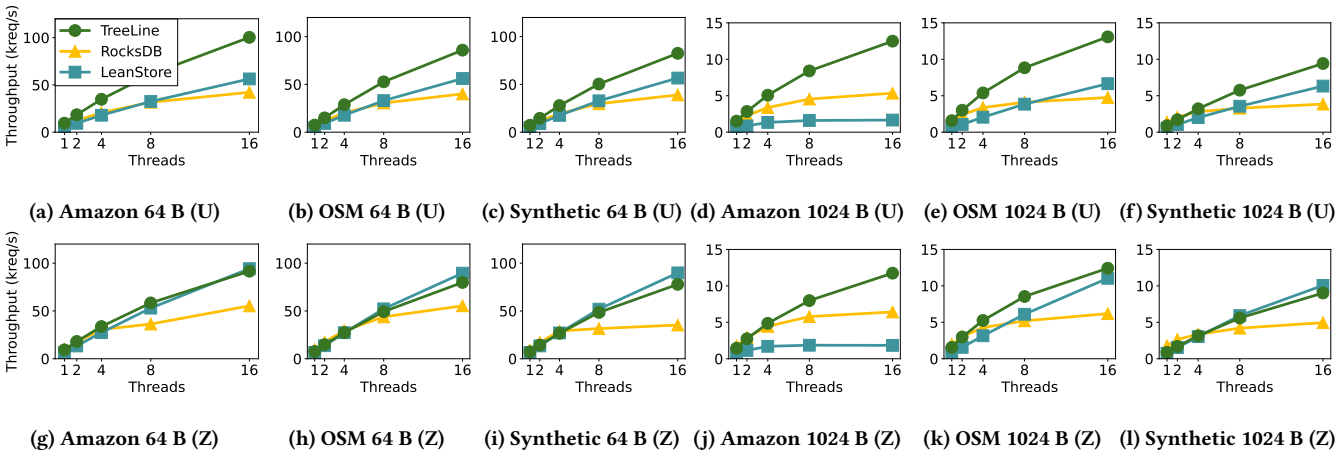


Figure 7: Scan-heavy (YCSB E) workloads on our three datasets for 64 byte records and 1024 byte records. The first (second) row shows uniformly (Zipfian) distributed request results.

cache to reduce write amplification while also (ii) providing efficient reads. We observe that on workload A (64 B), TreeLine writes 3.09 GiB of physical data while RocksDB and LeanStore write 4.27 GiB and 23.1 GiB respectively. TreeLine uses its record cache to reduce the amount of physical writes it makes to an amount that is less than that of RocksDB, thereby mitigating its potentially high write amplification. LeanStore writes significantly more than TreeLine and RocksDB because it uses a page cache and (i) the YCSB workloads have *key skew* (hot records are not necessarily clustered on the same pages), (ii) the records are small relative to the page (64 byte records, 4 KiB pages), and (iii) the pages with hot keys do not all fit in memory. On the same workload, TreeLine reads 12 GiB of data while RocksDB reads 27 GiB (2.3× more). RocksDB reads more data than TreeLine because it also runs compaction jobs in the background. LeanStore reads 42.2 GiB of data (3.5× more than TreeLine) because record updates require a page read-modify-write.

The reason why TreeLine and RocksDB achieve a higher throughput on update-heavy workloads, compared to read-only workloads, is that caching updates allows reads of recent updates to be served from the cache without I/O. In contrast, for read-only workloads, caching a record for the first time requires I/O on the critical path.

When TreeLine outperforms RocksDB on read-heavy workloads, it is because it performs fewer physical reads. On workloads B and C (64 B), TreeLine reads 19 GiB, while RocksDB reads 31.55 GiB and 65.35 GiB respectively (1.6× and 3.5× more). For 1024 B, RocksDB reads 5.4× and 6.8× more data than TreeLine on workloads B and C respectively. There are two reasons for these differences in physical reads. First, as described above, RocksDB launches compaction jobs in the background to merge obsolete records; this work affects workload B because it contains updates. Second, RocksDB has SSTables present on multiple levels. Hence, a point read may need to query multiple SSTables, even with Bloom filters enabled. In contrast, point reads on TreeLine only ever require up to two page reads (base and overflow).

TreeLine surpasses LeanStore on the 64 B read-heavy workloads because of record caching. By design, reads are efficient in both TreeLine and LeanStore because they are update-in-place key-value stores. However, as mentioned previously, the YCSB workloads exhibit key skew. Key skew limits LeanStore's buffer pool's effectiveness when records are small relative to the page (e.g., 64 byte records). For example on workload C, LeanStore reads 42.2 GiB of data whereas TreeLine reads 18.8 GiB of data.

Table 4: Read performance statistics on our Amazon uniform scan-heavy workload (YCSB E) with 16 request threads.

Config.	Phys. Reads	Phys. Read Thpt.	Req. Thpt.
TreeLine 64 B	**13.4 GiB**	550 MiB/s	**100 kreq/s**
RocksDB 64 B	31.1 GiB	**797 MiB/s**	42 kreq/s
LeanStore 64 B	17.0 GiB	581 MiB/s	56 kreq/s
TreeLine 1024 B	**75.9 GiB**	**1079 MiB/s**	**12 kreq/s**
RocksDB 1024 B	147 GiB	958 MiB/s	5.4 kreq/s
LeanStore 1024 B	76.4 GiB	155 MiB/s	1.7 kreq/s

5.2.2 Scan-Heavy Workloads. Next, we examine TreeLine's performance on a scan-heavy workload (YCSB E), where page grouping influences performance. Since the effectiveness of page grouping depends on the dataset (see Section 5.4), we present results for all three of our datasets. For RocksDB, we also enable prefix Bloom filters, which help reduce the number of SSTables read during a scan, making RocksDB a strong baseline. We otherwise use the same experimental setup as our skewed point workload experiments.

Figure 7 shows our results for uniform and Zipfian-distributed scans. With enough parallelism (i.e., 16 request threads), TreeLine achieves average speedups of 1.74× (2.21×) and 1.88× (2.50×) over RocksDB for Zipfian (uniformly) distributed requests against the 64 B and 1024 B configurations respectively. When compared against LeanStore, TreeLine achieves average speedups of 0.91× (1.58×) and 1.86× (2.80×) for Zipfian (uniformly) distributed requests against the 64 B and 1024 B configurations respectively. LeanStore outperforms TreeLine in certain cases due to caching effects, discussed below. From these results, we draw three key conclusions.

TreeLine's speedup over RocksDB on the scan-heavy workloads comes from reading less data from disk, because of

its update-in-place design. Table 4 lists each system's (i) physical reads, (ii) physical read throughput, and (iii) workload request throughput for a uniform scan-heavy workload on the Amazon dataset with 16 request threads. The trends are similar across the other datasets. In both configurations shown in the table, TreeLine reads at least 1.9× less data from disk than RocksDB. This difference leads to TreeLine's throughput advantage despite it having a lower physical read throughput when scanning 64 byte records.

RocksDB reads more data from disk for two reasons. First, there may exist SSTables on multiple levels that overlap the scan range. RocksDB needs to examine all overlapping SSTables (not excluded by the prefix Bloom filters) to return the correct records. Second, RocksDB still runs compaction jobs, because the checkpoints we use contain recent updates (see Section 5.1) and RocksDB always launches a compaction job after starting up.

TreeLine has a lower physical read throughput than RocksDB when scanning 64 B records because of its I/O pattern. Although page grouping enables sequentially reading all the pages in a segment, its effectiveness ultimately depends on the dataset and record size (see Section 5.4). Thus, TreeLine's scans also contain random reads, whereas RocksDB's scans translate to long sequential reads.

TreeLine's scan performance is not significantly affected by request skew. TreeLine's throughput advantage over RocksDB and LeanStore is higher on uniform scan-heavy workloads because RocksDB's block cache and LeanStore's buffer pool become less effective. TreeLine cannot use its record cache to avoid I/O during a scan because it only caches individual records; it must always check the on-disk pages for the records that lie in the scanned ranges. RocksDB caches blocks and LeanStore caches pages, which allows them to avoid I/O when a block or page involved in the scan is already cached in memory. For workloads with long scans that exceed the cache's capacity (e.g., Extract-Transform-Loads [10]), we expect RocksDB's and LeanStore's caches to be less effective.

TreeLine's speedups over LeanStore on the 1024 B configurations are due to page grouping. As shown in Table 4, LeanStore and TreeLine read a comparable amount of physical data from disk. However, on the 1024 B configuration, TreeLine achieves a 7× higher physical read throughput, again due to the I/O request pattern. LeanStore scans require scattered 4 KiB page reads whereas TreeLine can read longer segments from disk due to page grouping.

5.2.3 Uniform Point Workloads. We also run our point workloads with a uniform request distribution. TreeLine achieves an average throughput speedup of 1.45× and 1.32× over RocksDB and LeanStore respectively on our 64 B and 1024 B configurations. On read-heavy uniform point workloads (B, C), TreeLine achieves an average throughput speedup over RocksDB (LeanStore) of 1.36× (1.11×); on update-heavy uniform point workloads (A, F), TreeLine has an average speedup over RocksDB (LeanStore) of 1.55× (1.57×).

The conclusions we draw about skewed point workloads in Section 5.2.1 hold for uniform point workloads as well, with one exception. The record cache is not as effective at decreasing write amplification in uniform update-heavy workloads. Such workloads, especially when the record size is much smaller than the page size, are fundamentally challenging for larger-than-memory update-in-place systems for two reasons. First, write buffering (e.g., with a cache) has a limited impact on reducing the number of page reads

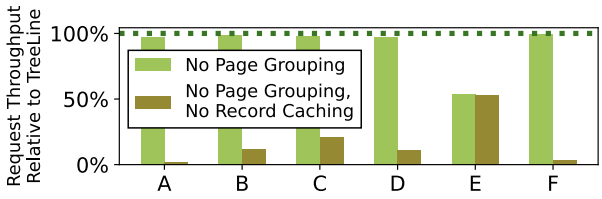


Figure 8: Impact of page grouping and record caching on the Zipfian YCSB workloads (Amazon dataset, 1024 B configuration, 16 request threads).

and writes in uniform workloads since there is no skew. Second, updates of a single record require reading and writing a entire page from disk—leading to high write amplification. TreeLine does not optimize for uniform update-heavy workloads, since skewed point workloads are more common in real-world scenarios [10].

5.3 TreeLine Factor Analysis

We study the impact of TreeLine's record cache and page grouping with a factor analysis, using the Amazon dataset and 1024 byte records. We run each of the Zipfian YCSB workloads with 16 request threads. Figure 8 shows our results, where we successively disable page grouping and then both page grouping and record caching. We report each configuration's throughput relative to that achieved by the full system. From these results, we draw three key conclusions.

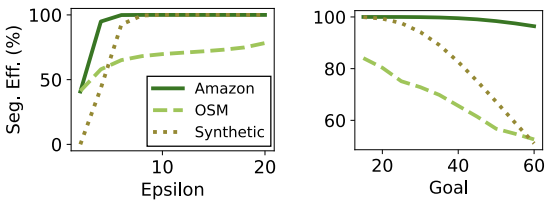
Record caching helps reduce read and write amplification in the read/update workload. TreeLine's record cache reduces the total amount of physical reads (writes) by an average of 3.11 × (2.60×), leading to an average throughput speedup of 8.7×. The workloads are highly skewed, so caching hot records reduces the necessary SSD accesses, improving throughput.

Page grouping does *not* negatively affect the point workloads. Even after enabling page grouping, TreeLine maintains its throughput on the point workloads A-D and F. Recall that TreeLine's in-memory index only stores the (i) key boundaries, (ii) the physical locations of the beginning of each segment (which can consist of multiple pages), and (iii) linear models that map records to pages within segments. Yet, TreeLine does not need to read in an entire multi-page segment to access a single record for point reads and updates; TreeLine can still operate at page granularity by using the linear model to find the page for a given record.

Page grouping accelerates scans by enabling longer physically contiguous reads. On the scan-heavy workload E, TreeLine's throughput increases by 1.87× once page grouping is enabled despite there being no significant change in the *amount* of physical reads. The reason for this improvement is that TreeLine achieves an increased physical read *throughput* by 1.87×, by making more sequential reads. Without page grouping, scans consist of scattered 4 KiB reads, which leads to lower physical read throughput.

5.4 Page Grouping Effectiveness

TreeLine uses page grouping (i) to boost scan performance and (ii) to shrink the in-memory index. In this section we (i) study how the dataset, goal, and epsilon affect page grouping; and (ii) analyze page grouping effectiveness in our configurations from Section 5.2.



(a) Goal = 44 (b) Epsilon = 5

Figure 9: The percentage of pages in multi-page segments as we fix "goal" and vary "epsilon" and vice-versa.

5.4.1 Page Grouping Sensitivity Study. We measure *segment efficiency* (the percentage of pages that are in multi-page segments) across our datasets; a higher segment efficiency is better. In Figure 9a, we fix goal and vary epsilon. In Figure 9b, we fix epsilon and vary goal. From these results, we draw three conclusions.

Increasing epsilon improves segment efficiency. Epsilon represents the error tolerance when building a page grouping linear model. Having a larger error tolerance translates to being able to fit a model through *more* records. This tolerance translates to a greater likelihood of creating a multi-page segment (for a fixed goal value).

Increasing goal decreases segment efficiency. Goal is the desired number of records to place on a page. Recall that page grouping uses fixed segment sizes (1, 2, 4, 8, and 16 pages). Thus, choosing a small goal decreases the maximum number of records that can be placed onto a segment (for a fixed epsilon). This is why Figure 9b shows a decreasing trend as goal increases: fitting a model through more records with a fixed epsilon becomes more difficult.

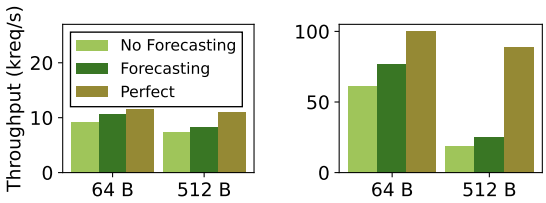
The dataset affects segment efficiency. Page grouping essentially fits linear models over a dataset's CDF. Intuitively, for a fixed epsilon, one needs fewer linear models for a CDF with many "linear regions". The Amazon and OSM datasets enable higher segment efficiencies, because their CDFs have many "linear regions" (see Figure 6). Interestingly, our uniform synthetic dataset can produce a lower efficiency, because its CDF is locally "bumpier".

5.4.2 Index Entries Reduction. With page grouping, TreeLine only indexes the boundaries of each segment (and the linear model), instead of each page boundary. Reducing the number of index entries has two benefits: (i) we save memory for other uses, and (ii) index operations are faster because the index is logically smaller.

On our 1024 B (64 B) workloads, page grouping reduces the number of entries in the index by 3.8× (4.8×), 4.5× (1.9×), and 1.8× (1.8×) for the Amazon, OSM, and synthetic datasets respectively. The reduction depends on the segment distribution; having more pages in larger segments means that fewer index entries are needed.

5.5 Insert Forecasting

Finally, we evaluate insert forecasting. Among the YCSB workloads, only D and E perform inserts, and only for 5% of the operations. To make the impact of insert forecasting more visible, we instead run a workload consisting of 50% reads/50% *inserts*. We compare the performance of TreeLine with insert forecasting enabled with the performance of TreeLine without insert forecasting, as well



(a) 1 application thread (b) 8 application threads

Figure 10: The performance of insert forecasting.

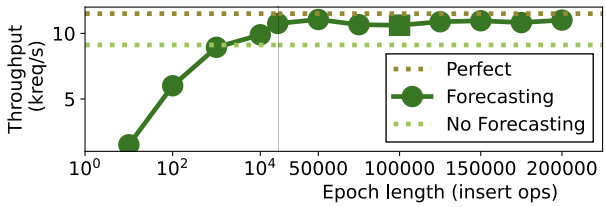


Figure 11: Insert forecasting performance across epoch lengths. A square marks the epoch length used in Figure 10a.

as with the performance of TreeLine given perfect information about the future stream of inserts, thus avoiding reorganizations. The forecasting epoch length is set to 100,000, we use $b = 20,000$ histogram partitions and forecast inserts for $f = 100$ future epochs.

For our experiments we use a dataset of taxi pickups in New York City, where we have inlined the pickup coordinates into an 8-byte key using the S2 geometry library [28]. This dataset has three interesting properties regarding hot keys: (i) they exist (popular taxi pickup areas), (ii) they are not necessarily co-located in the key space and (iii) they change over time (e.g., due to the time of day).

As shown in Figure 10, insert forecasting is mainly effective in the 64 B case, closing on average more than half of the gap between no forecasting and perfect forecasting and reducing reorganizations by an average of 63% (not plotted). With smaller records, the absolute number of inserts that a 4 KiB page can accommodate is higher. This means that insert forecasting can estimate and leave appropriate free space at a finer granularity, delaying reorganization for longer. On average, insert forecasting improves the throughput of TreeLine by 1.22×, reducing reorganizations by an average of 41%.

We also explore the sensitivity of insert forecasting to the epoch length for the 64 B configuration using 1 application thread, scaling b to keep it at 20% of the epoch length. As shown in Figure 11, very short epochs cannot justify their overhead, but even an epoch length of 10,000, 10% of the value in Figure 10a, delivers performance benefits. Thus, although the epoch length affects performance, coarse-grained tuning is enough to see improvements.

6 DISCUSSION

TreeLine bridges update-in-place and LSM-tree designs. Still, it has some limitations, discussed below alongside possible solutions.

Variable-Sized Records. Our current design assumes fixed-sized records. While the data pages support variable-sized records, we train linear models on record *positions* and divide by the maximum

records per page to map records to pages. Instead, we could train the model using record *offsets*. By setting epsilon accordingly, the model would still determine the correct base page for a key.

Non-Integer Keys. To handle string keys, we could follow a similar approach as prior work [58]: recursively build a radix tree of spline models with a node per 8-byte key prefix, until the model satisfies a pre-determined maximum prediction error. A similar approach is taken by the Adaptive Radix Tree (ART) [37], recursing until two keys can be differentiated, without storing the whole strings.

More Sophisticated Forecasts. To forecast inserts, we project the insert distribution of the previous epoch. This approach works well when distribution shifts happen smoothly, as is the case in the taxi dataset [55]. For more sophisticated insert distributions found in production workloads [10], we could use a timeseries forecasting module like Prophet [59]. We found Prophet efficient enough to perform the forecasting in the background, but its forecasts were comparable to those of our approach for the taxi dataset.

Pure Write Workloads. TreeLine currently cannot compete with RocksDB on pure write workloads. Even with perfect forecasting, inserts in TreeLine still suffer from its update-in-place paradigm, while RocksDB can simply write out its memtable without a prior read. To match RocksDB on pure write workloads, we could use a hybrid approach that stages inserts during burst periods in a log-structured file which is gradually merged during idle periods.

7 RELATED WORK

LSM Trees. LevelDB [27] pioneered the partitioned leveling merge policy [43], in which each level (except L0) contains multiple range-partitioned fixed-size files to improve concurrency. Building on LevelDB, RocksDB [48] provides more compaction policies and a merge operation to reduce contention with user requests [43]. WiscKey [42] extends LevelDB by storing values in a separate values log. This design reduces write amplification by avoiding unnecessarily copying values during the merge. TreeLine, in contrast, only merges a page with its one overflow page. It is hence unclear whether introducing key-value separation in TreeLine would yield benefits, considering the added cost of garbage collection.

Modern LSM-trees use Bloom filters to reduce disk accesses. Dayan et al. [19] show how to best allocate bits to each filter to improve the space-accuracy trade off. SlimDB [57] and Chucky [21] instead use a multi-level Cuckoo filter, improving pruning power.

Update-in-Place Designs. We are not first to revisit update-in-place designs for modern hardware. LeanStore [36] bridges the gap between in-memory and disk-based systems through a low overhead buffer manager. While efficient when the working set fits into memory, it is not optimized for the out-of-memory case, which we target. FASTER [11, 12] is another key-value store with in-place updates. Compared to TreeLine, FASTER is optimized for point accesses and does not support efficient range scans. KVell [39] is a key-value store with a similar design to ours. It uses an in-memory B+ tree to index the unsorted on-disk keys. Its drawbacks are that (i) it needs to index all keys (which can have a high memory overhead), and (ii) scans incur random I/O. In contrast, with page grouping, TreeLine only indexes segment boundaries (i.e., fewer index entries) and reads full segments sequentially when running range scans.

Learned Indexing. Learned indexes [34, 35, 44] build a model over sorted data to predict the position of a key. Abu-Libdeh et al. [1] integrate learned indexes into Bigtable [13] and show that the index's reduced size improves cache efficiency [1]. Like TreeLine's page grouping, FITing-Tree [25] also fits a piece-wise linear model over its data. However, FITing-Tree (i) is an in-memory data structure, and (ii) uses its error bound to bound lookup time. In contrast, TreeLine is a disk-based system and uses its error bound (epsilon) to control the "fill variation" among pages in a segment. Bourbon [18] extends WiscKey [42] by replacing the block index per SSTable with a faster, more compact learned index. The benefits, however, are limited to this single granularity at which learned indexes are employed. The reorganization process in TreeLine instead maximizes the usefulness of each linear model, since it creates the largest possible segment per model, based on the data distribution.

Since both WiscKey and Bourbon are closed source, we cannot perform an apples-to-apples comparison. Bourbon achieves a 1.61× speedup over WiscKey on the Amazon dataset for a 100% read workload [18]. Assuming Bourbon achieves a similar speedup when integrated into RocksDB, TreeLine's 2.4× and 5× speedup (64 B vs. 1024 B records) over RocksDB on that dataset would be competitive.

Instance Optimization. There have been limited proposals for instance-optimized storage systems. Idreos et al. [20, 30] aim to instance-optimize (or automatically assemble) an entire storage system based on data and workload characteristics. Cosine [14] is a similar proposal targeted at the cloud. Unlike TreeLine, Cosine does not aim to improve update-in-place designs but rather assembles hybrid solutions from existing designs, including update-in-place.

8 CONCLUSION

We present TreeLine: a new update-in-place persistent key-value store for NVMe SSDs. TreeLine captures the read benefits of a classical disk-based B+ tree and mitigates its drawbacks to be competitive against LSMs on write-heavy workloads. TreeLine uses three complementary techniques: (A) *record caching* to reduce read/write amplification in skewed workloads, (B) *page grouping* to execute scans as sequential reads, and (C) *insert forecasting* to reduce the I/O cost of "making space" for new records. We evaluate TreeLine on YCSB using synthetic and real world datasets. On YCSB, TreeLine outperforms RocksDB and LeanStore by 2.20× and 2.07× respectively on average across the point workloads, and by up to 10.95× and 7.52× overall. We have open-sourced TreeLine [60].

ACKNOWLEDGMENTS

This research is supported by Google, Intel, and Microsoft as part of DSAIL at MIT, and NSF IIS 1900933. Geoffrey X. Yu was partially supported by an NSERC PGS D. This research was also sponsored by the United States Air Force Research Laboratory and the United States Air Force Artificial Intelligence Accelerator and was accomplished under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the United States Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

REFERENCES

[1] Hussam Abu-Libdeh, Deniz Altınbüken, Alex Beutel, Ed H. Chi, Lyric Doshi, Tim Kraska, Xiaozhou, Li, Andy Ly, and Christopher Olston. 2020. Learned Indexes for a Google-scale Disk-based Database. arXiv:2012.12501 [cs.DB]

[2] Adnan Alhomssi and Viktor Leis. 2022. *LeanStore Commit.* Retrieved September 15, 2022 from https://github.com/leanstore/leanstore/commit/d3d83143ee74c54c901fe5431512a46965377f4e

[3] Amazon Web Services, Inc. 2022. *Amazon EC2 C5 Instances.* Retrieved September 15, 2022 from https://aws.amazon.com/ec2/instance-types/c5/

[4] Apache Software Foundation. 2008. *Apache HBase.* Retrieved September 15, 2022 from https://hbase.apache.org

[5] Jens Axboe. 2022. *fio.* Retrieved September 15, 2022 from https://fio.readthedocs.io/en/latest/

[6] Timo Bingmann. 2018. *TLX: Collection of Sophisticated C++ Data Structures, Algorithms, and Miscellaneous Helpers.* Retrieved September 15, 2022 from https://panthema.net/tlx

[7] Burton H. Bloom. 1970. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (jul 1970), 422–426. https://doi.org/10.1145/362686.362692

[8] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. 2013. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC'13).* https://www.usenix.org/conference/atc13/technical-sessions/presentation/bronson

[9] Mark Callaghan. 2018. *Name that compaction algorithm.* Retrieved September 15, 2022 from https://smalldatum.blogspot.com/2018/08/name-that-compaction-algorithm.html

[10] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST'20).*

[11] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin J. Levandoski, James Hunter, and Mike Barnett. 2018. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD'18).* 275–290. https://doi.org/10.1145/3183713.3196898

[12] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin J. Levandoski, James Hunter, and Mike Barnett. 2018. FASTER: An Embedded Concurrent Key-Value Store for State Management. *Proc. VLDB Endow.* 11, 12 (2018), 1930–1933. https://doi.org/10.14778/3229863.3236227

[13] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2006. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06).*

[14] Subarna Chatterjee, Meena Jagadeesan, Wilson Qin, and Stratos Idreos. 2021. Cosine: A Cloud-Cost Optimized Self-Designing Key-Value Storage Engine. *Proc. VLDB Endow.* 15, 1 (2021), 112–126. http://www.vldb.org/pvldb/vol15/p112-chatterjee.pdf

[15] Feng Chen, Rubao Lee, and Xiaodong Zhang. 2011. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA'11).* https://doi.org/10.1109/HPCA.2011.5749735

[16] Douglas Comer. 1979. The Ubiquitous B-Tree. *ACM Computing Surveys (CSUR)* 11, 2 (1979), 121–137.

[17] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC'10).* https://doi.org/10.1145/1807128.1807152

[18] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2020. From WiscKey to Bourbon: A Learned Index for Log-Structured Merge Trees. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20).* USENIX Association, 155–171. https://www.usenix.org/conference/osdi20/presentation/dai

[19] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal Navigable Key-Value Store. In *Proceedings of the 2017 International Conference on Management of Data (SIGMOD'17).* ACM, 79–94. https://doi.org/10.1145/3035918.3064054

[20] Niv Dayan and Stratos Idreos. 2019. The Log-Structured Merge-Bush & the Wacky Continuum. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD'19).* ACM, 449–466. https://doi.org/10.1145/3299869.3319903

[21] Niv Dayan and Moshe Twitto. 2021. Chucky: A Succinct Cuckoo Filter for LSM-Tree. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD'21).* ACM, 365–378. https://doi.org/10.1145/3448016.3457273

[22] Peter C. Dillinger and Stefan Walzer. 2021. Ribbon filter: practically smaller than Bloom and Xor. *CoRR* abs/2103.02515 (2021). arXiv:2103.02515 https://arxiv.org/abs/2103.02515

[23] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, and et al. 2020. ALEX: An Updatable Adaptive Learned Index. *Proceedings of the 2020 International Conference on Management of Data (SIGMOD'20)* (2020). https://doi.org/10.1145/3318464.3389711

[24] Paolo Ferragina and Giorgio Vinciguerra. 2020. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proc. VLDB Endow.* 13, 8 (2020), 1162–1175. https://doi.org/10.14778/3389133.3389135

[25] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. FITing-Tree: A Data-Aware Index Structure. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD'19).* https://doi.org/10.1145/3299869.3319860

[26] Sebastien Godard. 1999. *iostat.* Retrieved September 15, 2022 from https://github.com/sysstat/sysstat

[27] Google, Inc. 2011. *LevelDB.* Retrieved September 15, 2022 from https://github.com/google/leveldb

[28] Google, Inc. 2022. *S2 Geometry.* Retrieved September 15, 2022 from https://github.com/google/s2geometry

[29] Jun He, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. The Unwritten Contract of Solid State Drives. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys'17).* https://doi.org/10.1145/3064176.3064187

[30] Stratos Idreos, Niv Dayan, Wilson Qin, Mali Akmanalp, Sophie Hilgard, Andrew Ross, James Lennon, Varun Jain, Harshita Gupta, David Li, and Zichen Zhu. 2019. Design Continuums and the Path Toward Self-Designing Key-Value Stores that Know and Learn. In *Proceedings of the 9th Biennial Conference on Innovative Data Systems Research (CIDR'19).* http://cidrdb.org/cidr2019/papers/p143-idreos-cidr19.pdf

[31] Intel Corporation. 2017. *Intel DC P4510.* Retrieved September 15, 2022 from https://ark.intel.com/content/www/us/en/ark/products/122573/intel-ssd-dc-p4510-series-1-0tb-2-5in-pcie-3-1-x4-3d2-tlc.html

[32] Intel Corporation. 2019. *Intel Xeon Gold 6230 CPU.* Retrieved September 15, 2022 from https://ark.intel.com/content/www/us/en/ark/products/192437/intel-xeon-gold-6230-processor-27-5m-cache-2-10-ghz.html

[33] Intel Corporation. 2021. *Intel Optane Technology.* Retrieved December 15, 2021 from https://www.intel.ca/content/www/ca/en/architecture-and-technology/intel-optane-technology.html

[34] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2019. SOSD: A Benchmark for Learned Indexes. *NeurIPS Workshop on Machine Learning for Systems* (2019).

[35] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD'18).* https://doi.org/10.1145/3183713.3196909

[36] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-Memory Data Management Beyond Main Memory. In *Proceedings of the 34th IEEE International Conference on Data Engineering (ICDE'18).* IEEE Computer Society, 185–196. https://doi.org/10.1109/ICDE.2018.00026

[37] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases. In *Proceedings of the 29th IEEE International Conference on Data Engineering (ICDE'13).* IEEE Computer Society, 38–49. https://doi.org/10.1109/ICDE.2013.6544812

[38] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of Practical Synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware (DaMoN'16).* https://doi.org/10.1145/2933349.2933352

[39] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2019. KVell: the Design and Implementation of a Fast Persistent Key-Value Store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19).* ACM, 447–461. https://doi.org/10.1145/3341301.3359628

[40] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2020. KVell+: Snapshot Isolation without Snapshots. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20).* USENIX Association. https://www.usenix.org/conference/osdi20/presentation/lepers

[41] Kim-Hung Li. 1994. Reservoir-sampling algorithms of time complexity O(n(1+log(N/n))). *ACM Transactions on Mathematical Software (TOMS)* 20, 4 (1994), 481–493.

[42] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16).* USENIX Association, 133–148. https://www.usenix.org/conference/fast16/technical-sessions/presentation/lu

[43] Chen Luo and Michael J. Carey. 2019. LSM-based storage techniques: a survey. *The VLDB Journal* 29, 1 (Jul 2019), 393–418. https://doi.org/10.1007/s00778-019-00555-y

[44] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. 2020. Benchmarking Learned Indexes. *Proc. VLDB Endow.* 14, 1 (2020), 1–13.

[45] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S. Berger, Nathan Beckmann, and Gregory R. Ganger. 2021. Kangaroo: Caching Billions of Tiny Objects on Flash. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP'21)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3477132.3483568

[46] Meta Platforms, Inc. 2020. *RocksDB v6.14.6.* Retrieved September 15, 2022 from https://github.com/facebook/rocksdb/releases/tag/v6.14.6

[47] Meta Platforms, Inc. 2021. *RocksDB Tuning Guide.* Retrieved September 15, 2022 from https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide

[48] Meta Platforms, Inc. 2022. *RocksDB.* Retrieved September 15, 2022 from https://rocksdb.org

[49] Meta Platforms, Inc. 2022. *RocksDB Prefix Seek.* Retrieved September 15, 2022 from https://github.com/facebook/rocksdb/wiki/Prefix-Seek

[50] Meta Platforms, Inc. 2022. *Universal Compaction.* Retrieved September 15, 2022 from https://github.com/facebook/rocksdb/wiki/Universal-Compaction

[51] MongoDB, Inc. 2008. *WiredTiger.* Retrieved September 15, 2022 from https://source.wiredtiger.com/

[52] Michael A Olson, Keith Bostic, and Margo Seltzer. 1999. Berkeley DB. In *Proceedings of the 1999 USENIX Annual Technical Conference (USENIX ATC '99)*. 183–191.

[53] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica* 33, 4 (1996), 351–385.

[54] Joseph O'Rourke. 1981. An On-Line Algorithm for Fitting Straight Lines Between Data Ranges. *Commun. ACM* 24, 9 (September 1981), 574–578. https://doi.org/10.1145/358746.358758

[55] Varun Pandey, Andreas Kipf, Dimitri Vorona, Tobias Mühlbauer, Thomas Neumann, and Alfons Kemper. 2016. High-Performance Geospatial Analytics in HyPerSpace. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD'16)*. ACM, 2145–2148. https://doi.org/10.1145/2882903.2899412

[56] Tarikul Islam Papon and Manos Athanassoulis. 2021. A Parametric I/O Model for Modern Storage Devices. In *Proceedings of the 17th International Workshop on Data Management on New Hardware (DaMoN'21)*. ACM, 2:1–2:11. https://doi.org/10.1145/3465998.3466003

[57] Kai Ren, Qing Zheng, Joy Arulraj, and Garth Gibson. 2017. SlimDB: A Space-Efficient Key-Value Storage Engine For Semi-Sorted Data. *Proc. VLDB Endow.* 10, 13 (2017), 2037–2048. https://doi.org/10.14778/3151106.3151108

[58] Benjamin Spector, Andreas Kipf, Kapil Vaidya, Chi Wang, Umar Farooq Minhas, and Tim Kraska. 2021. Bounding the Last Mile: Efficient Learned String Indexing. *3rd International Workshop on Applied AI for Database Systems and Applications* (2021).

[59] Sean J. Taylor and Benjamin Letham. 2017. Forecasting at Scale. *PeerJ Prepr.* 5 (2017), e3190. https://doi.org/10.7287/peerj.preprints.3190v1

[60] Geoffrey X. Yu, Markos Markakis, Andreas Kipf, Per-Åke Larson, Umar Farooq Minhas, and Tim Kraska. 2022. *TreeLine open-source implementation.* Retrieved September 15, 2022 from https://github.com/mitdbg/treeline The first three authors contributed equally.