Rethinking Update-in-Place Key-Value Stores for Modern Storage

by

Markos Markakis

B.S.E. Electrical Engineering
Princeton University, 2020

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2022

Author . . . .
Department of Electrical Engineering and Computer Science
May 13, 2022

Certified by . . . .
Tim Kraska
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . .
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Rethinking Update-in-Place Key-Value Stores for Modern Storage

by

Markos Markakis

Submitted to the Department of Electrical Engineering and Computer Science
on May 13, 2022, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

Several widely-used key-value stores, like RocksDB, are designed around log-structured merge trees (LSMs). Optimizing for the performance characteristics of HDDs, LSMs provide good write performance by emphasizing sequential access to storage. However, this approach negatively impacts read performance: LSMs must employ expensive compaction jobs and memory-consuming Bloom filters in order to achieve reasonably fast reads. In the era of NVMe SSDs, we argue that this trade-off between read performance and write performance is sub-optimal. With enough parallelism, modern storage media have comparable random and sequential access performance, making update-in-place designs, which traditionally provide high read performance, a viable alternative to LSMs.

In this thesis, based on a research paper currently under submission, we close the gap between log-structured and update-in-place designs on modern SSDs by taking 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 2.18× and 2.05× respectively on average across the point workloads, and by up to 10.87× and 7.78× overall.

Thesis Supervisor: Tim Kraska
Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgments

The contents of this thesis have been adapted from a research paper[1] currently under submission to the 49th International Conference on Very Large Data Bases (VLDB).

I would like to thank all the members of the DSG group for their support and mentoring. First and foremost, I would like to thank my advisor, Tim Kraska, for welcoming me to his research group and providing invaluable guidance throughout the past two years. At the same time, the unwavering day-to-day mentoring of Andreas Kipf, as well as his development of the insert forecasting aspect of TreeLine, have been instrumental over the duration of this project. The key industry perspectives shared by Umar Farooq Minhas were also essential in positioning this work. Finally, I am grateful for my collaboration with Geoffrey Yu, without whose creativity and hard work TreeLine could not have been developed. Among other things, Geoffrey was responsible for implementing the page grouping aspect of our project, as well as for running extensive experiments for our evaluation. I would also like to thank all of the fellow students and post-docs in the DSG group for their contributions, questions and comments that pushed this project forward. Beyond our research group, I am extremely thankful for the generous support of these projects by MIT's Electrical Engineering and Computer Science Department.

Last but certainly not least, none of my academic endeavors would have been possible without the continuous supportive presence of my family and friends, both in the US and back home. Thank you for being there for me through it all.

[1] *TreeLine: An Update-In-Place Key-Value Store for Modern Storage*, Geoffrey Yu*, **Markos Markakis***, Andreas Kipf*, Per-Åke Larson, Umar Farooq Minhas and Tim Kraska.
* denotes equal contribution.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Modern persistent key-value stores, such as RocksDB [25] and LevelDB [30], are typically structured around log-structured merge trees (LSMs) [48]. LSMs are popular because they provide stellar write performance, utilizing the key idea of *buffered log structuring*. Writes are first buffered in memory, and only periodically flushed to immutable files on disk, reducing I/O traffic. These files are then infrequently compacted (i.e., merged) in the background to remove overwritten and deleted records. By leveraging this scheme, LSMs ensure that all disk writes are relatively large and sequential, which exploits the high sequential write bandwidth of traditional disks.

Despite these benefits, LSMs are not a silver bullet in key-value store design. While their approach makes writes efficient, it impacts reads negatively, since records for the same key can be present in multiple locations on disk, requiring several I/Os to retrieve the most recent version or determine that the record doesn't exist. To offset this performance impact, systems like RocksDB and LevelDB employ block caches, Bloom filters [6, 21], and various compaction strategies [8, 27, 18]—complex and hard-to-tune [23] techniques all aimed at reducing the I/O overhead of reads.

For traditional disks (e.g., HDDs and SATA SSDs), this 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 significant random I/O drawbacks associated with older storage media [32]. With enough request parallelism, NVMe SSDs can achieve their peak sequential write throughput through random writes [41, 32, 50]. 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?

We make the case for *update-in-place* designs as 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, 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 [47, 7] or analytics [42, 3, 10] are common in practice [9].

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 sequential reads still outperform random 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, our goal is to develop a new update-in-place design for NVMe SSDs that has the benefits of a classical disk-based B-tree while also mitigating its drawbacks to be competitive against LSMs on write-heavy workloads. We propose a new design consisting of three complementary techniques that address these challenges: (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 first (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 enables efficient use of the cache's capacity even when hot records are not clustered together on the same few pages.

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, TreeLine exploits the patterns in skewed insert workloads to *forecast* the future location and volume of inserts (key idea C). It uses the forecast to *leave enough 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).

These techniques provide benefits over other recent update-in-place designs. Let's look at two example recent systems. LeanStore [38] does not optimize for the out-of-memory case, which we target, even though it is efficient when the working set fits into memory. KVell [41] indexes all keys in the database, leading to a higher memory overhead than TreeLine, which only indexes segment boundaries using page grouping.

We evaluate TreeLine on YCSB [15] using synthetic and real-world datasets. We compare TreeLine against (i) RocksDB [25], a production-grade LSM key-value store, and (ii) LeanStore [38], 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 2.18× and 2.05× respectively on average. Although TreeLine makes random reads from disk, we find that it still outperforms RocksDB and LeanStore by 2.59× and 2.90× respectively with 16 threads on uniform scan-heavy workloads.

1.1 Contributions.

In summary, we make the following contributions:

- We analyze the key performance challenges associated with deploying an update-in-place key-value store on NVMe SSDs and make the case for addressing them with our key ideas.
- 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 10.87× and 7.78× respectively overall on YCSB. We will open source TreeLine.

Chapter 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.

2.1 Random $\approx$ Sequential Writes on NVMe SSDs.

In recent years, 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 (even though many SSDs still employ optimizations for the sequential case), and (ii) highly parallel requests are now feasible [14], which can be used to hide most of the access latency [50].

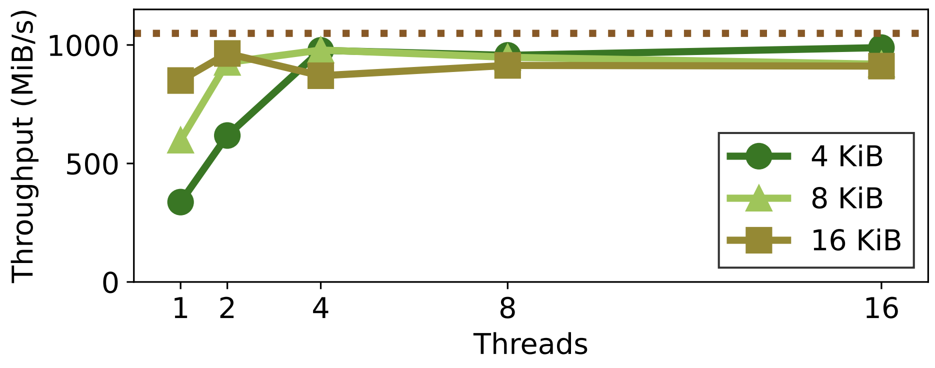


Figure 2-1: An NVMe SSD's random write throughput as we increase the number of concurrent writing threads. The dotted line is the SSD's advertised peak sequential write throughput. As evident, the random write throughput approaches the peak sequential write throughput for a sufficient number of concurrent threads.

To empirically confirm these characteristics, we run a series of experiments on an Intel DC P4510 NVMe SSD [16]. As shown in Figure 2-1, we use fio [4] 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, the device's *random* write throughput approaches its advertised peak *sequential* write throughput (1050 MiB/s [16]). As such, leveraging the high parallelism available in this storage medium is able to close the gap between random write and sequential write performance.

2.2 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 production-grade 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, 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.

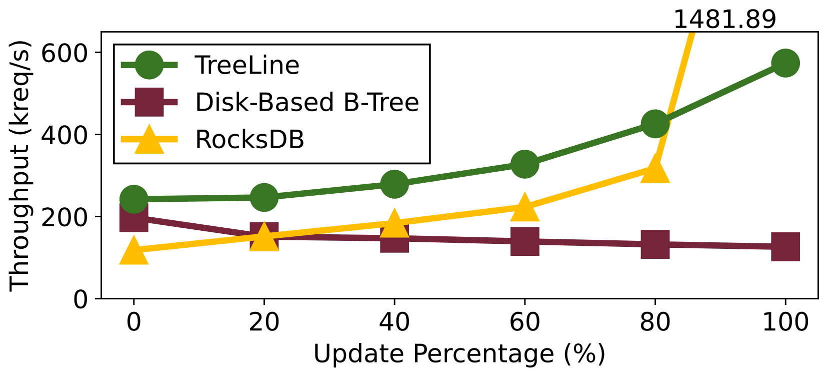


Figure 2-2: TreeLine, a disk-based B-tree, and RocksDB compared on Zipfian read/scan/update workloads. We vary the proportion of updates in the workload from 0% to 100%.

Figure 2-2 shows each system's throughput in thousands of requests per second. For read-heavy workloads (e.g., less than 40% updates), our naïve disk-based B-tree outperforms —or is competitive against— RocksDB, a highly optimized production-grade system (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. Note that RocksDB's exceptional performance for the update-only case is possibly related to some special handling of that workload, something that we have not implemented in TreeLine.

We outline the key ideas behind TreeLine in Section 3 and we describe why and when it outperforms RocksDB in Section 5.

2.3 Summary.

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

Chapter 3

TreeLine: Key Ideas

To best leverage the performance characteristics of modern storage media, we explore three key ideas for an efficient update-in-place key-value store, described below.

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 spilling to storage instead of a purely in-memory solution, it is likely 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, it is a well-known fact that some form of caching would be beneficial. One of the key design questions at this point is, 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. This contrasts with the design of systems like RocksDB, which a block cache. Certainly, a record cache 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 the case where the working set is completely uncorrelated with the key sort order — one example would be a database of user metadata, where the activity of users is not correlated with their `user_id`. An LSM design, like RocksDB [25], would consolidate updates from hot users in the write buffer and then create files containing only these records in level 0. 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, would not be able to make the most of a page cache. Because it aims to maintain a single "true" copy of the record, there are two costly alternatives: (i) employ frequent and costly reorganizations to consolidate hot records in "hot pages", which should then be cached, or (ii) distribute records into pages in some static workload-independent way (e.g. have each page responsible for a specific key range), which could mean that an entire page's worth of cache space could be needed to cache a single few-byte hot record. 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, pages should be *as small as possible* (often 4 KiB in practice), so that we only bring and keep in memory the exact pieces of data that we need. Indeed, pages can be seen as simply an unfortunate artifact of current storage technology, with novel media like Optane persistent memory [35] pushing the boundary towards byte addressability.

However, small pages can hurt range scans (assuming records are sorted on disk), which benefit the most from reading large amounts of contiguous data from storage. This is because we observe that random reads of small pages (e.g., 4 KiB) still do not perform as well as sequential reads on NVMe SSDs. If records are sorted on disk, the best way to utilize the full read bandwidth of the storage device is through large requests [32, 50], making large pages attractive.

To bridge this gap, we propose *page grouping*. The idea is that pages themselves are small, but stored contiguously 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 what we call *data segments*. This approach also gives us the opportunity to reduce the number of entries we have to index in memory: only one index entry per segment is needed. We are able to 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 [22, 1]).

The benefits of this approach are that it (i) allows for long reads when needed, and (ii) leads to a compact in-memory index without sacrificing page-level access for point lookups and updates. 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 record's true position in the dataset. A user would typically set these parameters to be as large as possible, within the constraints of the maximum number of records that can fit in a page. For example, when using 64 byte records, each page fits 55 records. So we set goal to 45 and epsilon to 5 in our experiments. 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 [1]

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, leveraging patterns in the inserts. We can maintain an in-memory histogram to track the distribution of inserts and use it to *forecast inserts* for each part of the key space. During reorganization, we then leave empty space according to this forecast, decreasing space amplification while still absorbing inserts in a high-performance manner.

[1] ...like this one

Chapter 4

TreeLine: Implementation Details

In this section we describe TreeLine, which implements our three key ideas. As shown in Figure 4-1, TreeLine responds to the performance characteristics of modern storage media by using an update-in-place design consisting of two 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.

4.1 Supported Operations

4.1.1 Lookups.

Lookups first check the record cache (Section 4.2), which will contain 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 that segment's linear model (if any) is used to find the appropriate data page within the segment (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.

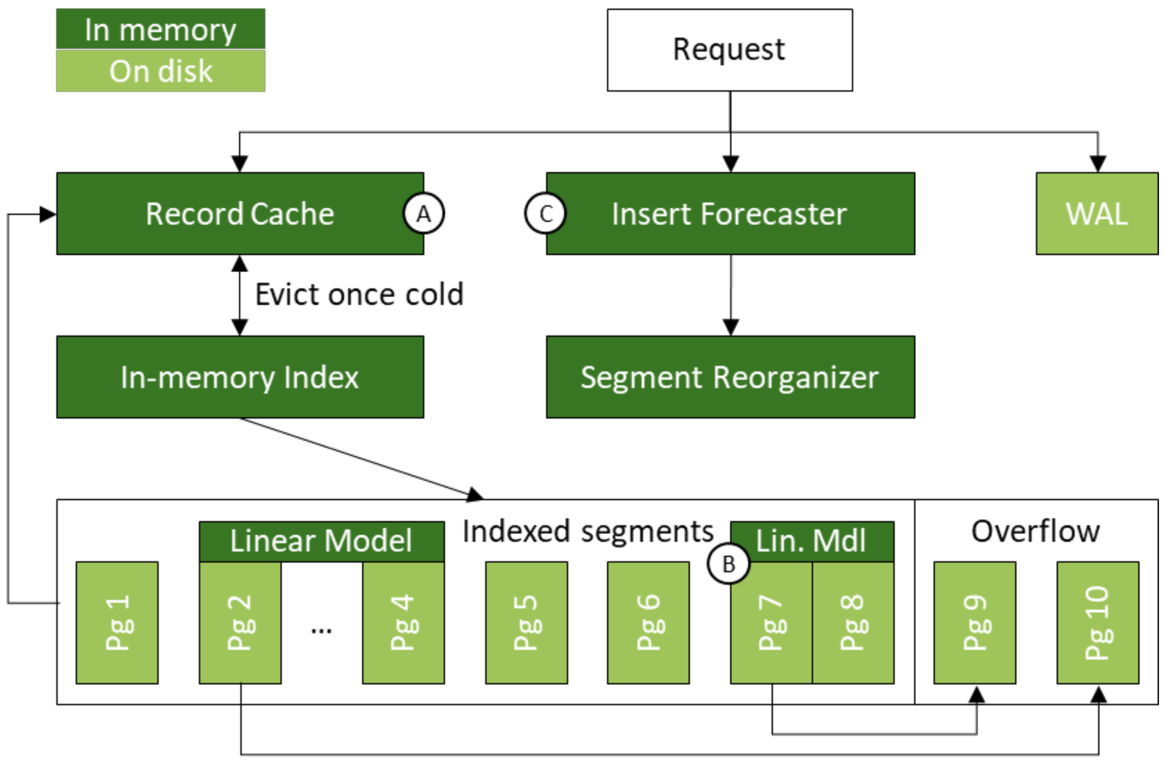


Figure 4-1: The system architecture of TreeLine, highlighting our three key ideas.

4.1.2 Data Modifications.

Inserts, updates, and deletes initially create or update an entry in the record cache. A special delete marker is used as the value, in the case of deletes. Once the entry is later selected for eviction as explained in Section 4.2.2, TreeLine finds the appropriate data page using the in-memory index and segment linear model (if any), and brings it into memory to perform the operation in place, 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) and the operation is performed afterwards.

4.1.3 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)

4.2.1 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 and decremented as part of the eviction protocol, described later.

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.

4.2.2 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 additional 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, the appropriate page is brought into memory, updated, and written back out to storage as part of the eviction protocol. In such cases, 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.

4.3 In-Memory Index

The in-memory index is our guide to the on-disk portion of the data. We employ the TLX `btree_map` [5], a fast but otherwise traditional B+tree.

4.3.1 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 a key. 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 together with the page identifier in the in-memory index, in order to select the correct data page within the segment without needing additional index nodes.

4.3.2 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 [40] instead in future work.

4.4 Pages and Segments

4.4.1 Data Pages.

TreeLine stores data in data pages. We adapt the physical page design from the implementation of `BTreeNode` in LeanStore [38, 2]. 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.

4.4.2 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)

4.5.1 Overflow Pages.

Once a data page is 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 designed 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 given for the key order across 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 to bound the cost of lookups and data modifications. Note that the full segment may still contain pages in it that can accommodate inserts. However, it contains at least one full base page with a full overflow page, so there is a sub-range of keys for which it can accept no more inserts.

4.5.2 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, 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.

4.5.3 Reorganization.

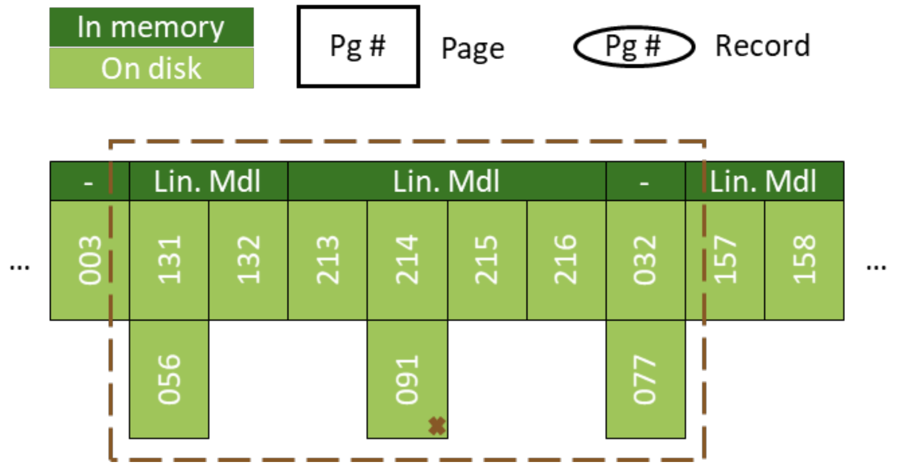
We now move on to the reorganization process, which is divided into four phases. *Range detection* is followed by one or more iterations of *model building* and *segment write-out*, followed by the *index update*. We provide an example in Figure 4-2.

Phase I: Range Detection.

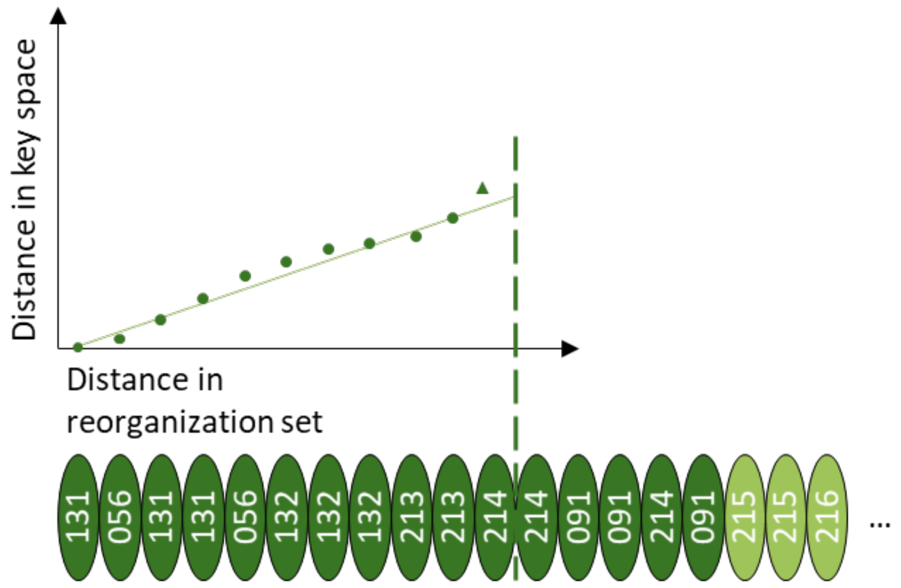
Only considering records from the full segment could lead to sub-optimal data layouts in the long run. Instead, TreeLine also looks "around" the full segment S for any segments with *reorganization potential*: segments including at least one base page with an overflow page. It examines neighboring segments in key order, up to a distance of r segments from S in each direction, for a configurable constant r (currently set to 5). 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 4-2a, Phase I finds two neighboring segments with reorganization potential around S.

Phase II: Model Building.

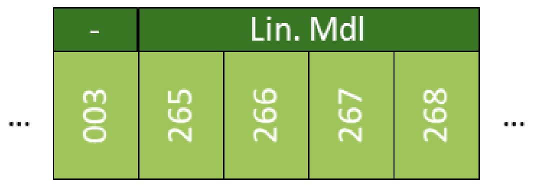
This phase operates on the sorted set of all records in the pre segments, called the *reorganization set*, if it is non-empty. It is visualized as the ovals in Figure 4-2b.



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



(b) Phase II works through the reorganization set and fits a linear model. 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 4-2: An example reorganization.

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 4-2b, Pages 131, 056, 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. To simplify the management of its on-disk files, TreeLine only creates fixed-size segments containing 1, 2, 4, 8, or 16 pages. So if the largest possible segment contains 10 pages, TreeLine will only create an 8 page segment and will leave the leftover records in the reorganization set for processing in another iteration of Phase II.

While considering each record, TreeLine builds a linear model (light green line in Figure 4-2b) using the GreedyPLR algorithm [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* (called δ in GreedyPLR [54]), indicated by the dashed dark green line in Figure 4-2b. 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 is complete. Any records we processed are forwarded to Phase III as a *write-out set* and removed from the reorganization set, before moving to the next iteration of 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 to fit a 16 page segment without exceeding epsilon.

Phase III: Segment Write-out.

Given a write-out set and the associated model from Phase II, Phase III materializes the proposed post segment. As shown in Figure 4-2c, the model is used to place the records in a number of logically contiguous pages on the SSD.

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 a globally unique *production ID* that is stored on the first page of each segment with a special value, and added to a free list.

4.6 Insert Forecasting (Key Idea C)

4.6.1 Utilizing Forecasts.

During reorganization, we use insert forecasting to leave empty space in each page, by setting the goal parameter. Using an estimate of the number of keys in the current segment (we pessimistically assume that all pages in the segment are full), 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 across these pages.

4.6.2 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* [43] 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 $10 * b$.

4.6.3 Generating Forecasts.

To forecast inserts for a given segment, we use the segment boundaries to query the "frozen" histogram and sum up the counters of all intersecting partitions. For partitions that only partially overlap the query range, we use interpolation. We can then forecast the inserts for the f epochs following the epoch of the "frozen" histogram.

4.7 Crash Consistency & Recovery

4.7.1 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 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.

4.7.2 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 examined again in the end of the scan, by which point the corresponding base page should be already indexed. We then use the lower bound of the overflow page to find the correct base page through the in-memory index, and ensure the base page points to the overflow page (**Check I**).

Table 4.1: 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

Before the data page scan, TreeLine checks the write-ahead log for any START REORG records not accompanied by an END REORG record with the same production ID (**Check II**). 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 recorded in the START REORG record. In this case, we consider the reorganization unsuccessful, since we do not know whether Phase IV was reached. We 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.

At each crash, there could be at most one concurrent reorganization per user thread, providing an upper bound on the amount of work needed during recovery.

4.8 Thread Synchronization

TreeLine uses both segment locks and page locks. Table 4.1 provides their compatibility matrices. Segment locks can be acquired in IR (intention read), IW (intention write), O (reorganization) or OX (reorganization exclusive) mode. Page locks can be acquired in S (shared) or X (exclusive) mode. Each page lock protects both a base page and its overflow page, if any. We will now explain the locking strategy.

4.8.1 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.

4.8.2 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.

4.8.3 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 the risk of an intervening reorganization, by first acquiring an IR lock on the next segment and then releasing the IR lock on the previous one.

4.8.4 Reorganization.

After Phase I, reorganization locks all the pre segments in key order in O mode. This lets reads and scans 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. Once the in-memory index has been updated and at least one pre segment has been invalidated, the OX lock is released.

Chapter 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 [25] (an LSM-based key-value store) and LeanStore [38] (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 2.18× and LeanStore by 2.05× on average across our 1024 byte YCSB point workloads. With 16 request threads, TreeLine outperforms RocksDB and LeanStore by 2.59× and 2.90× respectively on uniformly distributed scan-heavy workloads (YCSB E), averaged across three datasets.

Table 5.1: Our system configurations for the Amazon dataset.

Config.	**System**	**Setup Details**
64 B	TreeLine	45/5 page grp. goal/epsilon, 683 MiB rec. cache
	RocksDB	107 MiB memtables (×2), 469 MiB block cache
	LeanStore	683 MiB buffer pool
1024 B	TreeLine	2/1 page grp. goal/epsilon, 10903 MiB rec. cache
	RocksDB	1715 MiB memtables (×2), 7473 MiB block cache
	LeanStore	10903 MiB buffer pool

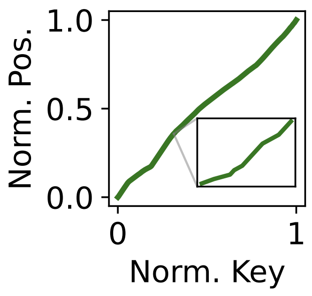
5.1 Experimental Setup

5.1.1 Baselines.

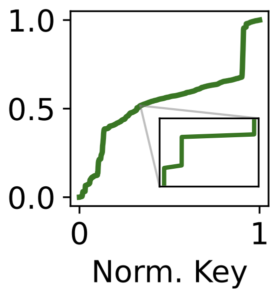
We compare TreeLine against RocksDB [25] (an LSM key-value store) and LeanStore [38] (a state-of-the-art update-in-place key-value store). We use RocksDB version 6.14.6 [24] and LeanStore at commit `d3d8314` [2]. For a fair comparison, we disable block compression and sstable checksums in RocksDB, since 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 [26] 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 from the logging overhead.

5.1.2 System Configurations.

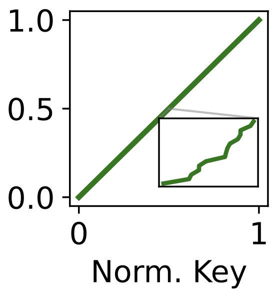
We use 64 byte records *(64 B)* (8 byte key, 56 byte value) and 1024 byte records *(1024 B)* (8 byte key, 1016 byte value). Depending on the dataset, we give each system enough memory to store up to 33% of the dataset in memory. Table 5.1 describes how this memory is used for the Amazon dataset; it also lists the page grouping parameters used. We give TreeLine, RocksDB, and LeanStore access to 4 background threads.



(a) Amazon



(b) OSM



(c) Synthetic

Figure 5-1: Our datasets' key CDFs.

5.1.3 Hardware and Environment.

We use a machine equipped with a 20-core 2.10GHz Intel Xeon Gold 6230 CPU [34] and 131 GB of memory, running Linux 5.12.5. We use a 1 TB Intel DC P4510 NVMe SSD [16] and the ext4 file system for all our experiments.

5.1.4 Workloads & Datasets.

We use our own C++ implementation of the Yahoo! Cloud Serving Benchmark (YCSB) [15] to perform our evaluation. 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) [17]. We plot their key cumulative distribution functions (CDFs) in Figure 5-1.

5.1.5 Checkpoints.

To ensure consistent results, we start each experiment from a copy of 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.

5.1.6 Metrics.

We primarily measure throughput and report it as thousands of requests processed per second (kreq/s). Using iostat [29], 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.

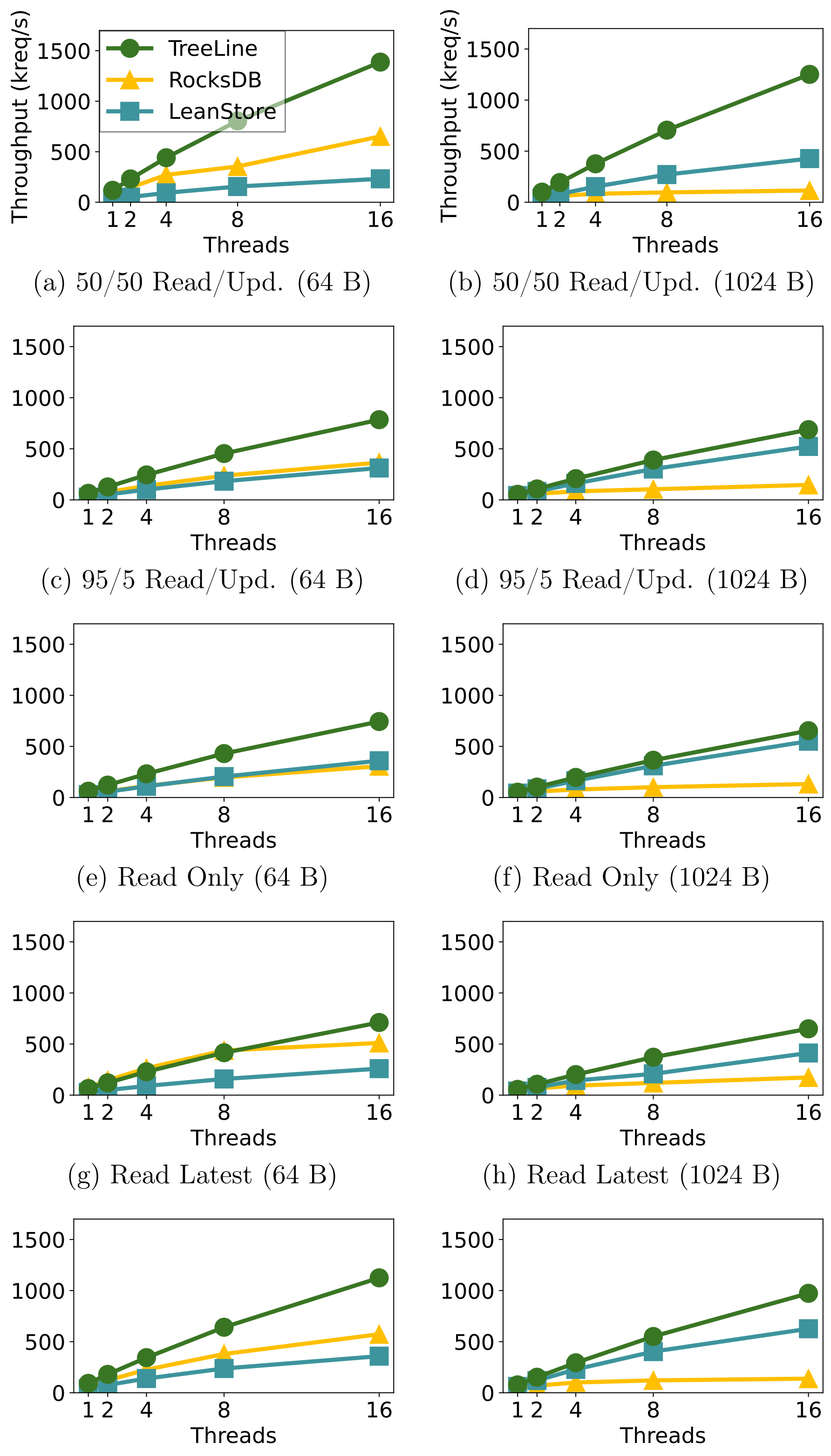
5.2 End-to-End Performance

5.2.1 Skewed Point Workloads

We begin by running the YCSB point workloads with Zipfian-distributed requests ($\theta = 0.99$). 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). Figure 5-2 shows TreeLine's, RocksDB's, and LeanStore's throughputs on these aforementioned workloads. TreeLine outperforms RocksDB (LeanStore) by 1.61× (2.79×) and 2.96× (1.51×) 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 cache to reduce write amplification while also (ii) providing efficient reads.

We observe that on workload A (64 B), TreeLine writes 3.02 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.



(a) 50/50 Read/Upd. (64 B) (b) 50/50 Read/Upd. (1024 B)

(c) 95/5 Read/Upd. (64 B) (d) 95/5 Read/Upd. (1024 B)

(e) Read Only (64 B) (f) Read Only (1024 B)

(g) Read Latest (64 B) (h) Read Latest (1024 B)

(i) 50/50 Read/RMW (64 B) (j) 50/50 Read/RMW (1024 B)

Figure 5-2: Zipfian YCSB point workloads on the Amazon dataset. The first (second) column shows results for 64 (1024) byte records.

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.

TreeLine and RocksDB achieve a higher throughput on update-heavy workloads, compared to read-only workloads, because 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 5.1× and 5.3× more data than TreeLine on workloads B and C respectively. Two reasons cause 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.

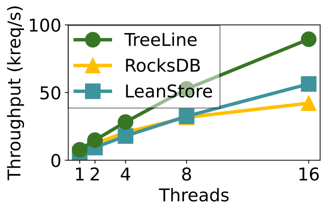
5.2.2 Scan-Heavy Workloads

Next, we examine TreeLine's performance on scan-heavy workloads (YCSB E). For scan workloads, page grouping influences performance. Since page grouping's effectiveness depends on the dataset, we show our scan-heavy results for all three of our datasets. We otherwise use the same experimental setup as our skewed point workload experiments. Note that we discuss page grouping effectiveness in more detail in Section 5.4.

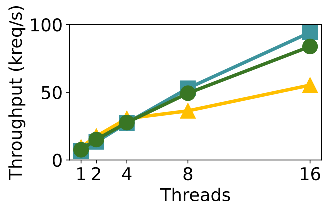
Figure 5-3 shows our results for uniformly-distributed scans and Zipfian-distributed scans. With enough parallelism (i.e., with 16 request threads), TreeLine achieves average speedups of 1.65× (2.02×) and 1.93× (2.59×) over RocksDB for Zipfian (uniformly) distributed requests against the 64 B and 1024 B configurations respectively. When compared against LeanStore, TreeLine achieves average speedups of 0.86× (1.45×) and 1.90× (2.90×) for Zipfian (uniformly) distributed requests against the 64 B and 1024 B configurations respectively. LeanStore surpasses TreeLine on some Zipfian-distributed scan workloads due to caching effects, which we discuss below. Note that we enable prefix Bloom filters in RocksDB. These filters help RocksDB reduce the number of sstables it needs to read during a scan, making it a strong baseline. 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.

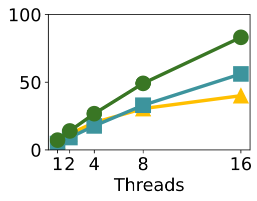
Table 5.2 lists each system's (i) physical reads, (ii) physical read throughput, and (iii) workload request throughput for a uniform scan-only workload on the Amazon dataset with 16 request threads. The trends are similar across the other datasets. In both configurations shown in the table, TreeLine 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.



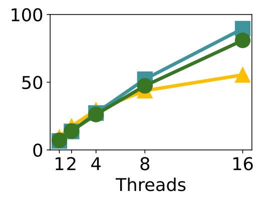
(a) Amazon 64 B (U)



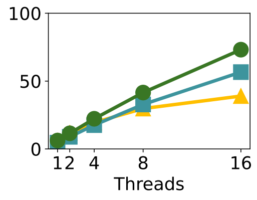
(b) Amazon 64 B (Z)



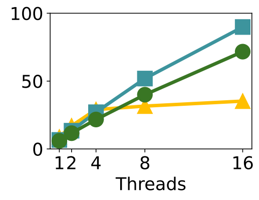
(c) OSM 64 B (U)



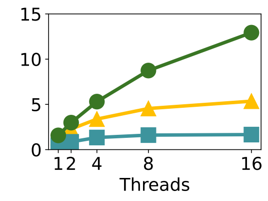
(d) OSM 64 B (Z)



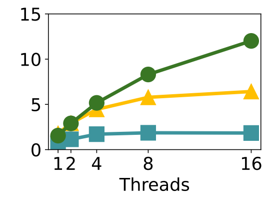
(e) Synthetic 64 B (U)



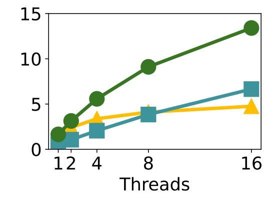
(f) Synthetic 64 B (Z)



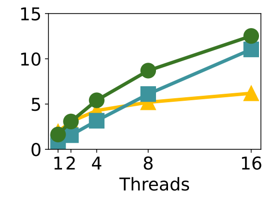
(g) Amazon 1024 B (U)



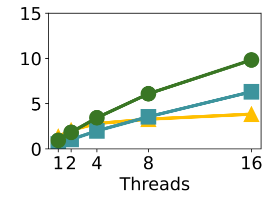
(h) Amazon 1024 B (Z)



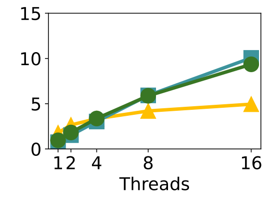
(i) OSM 1024 B (U)



(j) OSM 1024 B (Z)



(k) Synthetic 1024 B (U)



(l) Synthetic 1024 B (Z)

Figure 5-3: Scan-heavy (YCSB E) workloads on our three datasets for 64 byte records and 1024 byte records. The first (second) column shows uniformly (Zipfian) distributed request results.

Table 5.2: 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	**13.2 GiB**	500 MiB/s	**89 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	**76.0 GiB**	**1112 MiB/s**	**13 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

RocksDB reads more data from disk for two reasons: (i) there may exist sstables on multiple levels that overlap the scan range; RocksDB needs to read blocks from any such sstables (not excluded by the prefix Bloom filters) to find the correct records to return, and (ii) RocksDB still runs compaction jobs despite the workload being scan-only; this happens because the RocksDB checkpoints we use contain recent updates (see Section 5.1) and RocksDB always launches a compaction job after starting up.

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

TreeLine's scan performance is not significantly affected by request skew.
TreeLine's throughput advantage over RocksDB and LeanStore is higher on uniform scan only 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-Load [9]), we expect RocksDB's and LeanStore's caches to be less effective.

Page grouping drives TreeLine's speedups over LeanStore on 1024 B.

As shown in Table 5.2, 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. The reason for the difference is again due to the I/O request pattern. LeanStore scans result in scattered 4 KiB random 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 1.44× and 1.31× 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 1.54× (1.56×).

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 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 [9].

Table 5.3: TreeLine's performance on update-heavy and scan-only workloads as we enable record caching and page grouping.

50%/50% Read/Update			
Configuration	**Physical R/W (GiB)**	**Thpt. (kreq/s)**	**Speedup**
TreeLine Base (1024 B)	47.92 / 25.76	24.41	—
+ Record Cache	12.32 / 06.73	1251.47	**51.26**×
+ Page Grouping	12.25 / 07.07	1265.30	1.01×

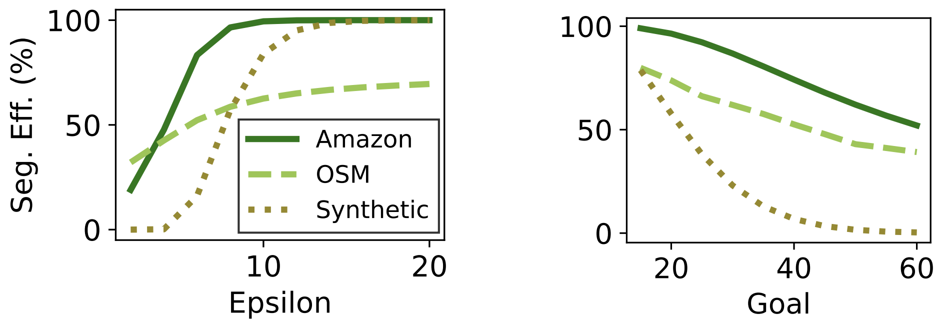
100% Scan			
Configuration	**Physical R/W (GiB)**	**Thpt. (kreq/s)**	**Speedup**
TreeLine Base (1024 B)	83.12 / 0	5.59	—
+ Record Cache	83.12 / 0	5.59	1.00×
+ Page Grouping	79.75 / 0	10.22	**1.83**×

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 a 50/50 read/update workload (YCSB A) and a scan-only workload with 16 request threads. We select these workloads to show how the two components affect TreeLine's performance on update-heavy point workloads and scan workloads. Table 5.3 shows our results. "TreeLine Base" is a configuration of TreeLine where we disable both the record cache and page grouping. We then enable the record cache, followed by page grouping. We report each configuration's throughput and the amount of physical reads and writes performed. From these results, we draw three key conclusions.

Record caching helps reduce I/O amplification in the point workload.

TreeLine's record cache reduces the amount of physical reads and writes by 3.89× and 3.83× respectively, leading to a throughput speedup of 51.26×. The workload is highly skewed. Thus by caching records that are read and/or updated frequently, TreeLine reduces the frequency with which it needs to access the SSD, which translates to throughput improvements.



(a) Goal = 45 (b) Epsilon = 5

Figure 5-4: The percentage of pages in multi-page segments as we fix "goal" and vary "epsilon" and vice-versa.

Page grouping does *not* negatively affect the read/update workload.

Even after enabling page grouping, TreeLine maintains its throughput on the read/update workload. 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 through longer physically contiguous reads.

On our scan-only workload, TreeLine's throughput increases by 1.83× 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 a higher physical read throughput when it makes 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

Now we (i) study how the dataset, goal, and epsilon affect page grouping effectiveness; and (ii) analyze page grouping effectiveness in our configurations from Section 5.2.

5.4.1 Page Grouping Sensitivity Study

We measure *segment efficiency* (the percentage of pages that are in a multi-page segment) across our datasets; a higher segment efficiency is better. In Figure 5-4a, we set goal to 45 and vary epsilon. In Figure 5-4b, we set epsilon to 5 and vary goal. From these results, we draw three conclusions.

Increasing epsilon improves segment efficiency.

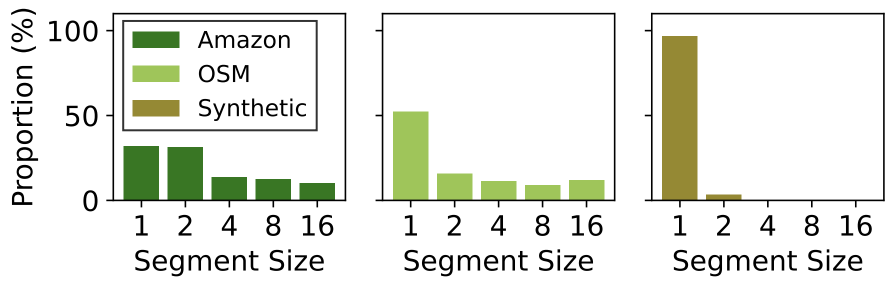
Epsilon represents the error tolerance when building a page grouping linear model. Having a larger error tolerance translates to being able to fit a model through *more* records. This tolerance translates to a greater likelihood of creating a multi-page segment (for a fixed goal value).

Increasing goal decreases segment efficiency.

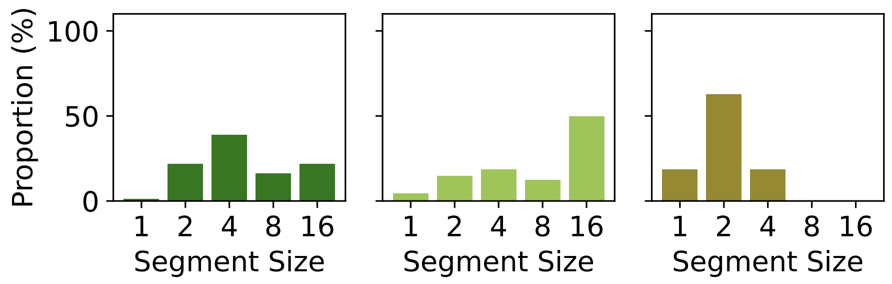
Goal is the desired number of records to place on a page. Recall that page grouping uses fixed segment sizes (1, 2, 4, 8, and 16 pages). Thus, choosing a small goal value decreases the maximum number of records that can be placed onto a segment (for a fixed epsilon). This is why Figure 5-4b shows a decreasing trend as goal increases: fitting a model through more records with a fixed epsilon becomes more difficult.

The dataset affects segment efficiency.

Page grouping's linear models map keys to positions, essentially fitting linear models over a dataset's CDF. Figure 5-1 plots of our datasets' CDFs. Intuitively, one needs fewer linear models to fit a CDF with many "linear regions" compared to a highly non-linear CDF (for a fixed epsilon). This is why the Amazon and OSM datasets appear to enable higher segment efficiencies: although their CDFs are non-linear, they have many "linear regions". Interestingly, our synthetic dataset generally produces lower segment efficiencies. Although the synthetic dataset is uniformly distributed, local regions of the CDF are generally "bumpier" than the Amazon and OSM datasets.



(a) 64 byte records. Goal = 45, Epsilon = 5.



(b) 1024 byte records. Goal = 2, Epsilon = 1.

Figure 5-5: Histograms of segment sizes across our experiments from Section 5.2.

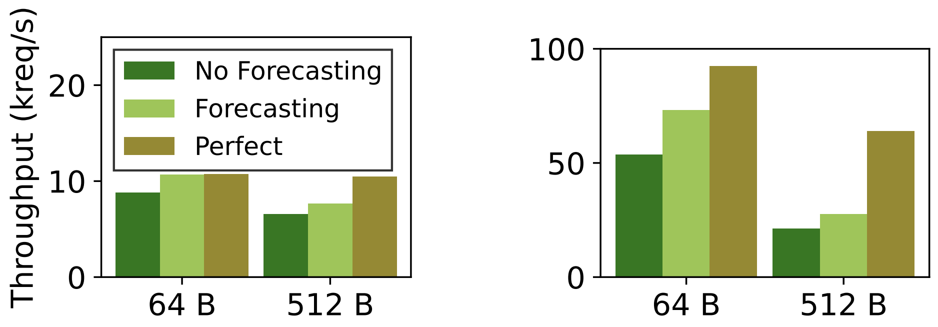
5.4.2 Page Grouping in Our Experiments from Section 5.2

Segment Distribution.

Figures 5-5a and 5-5b show the segment distributions for our 64 B and 1024 B configurations across our datasets. The figures show that our 1024 B configurations tend to have larger segments compared to the 64 B configurations. The primary reason is that the 1024 B configurations use a lower goal value (because fewer 1024 B records fit on a 4 KiB page), which makes it easier to form larger segments. Both configurations have comparable epsilon values, so epsilon has a smaller effect.

Index Entries Reduction.

Without page grouping, TreeLine needs to index each page boundary. With page grouping, TreeLine only indexes the boundaries of each segment (and the linear model). Reducing the number of index entries has two benefits: (i) we save memory for other uses, and (ii) index operations are accelerated.



(a) 1 application thread

(b) 8 application threads

Figure 5-6: Insert forecasting performance across record sizes and thread counts.

On our 1024 B (64 B) workloads, page grouping reduces the number of entries in the index by 4.2× (1.9×), 4.8× (1.5×), and 2.0× (1.0×) for the Amazon, OSM, and synthetic datasets respectively (higher is better). The reduction depends on the segment distribution; having more pages in larger segments means that fewer entries need to go into the index. Our results above explain why the reduction is larger for our 1024 B workloads than the 64 B workloads.

5.5 Insert Forecasting

Finally, we evaluate the third of our key ideas, insert forecasting. Among the YCSB workloads, only D and E perform inserts, and only for 5% of the operations. To showcase the impact of insert forecasting, we instead run a workload consisting of 50% reads/50% inserts. We compare the performance of TreeLine with forecasting (i) disabled, (ii) enabled, with the performance of TreeLine given perfect information about the future stream of inserts. 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 [31]. 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 5-6, insert forecasting is most effective in the 64 B case, closing on average 70% of the gap between no forecasting and perfect forecasting and reducing reorganizations also by an average of 70% (not plotted). This is explained by the fact that, with smaller records, the absolute number of inserts that a 4 KiB page can accommodate is higher. With accurate forecasting, we can leave enough free space to delay reorganization for longer. Overall, insert forecasting is able to improve the throughput of TreeLine by an average of 1.25×, reducing reorganizations by an average of 41%.

Chapter 6

Discussion

TreeLine can close the gap between update-in-place and LSM-tree designs for update-heavy workloads. However, it has a few limitations which we discuss below; we also sketch possible solutions.

6.1 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.

6.2 Non-Integer Keys.

We currently build a linear model on 8-byte integer keys. To handle string keys, we could follow a similar approach as in prior work [52]. The idea is to recursively build a radix tree of spline models with each node representing an 8-byte key prefix. We recurse until the model satisfies a pre-determined maximum prediction error. This is similar to the approach taken by the Adaptive Radix Tree (ART) [39], which only recurses until two keys can be differentiated and does not store the whole strings.

6.3 More Sophisticated Forecasts.

To forecast the distribution of inserts, we project the distribution of the last epoch into the future. This approach works well when there are no distribution shifts over short periods of time. For the taxi dataset, the insert distribution (taxi pickup locations) is similar over short periods of time [49]. To address more sophisticated insert distributions, such as cyclic patterns, we could employ a timeseries forecasting module such as Prophet [53]. We find Prophet to be efficient enough to continuously perform the forecasting in the background. However, we also find its forecasts to not be significantly better than those produced by our simple approach. This finding may obviously change with more complex insert distributions found in production workloads [9].

6.4 Pure Write Workloads.

TreeLine currently cannot compete with RocksDB on pure write workloads. We narrow that gap with insert forecasting, reducing reorganizations. However, even with perfect forecasting, TreeLine still suffers from its update-in-place paradigm as inserts follow a read-modify-write pattern (needing up to two I/Os) — RocksDB can simply write out its memtable to disk 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.

Chapter 7

Related Work

7.1 LSM Proposals.

LevelDB [30] pioneered the partitioned leveling merge policy [45], in which each level (except L0) contains multiple range-partitioned fixed-size files to improve concurrency. Building on LevelDB, RocksDB [25] provides additional compaction policies and a merge operation to reduce contention with user requests [45]. WiscKey [44] extends LevelDB and proposes separating values from keys by storing values in a separate values log. This design reduces write amplification by avoiding unnecessarily copying values during the merge. TreeLine, in contrast, might only need to merge a page with its single overflow page. It is hence unclear whether introducing key-value separation in TreeLine would yield significant benefits, considering the added cost of garbage collection.

Modern LSM-trees use in-memory Bloom filters to prune unnecessary disk accesses. Dayan et al. [18] show how to best allocate bits to each filter to improve the space-accuracy trade off. SlimDB [51] and Chucky [20] propose replacing per-level Bloom filters with a single multi-level Cuckoo filter, further improving pruning power.

7.2 Update-in-Place Designs.

We are not first to revisit update-in-place designs for modern hardware. LeanStore [38] 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 here. FASTER [10, 11] 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 [41] is a key-value store with a similar design to ours. It uses an in-memory B-tree that indexes every key in the database; on-disk data is kept unsorted. Its drawbacks are that (i) it needs to index all keys (which can have 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.

7.3 Learned Indexing.

Learned indexes [37, 36, 46] build a model over sorted data to predict the position of a key. Abu-Libdeh et al. [1] integrate learned indexes into Bigtable [12] and show that the index's reduced size improves cache efficiency [1]. Like TreeLine's page grouping, FITing-Tree [28] fits a piece-wise linear model over its data. However, FITing-Tree (i) is purely in-memory, 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 record "fill variation" among pages in a segment.

Bourbon [17] extends WiscKey [44] using a learned index per sstable. However, unlike TreeLine, Bourbon does not alter the physical storage layout based on the index. 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 [17]. 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.

7.4 Instance Optimization.

There have been limited proposals for instance-optimized storage systems. Idreos et al. [33, 19] aim to instance-optimize (or automatically assemble) an entire storage system based on data and workload characteristics. Cosine [13] is a similar proposal targeted at the cloud. Different from TreeLine, Cosine does not propose new techniques to make update-in-place designs more effective for writes but rather assembles hybrid solutions based on existing designs, including update-in-place.

Chapter 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 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 costs needed to "make space" for new records. We evaluate TreeLine on YCSB using synthetic and real world datasets. On YCSB, we find that TreeLine outperforms RocksDB and LeanStore by 2.18× and 2.05× respectively on average across the point workloads, and by up to 10.87× and 7.78× overall. We will open source TreeLine.

Bibliography

[1] Abu-Libdeh, H., Altinbüken, D., Beutel, A., Chi, E. H., Doshi, L., Kraska, T., Xiaozhou, Li, Ly, A., and Olston, C. Learned Indexes for a Google-scale Disk-based Database, 2020.

[2] Alhomssi, A., and Leis, V. Leanstore commit, 2022. `https://github.com/leanstore/leanstore/commit/d3d8314`.

[3] Apache Software Foundation. Apache HBase, 2008. `https://hbase.apache.org`.

[4] Axboe, J. fio, 2022. `https://fio.readthedocs.io/en/latest/`.

[5] Bingmann, T. TLX: Collection of Sophisticated C++ Data Structures, Algorithms, and Miscellaneous Helpers, 2018. `https://panthema.net/tlx`, retrieved Sep. 13, 2021.

[6] Bloom, B. H. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM 13*, 7 (jul 1970), 422–426.

[7] Bronson, N., Amsden, Z., Cabrera, G., Chakka, P., Dimov, P., Ding, H., Ferris, J., Giardullo, A., Kulkarni, S., Li, H., Marchukov, M., Petrov, D., Puzar, L., Song, Y. J., and Venkataramani, V. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC'13)* (2013).

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

[9] Cao, Z., Dong, S., Vemuri, S., and Du, D. H. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST'20)* (2020).

[10] Chandramouli, B., Prasaad, G., Kossmann, D., Levandoski, J. J., Hunter, J., and Barnett, M. FASTER: A concurrent key-value store with in-place updates. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15,*

2018 (2018), G. Das, C. M. Jermaine, and P. A. Bernstein, Eds., ACM, pp. 275–290.

[11] Chandramouli, B., Prasaad, G., Kossmann, D., Levandoski, J. J., Hunter, J., and Barnett, M. FASTER: an embedded concurrent key-value store for state management. *Proc. VLDB Endow. 11*, 12 (2018), 1930–1933.

[12] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06)* (2006).

[13] Chatterjee, S., Jagadeesan, M., Qin, W., and Idreos, S. Cosine: A cloud-cost optimized self-designing key-value storage engine. *Proc. VLDB Endow. 15*, 1 (2021), 112–126.

[14] Chen, F., Lee, R., and Zhang, X. 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)* (2011).

[15] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC'10)* (2010).

[16] Corporation, I. Intel dc p4510, 2017. `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`.

[17] Dai, Y., Xu, Y., Ganesan, A., Alagappan, R., Kroth, B., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. From WiscKey to Bourbon: A learned index for log-structured merge trees. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020* (2020), USENIX Association, pp. 155–171.

[18] Dayan, N., Athanassoulis, M., and Idreos, S. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017* (2017), S. Salihoglu, W. Zhou, R. Chirkova, J. Yang, and D. Suciu, Eds., ACM, pp. 79–94.

[19] Dayan, N., and Idreos, S. The log-structured merge-bush & the wacky continuum. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019* (2019), P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska, Eds., ACM, pp. 449–466.

[20] Dayan, N., and Twitto, M. Chucky: A succinct cuckoo filter for lsm-tree. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021* (2021), G. Li, Z. Li, S. Idreos, and D. Srivastava, Eds., ACM, pp. 365–378.

[21] Dillinger, P. C., and Walzer, S. Ribbon filter: practically smaller than bloom and xor. *CoRR abs/2103.02515* (2021).

[22] Ding, J., Minhas, U. F., Yu, J., Wang, C., Do, J., Li, Y., Zhang, H., Chandramouli, B., Gehrke, J., Kossmann, D., and et al. ALEX: An Updatable Adaptive Learned Index. *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (May 2020).

[23] Facebook, Inc. RocksDB Tuning Guide, 2020. https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide.

[24] Facebook, Inc. Rocksdb v6.14.6, 2020. https://github.com/facebook/rocksdb/releases/tag/v6.14.6.

[25] Facebook, Inc. Rocksdb, 2021. https://rocksdb.org.

[26] Facebook, Inc. Rocksdb prefix seek, 2021. https://github.com/facebook/rocksdb/wiki/Prefix-Seek.

[27] Facebook, Inc. Universal compaction, 2021. https://github.com/facebook/rocksdb/wiki/Universal-Compaction.

[28] Galakatos, A., Markovitch, M., Binnig, C., Fonseca, R., and Kraska, T. FITing-Tree: A Data-Aware Index Structure. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD'19)* (2019).

[29] Godard, S. iostat, 1999. https://github.com/sysstat/sysstat.

[30] Google, Inc. Leveldb, 2011. https://github.com/google/leveldb.

[31] Google, Inc. S2 geometry, 2022. https://github.com/google/s2geometry.

[32] He, J., Kannan, S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. The Unwritten Contract of Solid State Drives. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys'17)* (2017).

[33] Idreos, S., Dayan, N., Qin, W., Akmanalp, M., Hilgard, S., Ross, A., Lennon, J., Jain, V., Gupta, H., Li, D., and Zhu, Z. Design continuums and the path toward self-designing key-value stores that know and learn. In *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings* (2019), www.cidrdb.org.

[34] Intel Corporation. Intel Xeon Gold 6230 CPU, 2019. `https://ark.intel.com/content/www/us/en/ark/products/192437/intel-xeon-gold-6230-processor-27-5m-cache-2-10-ghz.html`.

[35] Intel Corporation. Intel Optane Technology, 2021. `https://www.intel.ca/content/www/ca/en/architecture-and-technology/intel-optane-technology.html`.

[36] Kipf, A., Marcus, R., van Renen, A., Stoian, M., Kemper, A., Kraska, T., and Neumann, T. SOSD: A Benchmark for Learned Indexes, 2019.

[37] Kraska, T., Beutel, A., Chi, E. H., Dean, J., and Polyzotis, N. The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD'18)* (2018).

[38] Leis, V., Haubenschild, M., Kemper, A., and Neumann, T. LeanStore: In-memory data management beyond main memory. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018* (2018), IEEE Computer Society, pp. 185–196.

[39] Leis, V., Kemper, A., and Neumann, T. The adaptive radix tree: Artful indexing for main-memory databases. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013* (2013), C. S. Jensen, C. M. Jermaine, and X. Zhou, Eds., IEEE Computer Society, pp. 38–49.

[40] Leis, V., Scheibner, F., Kemper, A., and Neumann, T. The ART of Practical Synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware (DaMoN'16)* (2016).

[41] Lepers, B., Balmau, O., Gupta, K., and Zwaenepoel, W. Kvell: the design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019* (2019), T. Brecht and C. Williamson, Eds., ACM, pp. 447–461.

[42] Lepers, B., Balmau, O., Gupta, K., and Zwaenepoel, W. Kvell+: Snapshot Isolation without Snapshots. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)* (2020), USENIX Association.

[43] Li, K.-H. Reservoir-sampling algorithms of time complexity o (n (1+ log (n/n))). *ACM Transactions on Mathematical Software (TOMS) 20*, 4 (1994), 481–493.

[44] Lu, L., Pillai, T. S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. WiscKey: Separating keys from values in ssd-conscious storage. In *14th USENIX Conference on File and Storage Technologies, FAST 2016, Santa Clara, CA, USA, February 22-25, 2016* (2016), A. D. Brown and F. I. Popovici, Eds., USENIX Association, pp. 133–148.

[45] Luo, C., and Carey, M. J. LSM-based Storage Techniques: A Survey. *The VLDB Journal 29*, 1 (Jul 2019), 393–418.

[46] Marcus, R., Kipf, A., van Renen, A., Stoian, M., Misra, S., Kemper, A., Neumann, T., and Kraska, T. Benchmarking Learned Indexes, 2020.

[47] McAllister, S., Berg, B., Tutuncu-Macias, J., Yang, J., Gunasekar, S., Lu, J., Berger, D. S., Beckmann, N., and Ganger, G. R. Kangaroo: Caching billions of tiny objects on flash. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP'21)* (New York, NY, USA, 2021), Association for Computing Machinery.

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

[49] Pandey, V., Kipf, A., Vorona, D., Mühlbauer, T., Neumann, T., and Kemper, A. High-performance geospatial analytics in hyperspace. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016* (2016), F. Özcan, G. Koutrika, and S. Madden, Eds., ACM, pp. 2145–2148.

[50] Papon, T. I., and Athanassoulis, M. A parametric I/O model for modern storage devices. In *Proceedings of the 17th International Workshop on Data Management on New Hardware, DaMoN 2021, 21 June 2021, Virtual Event, China* (2021), D. Porobic and S. Blanas, Eds., ACM, pp. 2:1–2:11.

[51] Ren, K., Zheng, Q., Arulraj, J., and Gibson, G. Slimdb: A space-efficient key-value storage engine for semi-sorted data. *Proc. VLDB Endow. 10*, 13 (2017), 2037–2048.

[52] Spector, B., Kipf, A., Vaidya, K., Wang, C., Minhas, U. F., and Kraska, T. Bounding the last mile: Efficient learned string indexing. *3rd International Workshop on Applied AI for Database Systems and Applications* (2021).

[53] Taylor, S. J., and Letham, B. Forecasting at scale. *PeerJ Prepr. 5* (2017), e3190.

[54] Xie, Q., Pang, C., Zhou, X., Zhang, X., and Deng, K. Maximum error-bounded Piecewise Linear Representation for online stream approximation. *The VLDB Journal 23*, 6 (2014), 915–937.