From Learning to Recycling: A Log-Structured Learned-Less Index

HERA KOO, Department of Computer Science and Engineering, Sungkyunkwan University, Korea
SUNGHO MOON, Department of Computer Science and Engineering, Sungkyunkwan University, Korea
SANGEUN CHAE, Department of Computer Science and Engineering, Sungkyunkwan University, Korea
WOOK-HEE KIM, Department of Computer Science and Engineering, Konkuk University, Korea
JONGMOO CHOI, Department of Computer Science and Engineering, Dankook University, Korea
BEOMSEOK NAM, Department of Computer Science and Engineering, Sungkyunkwan University, Korea

In this paper, we present *Learned-Less Index* - a learned index built by merging existing linear regression models instead of training new ones from scratch. Learned-Less Index does not require scanning sorted keys, which makes it up to 615 times faster than traditional Greedy-PLR method, although it comes with higher prediction errors. *Learned-Less Index* is particularly effective in LSM tree-based KVStores in highly concurrent environments, where computational resources for learning are often limited. Merging existing linear regression models enables models to be created prior to compaction. This allows queries to benefit from model-based search without falling back to conventional index block searches since models are immediately available for use as soon as new SSTables are created. Additionally, these pre-built merged models can help improve the compaction process. Our experiments show that Learned-Less Index significantly reduces CPU overhead and improves query performance by a large margin in highly concurrent environments.

CCS Concepts: • **Computing methodologies** → **Learning linear models**; • **Information systems** → **Data structures**.

Additional Key Words and Phrases: Learned Indexes, Merging Linear Models, LSM Tree

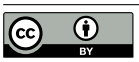
ACM Reference Format:
Hera Koo, Sungho Moon, Sangeun Chae, Wook-Hee Kim, Jongmoo Choi, and Beomseok Nam. 2026. From Learning to Recycling: A Log-Structured Learned-Less Index. *Proc. ACM Manag. Data* 4, 1 (SIGMOD), Article 40 (February 2026), 26 pages. https://doi.org/10.1145/3786654

1 Introduction

Machine learning is extending its impact beyond AI applications to computer systems and database systems [11, 14, 23, 32, 34, 36]. In various computing domains, including database systems, where indexing is a core component, learned indexing has gained attention for its ability to significantly reduce the index size while improving search performance. By learning the distribution of sorted keys, learned indexes can predict key positions, and thereby reduce the number of key comparisons as well as storage accesses.

Learned indexing offers fast search performance but faces challenges with dynamic and concurrent insertions and deletions [13, 23]. To overcome the limitations, various approaches are

Authors' Contact Information: Hera Koo, Department of Computer Science and Engineering, Sungkyunkwan University, Korea; Sungho Moon, Department of Computer Science and Engineering, Sungkyunkwan University, Korea; Sangeun Chae, Department of Computer Science and Engineering, Sungkyunkwan University, Korea; Wook-Hee Kim, Department of Computer Science and Engineering, Konkuk University, Korea; Jongmoo Choi, Department of Computer Science and Engineering, Dankook University, Korea; Beomseok Nam, Department of Computer Science and Engineering, Sungkyunkwan University, Korea.



ACM 2836-6573/2026/2-ART40
https://doi.org/10.1145/3786654

currently being actively explored [11, 13, 14, 23, 38]. In parallel, there has been growing interest in integrating learned indexes into LSM tree-based KVStores [10, 26, 37]. Unlike conventional tree-structured indexes, LSM tree-based KVStores are multi-threaded indexing systems optimized for write-intensive workloads, where background threads perform maintenance tasks such as flushing in-memory data to disk and merge-sorting on-disk data through compaction.

The LSM tree naturally mitigates the limitations of learned index. Specifically, LSM trees address the challenge of dynamic updates through compaction of SSTables, which adhere to a write-once-read-many (WORM) semantic. This allows an immutable learned index to be created and used for each SSTable without complex update mechanisms. Additionally, the LSM tree's support for concurrent access inherently addresses the concurrency limitations of learned indexes. Since Bourbon [10] was proposed as a seminal work in this field, subsequent studies such as TridentKV [26] and LeaderKV [37] have explored the application of learned indexing to LSM trees. We will refer to these systems collectively as *Learned-LSM trees*.

Learned-LSM trees learn the key distribution of SSTables in background. Therefore, it should ideally have minimal impact on foreground client query performance. However, LSM tree-based KVStores employ an artificial governor called the *stop trigger*, which halts foreground threads when too many compaction tasks accumulate [12]. This *write stall* problem has been extensively studied, and insufficient computational resources has been identified as one of the key factors [2–5, 12, 18–20, 24, 31, 39, 40]. If background threads use CPU cycles for additional computations, such as linear regression for learned index, the write stall problem can become even further aggravated.

To address this issue, existing Learned-LSM trees have proposed *offline learning* [10], where learned indexes are only created when KVStores are not processing foreground queries. However, offline learning has limited applicability. Alternatively, *Cost-Benefit Analyzer* (CBA), proposed in Bourbon, creates a learned index only when the performance benefit of using the learned index is greater than the learning overhead. Unfortunately, our analysis shows that CBA identifies performance improvements from the learned index only in a small fraction of cases. As a result, it rarely chooses to create learned models. Furthermore, even when potential benefits are identified, the learning process is time-consuming, often comparable to the duration of compaction itself. Therefore, in Bourbon, flush and compaction tasks trigger learning but does not wait for it to complete, since waiting would considerably slow down compaction and delay the resolution of write stalls. Instead, the learning is delegated to a separate dedicated thread. This asynchronous approach exposes the compacted SSTable to subsequent queries before its learned model becomes available. In our experiments, only 7-14% of queries use the learned models in highly concurrent environments, while the majority still access conventional index blocks.

In this study, we propose a novel learned indexing method - *Learned-Less Index*, which constructs learned indexes by merging existing linear regression models. Unlike previous Learned-LSM trees that learn the distribution of keys through linear regression, Learned-Less Index recycles existing models of victim SSTables and merges them without the need to scan sorted keys. Thereby, Learned-Less Index builds *merged models* 434-615 times faster than the Greedy-PLR (Piecewise Linear Regression) method. However, as a trade-off, merged models exhibit 10–30% of predicted positions with much higher errors than the Greedy-PLR in our experiments. While this high error leads to a 5-23% slowdown in search time compared to the Greedy-PLR method, searching merged models is still 11-44% faster than conventional index block searches.

Taking advantage of the fast model merging process and the fact that it requires no sorted keys, Learned-Less Index allows to construct merged models before compaction and leverages them to accelerate the compaction itself, unlike Bourbon which triggers learning only afterward. In particular, compaction tasks predict the offsets of keys using the merged model, adjust a small number of conflicting offsets through key comparisons, and then copy the key-value data to the

output SSTable. We show that this model-based compaction is faster than traditional compaction, as it reduces the number of key comparisons and CPU usage. Additionally, generating models before compaction allows merged models to be ready for use by queries as soon as new SSTables are generated.

In summary, the key contributions of this study are as follows:

- We propose Learned-Less Index, a lightweight learned indexing technique that constructs merged models by recycling existing ones, thereby saving CPU cycles. This helps minimize additional pressure on the system and avoids the risk of aggravating the write stall problem.
- Using Learned-Less Index, which can be constructed with negligible overhead, we build an LSM tree–based KVStore - *LearnedLessDB*. LearnedLessDB makes merged models available before compaction. This allows learned indexes to be immediately available upon SSTable creation and thereby increase their utilization.
- With models built prior to compaction, compaction tasks can use them during merge-sort to reduce key comparisons and CPU usage. We show that the model-based compaction can improve the write path of LSM trees.

The rest of this paper is organized as follows. In Sections 2 and 3, we present the background and motivation of this study. In Section 4, we present how Learned-Less Index works. In Section 5, we evaluate the performance of LearnedLessDB, and present related work in Section 6. Finally, we conclude the paper in Section 7.

2 Background

2.1 Learned Indexes

Learned indexes are machine learning-based indexing structures that estimate the position of a given key [23], which offer low lookup latency and a small memory footprint by representing large datasets with a small number of line segments. As a prediction-based technique, each lookup must account for an associated error bound. Nonetheless, each lookup operation requires only a single computation followed by a binary search within the small error bound, which can significantly improve read performance.

Although learned indexing has great potential, it does not come without limitations. In particular, learned indexes are not well suited for dynamic workloads, i.e., frequent insertions and deletions. When keys are inserted or deleted, the positions of existing keys change and the model's accuracy may degrade. In the worst case, if keys do not fall within the error bound of the model, a new model that reflects the updated key distribution needs to be created.

To address this limitation, several recursive model indexes (RMIs) have been proposed. AIDEL [25] proposes the use of linked lists instead of shifting array elements when new keys are inserted. ALEX [11], a learned index with a hierarchical structure similar to B+trees, stores a single linear regression model (line segment) for each internal tree node. To enable data insertion without updating the model, ALEX introduces a *gapped array* structure, which allocates extra space between array elements, allowing new data to be inserted without shifting existing elements. This design makes ALEX structurally similar to a radix tree and achieves comparable efficiency. XIndex [35] is another learned index that adopts a two-level structure to support dynamic workloads. When updates occur at the lower level and retraining becomes necessary, it performs split or merge operations.

2.2 Learned-LSM Trees

LSM tree-based KVStores are multi-threaded indexing systems that manage large-scale datasets in both DRAM and disks. They adopt a multi-level structure that uses small in-memory indexes

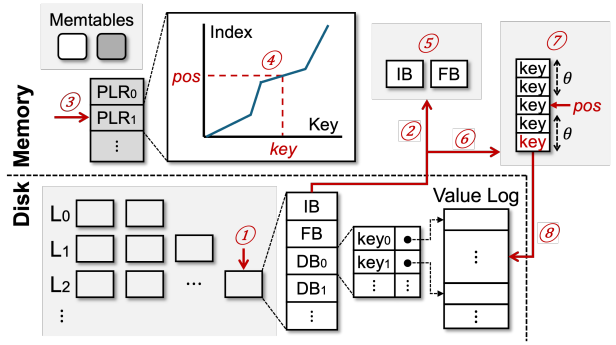


Fig. 1. Bourbon's Architecture and Model Path. ① As in conventional LSM trees, Bourbon finds an SSTable to read. ② It loads the index and filter blocks if they are not cached. ③ & ④ If a trained model is available, it predicts the position range of the search key. ⑤ It checks the filter(s) of the corresponding data block(s). ⑥ If the filter indicates that the key may be present, the data block(s) are loaded. ⑦ & ⑧ Once the key is located, its value is retrieved from the value log.

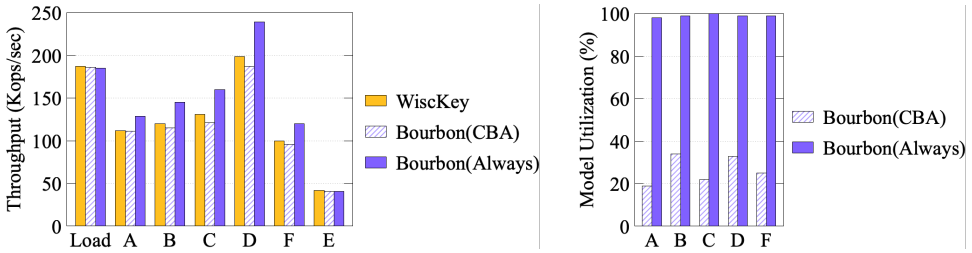
(MemTables) to quickly absorb writes, periodically flushing data to disk (SSTables) and performing merge sort in the background. This design avoids random writes and is optimized for write-intensive workloads. However, the use of multiple levels leads to a trade-off in read performance. Recently, several studies have proposed combining write-optimized LSM trees with read-optimized learned indexes to complement each other's weaknesses [10, 26, 37].

While learned indexes leverage various models of the key distribution, such as linear, polynomial, or even neural networks, to predict key positions [23], most Learned-LSM tree systems adopt linear regression due to its simplicity and efficiency. Our study also focuses on linear regression models.

2.2.1 Bourbon. Bourbon [10] is the first seminal study that employs learned indexing, i.e., Greedy-PLR, in LSM trees. Since SSTables of LSM Trees are immutable, they align well with the static nature of learned index. As an extension of WiscKey [27], a variant of LevelDB[33] that stores variable-length values in a separate data structure called a *value log*, Bourbon makes the key-value entries fixed-size, and simplifies byte offset calculations. When Bourbon accesses an SSTable, it follows one of two lookup paths depending on whether a learned index, trained on that SSTable's key distribution, is available or not, as illustrated in Figure 1. If an SSTable has a learned model, it uses the model to predict which data block contains the search key (referred to as the *model path*). If a learned index is not available for the SSTable, it falls back to the *baseline path*, i.e., it accesses index blocks to locate the target data block.

In Bourbon, only a single compaction thread runs at any given time. In addition to this compaction thread, Bourbon introduces an extra thread dedicated to learning. When the compaction thread places a learning task into a queue, the learning thread dequeues the task and builds a learned model by scanning the SSTable sequentially. Specifically, it transforms key-to-offset mappings to line segments using the Greedy-PLR algorithm. During scanning, if the learning thread detects the addition of the next key-to-offset mapping exceeds a predefined error bound, a new line segment is created. Therefore, each resulting line segment strictly satisfies the predefined error bound.

Although the Greedy-PLR algorithm has linear complexity, the learning overhead is still substantial, accounting for approximately 50% of the compaction overhead. In our experiments using Bourbon, running YCSB A workload on a dataset loaded with 500 million keys (via YCSB Load A), the average learning time (including the I/O time with the block cache enabled, but without



(a) YCSB Throughput (b) Proportion of Model Path

Fig. 2. Performance of Bourbon. (Single Client, Single Compaction Thread, Single Learning Thread)

including the queueing delay) for 2 MB SSTable is 8.76 msec while it takes 16.8 msec for compaction to generate and write a 2 MB SSTable.

To reduce the learning overhead, Bourbon employs several optimizations. One approach is to *delay model creation* for a small period[1] after compaction generates an SSTable, even if the learning thread is idle. This delay ensures that learning is performed only on SSTables with a sufficiently long lifetime, such that unnecessary work on soon-to-be-deleted SSTables can be avoided. Bourbon also proposes *offline learning*, where model creation is deferred until the system becomes idle, such that model creation is not on the critical path of user query processing. In addition, Bourbon incorporates *Cost-Benefit Analyzer* (CBA) to selectively create models based on a comparison between learning costs and expected performance gains. CBA maintains lookup statistics for each SSTable, updated by client threads. When a new SSTable is created, CBA uses these statistics to decide if creating a model is worthwhile.

2.2.2 TridentKV and LeaderKV. TridentKV [26] is another Learned-LSM tree, which is built on top of RocksDB [29] and constructs a two-level RMI for each SSTable. The first layer of the RMI consists of PLR models partitioned by key range, while the second layer comprises multiple linear regression models that learn the key-to-offset mapping within each partition. TridentKV introduces an *adaptive learning* technique that adjusts the learning strategy based on workload characteristics. Under write-intensive workloads, it follows the same approach as Bourbon, i.e., it creates a model using a separate learning thread once compaction is complete. In contrast, under read-intensive workloads, learned models are created by the flush and compaction thread itself, despite it slows down compacton. During compaction, it performs a merge-sort to obtain a sorted array of keys, builds an RMI model based on the key distribution, and then determines the key ranges for each data block according to the model before writing them to the SSTable file. While this approach improves the model's accuracy to 100%, we observe that it slows down background jobs and severely degrades performance.

LeaderKV [37] is another Learned-LSM tree design in which each SSTable is associated with a three-level RMI model. Similar to Bourbon, it creates RMI models in a background learning thread when the compaction thread creates an SSTable. It replaces SSTables with *DKTables*, which store keys and values in separate table spaces to reduce disk access costs. Its three-level RMI structure consists of precise models in the top two levels and approximate models at the leaf level. Another key design of LeaderKV is the *redirect* mechanism at each level of RMI, which prevents unnecessary data block accesses caused by mispredictions.

[1]Bourbon sets this delay to 50 msec, given that the maximum learning time for 4 MB SSTables is 40 msec on their testbed [10].

3 Motivation

3.1 Learning Overhead

3.1.1 Learning in Low-Concurrency Configuration. The techniques proposed by Bourbon to reduce learning overhead are far from satisfactory. First, Bourbon's CBA rarely triggers learning during workload execution. On our testbed machine described in Section 5.1, we load a 500 GB dataset and run 10 million queries in each YCSB workload using WiscKey and Bourbon, as shown in Figure 2. All YCSB experiments use 16B string keys and 64B values. Following the same configuration used in the Bourbon paper, string keys are transformed into 8B integer keys for Greedy-PLR. Bourbon(Always) denotes the performance where we disable CBA such that Bourbon triggers learning for all SSTables. Since Bourbon does not support multi-client execution, all experiments were conducted with a single client thread. For the same experiments, Figure 2(b) presents the percentage of lookup operations that use the learned models. All experiments were conducted with offline learning disabled.

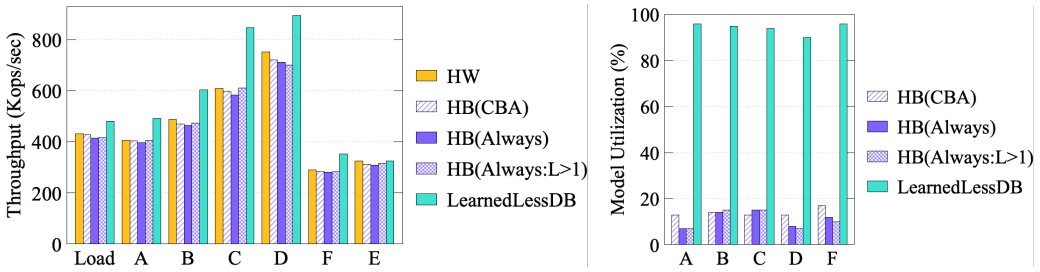
The results show that Bourbon(CBA) does not outperform WiscKey across all workloads. This is in part because Bourbon's CBA selectively triggers learning during execution. In total, less than 34% of lookups are served using learned indexes. The remaining lookups fall back to the baseline path, i.e., index block lookups, as in WiscKey. Another source of the issue is the locking mechanism, i.e., CBA collects statistics to determine whether learning should be triggered. Since a lock is used when accessing the statistics, contention between the client, the compaction, and the learning thread causes performance degradation.

Interestingly, in the read-only workload C, the number of queries that utilize learned models is even smaller than workload B when CBA is enabled. This is because, when using CBA, models are not generated during the write-intensive Load phase, and most of the models built during workloads A and B are for SSTables in level 0. These L0 SSTables are quickly deleted through compaction. Ironically, the workload C that would benefit most from learned indexes ends up using them only to a limited extent.

In contrast, Bourbon(Always) eventually generates learned indexes for all SSTables, even if learning is delayed. With a single client and a single compaction thread, a single learning thread can prepare learned models without incurring significant delay. As a result, it enables 99–100% of lookups to use the learned models and achieves up to 22% higher throughput than WiscKey.

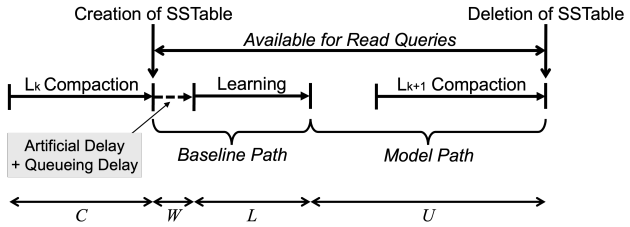
This experiments lead to two key observations: (1) Bourbon's CBA is ineffective in practice and fails to create sufficient learned models during workload execution. (2) In a single-threaded, LevelDB-based environment, compaction leaves sufficient computing resources unused, which can be leveraged to generate learned models for all SSTables without noticeable overhead. However, we focus on the fact that its low-concurrency configuration limits the performance benefit from learned index, as the system's throughput is bottlenecked.

3.1.2 Learning in High-Concurrency Configuration. To evaluate the effect of learned index in high-concurrency environments, we ported Bourbon into HyperLevelDB [17], a high-performance variant of LevelDB that enables concurrent reads and writes through fine-grained locking. However, unlike RocksDB, HyperLevelDB still performs compaction using a single thread, which limits its scalability. To address this, we ported the multi-threaded compaction mechanism of RocksDB into HyperLevelDB and further extended it by enabling multiple learning threads. We also replaced the locking mechanism used for managing CBA statistics with a lock-free implementation using atomic CAS instructions. We refer to this high-concurrency version of Bourbon as *HyperBourbon* (denoted as `HB` in Figures). We also ported WiscKey in the same manner, and refer to it as *HyperWiscKey* (denoted as `HW` in Figures).

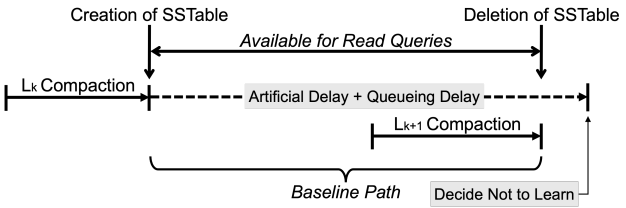


(a) YCSB Throughput (b) Proportion of Model Path

Fig. 3. Performance in High-Concurrency Environments. (32 Clients, 8 Compaction Threads, 4 Learning Threads in HyperBourbon, 1 Learning Thread in LearnedLessDB, 16MB MemTables)



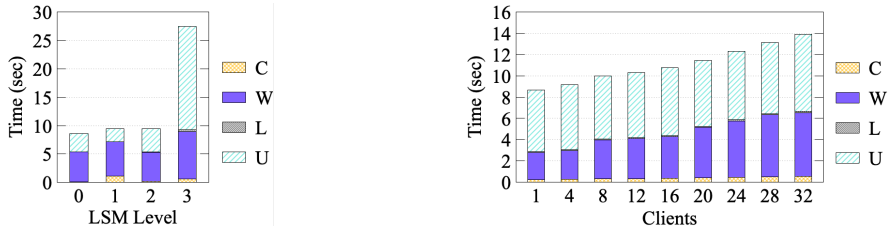
(a) Expected SSTable lifespan. Most queries are processed following the model path. *C: Time spent building the SSTable. W: Time spent waiting in the learning queue. L: Time spent on learning. U: Time when the model is available for use.*



(b) Actual SSTable lifespan. SSTable is often deleted before learning is complete.

Fig. 4. Lifespan of SSTable.

In the experiments shown in Figure 3, we run YCSB benchmark using 32 clients and measure the throughput and the proportion of queries that follow the model path in each KVStore. We set the number of compaction threads and learning threads to 8 and 4, respectively. Due to the increased concurrency and optimizations of HyperLevelDB codebase, all KVStores demonstrate improved performance. However, similar to the results shown in Figure 2, HyperBourbon(CBA) rarely triggers learning. Specifically, it constructs learned models for 120 out of 42,033 SSTables during the Load phase, which accounts for only 0.285%. Therefore, it performs similarly to the baseline HyperWiscKey. Interestingly, it is noteworthy that there is a significant decrease in the proportion of queries that follow the model path in HyperBourbon(Always), sometimes even lower than HyperBourbon(CBA) whose model utilization also declines. Specifically, in HyperBourbon(Always), only 11% of queries use learned indexes on average, despite it attempts to generate learned indexes for all SSTables. This is because learning is delayed in high-concurrency configurations. To address excessive learning overhead, HyperBourbon(Always:L>1) skips model creation for short-lived



(a) Lifespan Breakdown across LSM Tree Level (32 Clients) (b) Lifespan Breakdown w/ Varying Number of Client Threads

Fig. 5. SSTable Lifespan Breakdown. (HyperBourbon(Always), 4 Learning Threads, The total runtime of the experiment with 32 clients: 1226 seconds)

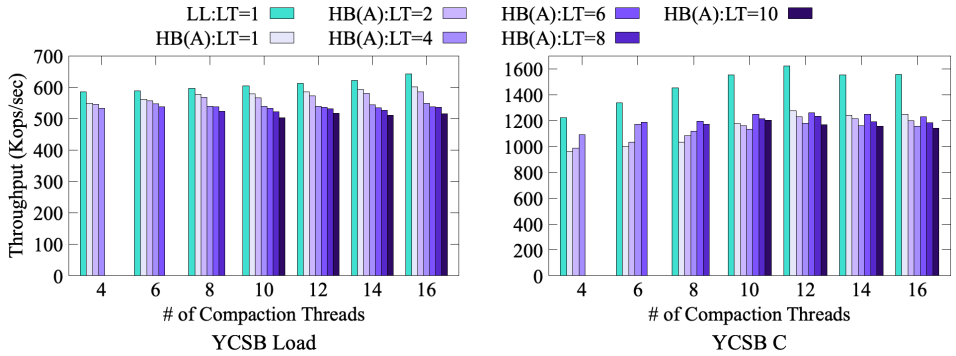


Fig. 6. Throughput Sensitivity to Compaction and Learning Thread Configuration. (32 client threads, HB(A): HyperBourbon(Always), LL: LearnedLessDB, LT: # of Learning Threads)

SSTables in L0 and L1, such that the number of models to be created is reduced. However, it still fails to alleviate the learning backlog, as many models are still required in upper levels.

Figure 4(a) illustrates the ideal lifespan of an SSTable as intended by Bourbon. After an SSTable is created via compaction, it waits for a certain period before creating a learned index model. Lookups during this period are served through the baseline path, and once a learned index model is generated, subsequent lookups access the SSTable via the model path. Later, the SSTable is selected as a victim SSTable for compaction and removed, and the associated learned index model is discarded as well.

However, under high-concurrency settings, only a small fraction of SSTables benefit from the model path because learning takes a long time and the queueing delay continues to grow. Therefore, learning often does not complete before the SSTable is deleted by subsequent compaction, as shown in Figure 4(b). Consequently, most lookups are served through the baseline path.

To quantify this observation, we divide the lifespan of SSTables into the four phases - C: Time for creating the SSTable, W: Time spent waiting in the queue for model creation to start, i.e., queueing delay, L: Time spent for learning, and U: Time the model is used. When we set the number of learning threads to one in HyperBourbon(Always), the waiting time (W) dominates the lifespan and model training lags behind compaction, causing most SSTables to be deleted before learning completes.

In the experiments shown in Figures 5(a) and 5(b), we set the number of learning threads of HyperBourbon(Always) to four. Still, even with four learning threads, the model usage time (U) accounts for only 52% of the total lifespan with 32 clients. Notably, the model usage time for SSTables at lower levels of the LSM tree is even shorter than the wait time. Since these lower-level

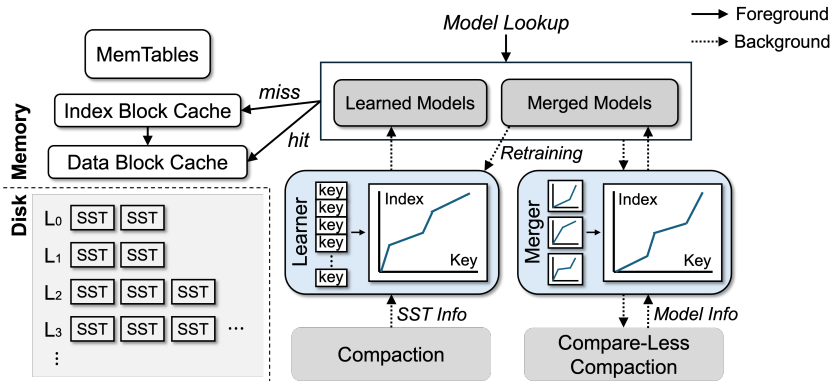


Fig. 7. The Architecture of LearnedLessDB.

SSTables are accessed more frequently than upper-level SSTables, this waiting time should be reduced to increase the utilization of learned models.

Figure 6 shows the performance impact of adjusting the number of compaction and learning threads to reduce the wait time of learning tasks. The number of client threads is fixed at 32. Each experiment loads a 200 million key-value dataset, followed by 10 million lookup queries. Unfortunately, increasing the number of learning threads causes CPU resource contention, which degrades write throughput, while offering marginal improvements in read throughput only when the number of compaction threads is small.

3.2 Motivations of Learned-Less Index

Availability of Learned Index: While previous Learned-LSM tree studies have demonstrated the potential of improving read performance of LSM trees using learned indexes, our experiments reveal that generating and maintaining learned models incurs non-trivial overhead. In particular, initiating model generation only after compaction often fails to prepare the learned models in time. As a result, the system pays the cost of learning, but read queries receive little to no performance benefit. This observation leads to the key question of this study: *Motivation 1: How can we reduce the cost of building learned indexes in LSM trees?*

Use of Learned Index in Write Path: This limitation also raises a broader question regarding the role of learned index in LSM trees. Previous research on Learned-LSM trees has primarily focused on improving search performance through the use of learned index, and they have neglected the impact on write performance. However, in conventional LSM trees, the write stall problem has been pointed out as one of the most critical performance bottlenecks, and numerous studies [2–5, 12, 18, 19, 24, 39, 40] have sought to address this issue. Meanwhile, RMI-based learned indexes, such as ALEX [11], have attempted to accelerate write operations by predicting the positions of inserted keys. However, the use of learned index within the write path remains rather unexplored in LSM trees. This leads to the second question: *Motivation 2. How can learned index be used to improve write performance in LSM trees?*

4 Design of LearnedLessDB

Figure 7 shows the overall architecture of LearnedLessDB. LearnedLessDB employs two background engines for learned index model construction: the *Learner* and the *Merger*. The choice between the two engines depends on whether all victim SSTables selected for compaction have learned index models or not.

Algorithm 1 Learned-Less Merge Algorithm

Require: Input SSTables SST_n of level n, SST_{n+1} of level $n+1$
Ensure: Merged PLR model $\mathcal{S}_{out}$ for output SSTable

$\mathcal{S}_n = S_1, S_2, \ldots, S_k \leftarrow GetModel(SST_n)$ ▷ Set of line segments
$\mathcal{S}_{n+1} = S_{1'}, S_{2'}, \ldots, S_{k'} \leftarrow GetModel(SST_{n+1})$
$B \leftarrow \emptyset$ ▷ Set of key range boundaries
for all $S_i \in \mathcal{S}_n, \mathcal{S}_{n+1}$ **do**
  if $i = 1$ **then**
    $B \leftarrow B \cup \{S_i.start\_key, S_i.end\_key\}$
  else
    $B \leftarrow B \cup \{S_i.end\_key\}$
  end if
end for
Sort B in ascending order
$\mathcal{S}_{out} \leftarrow \emptyset$ ▷ Initialize the merged model
$x_{start} \leftarrow B[0]$ ▷ The beginning of the current interval
$y_{start} \leftarrow 0$ ▷ The starting y-position
$merge \leftarrow false$
for $j = 1$ to $|B| - 1$ **do**
  $x_{end} \leftarrow B[j]$
  if not $merge$ **then**
    $count \leftarrow 0$
  end if
  for all $S_i \in \mathcal{S}_n, \mathcal{S}_{n+1}$ **do**
    $count \leftarrow count + NumKeysInRange(S_i, x_{start}, x_{end})$
  end for
  if $count \leq minNumKeys$ **then** ▷ Merge too short intervals
    $merge \leftarrow true$
    continue
  end if
  $y_{end} \leftarrow y_{start} + count$
  Compute slope $a = \frac{y_{end} - y_{start}}{x_{end} - x_{start}}$
  Compute intercept $b = y_{start} - a \cdot x_{start}$
  Add line segment $(a, b, x_{start}, x_{end}, y_{end})$ to $\mathcal{S}_{out}$
  $x_{start} \leftarrow x_{end} + 1$
  $y_{start} \leftarrow y_{end} + 1$
  $merge \leftarrow false$
end for
return $\mathcal{S}_{out}$

If any victim SSTable lacks a model, the *Learner* is used. In this case, the compaction thread reads the keys from victim SSTables, performs a merge sort, generates new output SSTables, and places learning tasks into the learning queue. Also, when a new SSTable is created through MemTable flush, its learning task is inserted into the queue. The *Learner* then dequeues a learning task, reads the keys from the newly created SSTables, and constructs learned models, as in Bourbon.

In contrast, if all victim SSTables already have models, the *Merger* combines the learned index models from the victim SSTables to produce a new model for the output SSTable (Section 4.1). Using the merged model, LearnedLessDB then performs compaction to create output SSTables. Since the model is created in advance, the output SSTables can be immediately accessed via the model search path upon creation (Section 4.2). However, because the merged models are not created

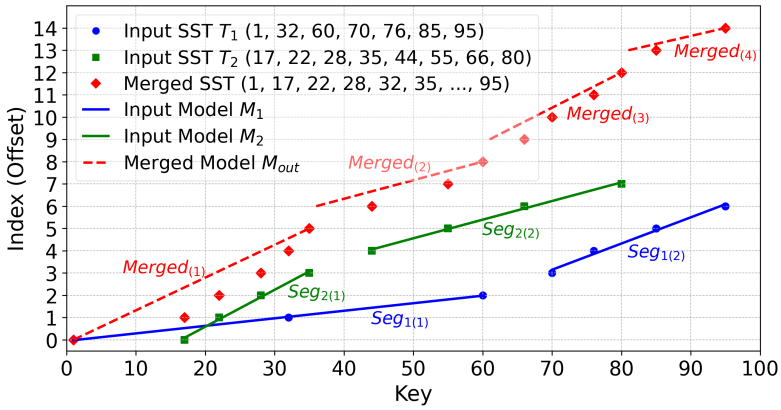


Fig. 8. An Example of Merging Two Models.

based on actual key distribution, their error bounds can be high. (Section 4.3). To mitigate this, LearnedLessDB performs retraining when a prediction error is too high (Section 4.4).

4.1 The Merger

The Merger combines the Greedy-PLR models of victim SSTables to generate *merged models* for output SSTables. If models of the victim SSTables are available, the output key distribution can be approximated without reading all keys from the victim SSTables.

At a high level, the overall logic of the algorithm shown in Algorithm 1 works as follows: First, the start and end points of the key ranges from the line segments in the input SSTables' models are sorted to divide the key space into finer-grained intervals. For each interval, we estimate the number of keys within that interval to determine the slope of the corresponding line segment. The resulting model approximates the one that would have been obtained through the Greedy-PLR.

The detailed algorithm is as the following. First, the x-coordinate boundaries (keys) from all line segments are collected (lines 1–10). For the first line segment in each model, the start and end keys are collected. For subsequent line segments, only the end keys are collected. These boundaries are where piecewise changes occur. Hence, the boundaries are sorted to get ordered intervals (line 11).

Then, for each consecutive pair of boundary points (x_{start}, x_{end}) (lines 16–35), the number of keys falling in the current interval $[x_{start}, x_{end}]$ is estimated using each line segment S_i (lines 21–23). The sum of these counts estimates the number of merge-sorted keys in the current interval in the output model.

If the interval contains too few keys, it is merged with the next interval (lines 24–27). Using the number of keys, the y-coordinate at the end of this interval is calculated, which determines the slope.

Given the two PLR segments over the interval $[x_{start}, x_{end}]$:

$$S_1 : \quad y = a_1 x + b_1 \quad \text{with error} \le \epsilon_1 \quad \text{over } [x_{start}, x_{end}] \tag{1}$$

$$S_2 : \quad y = a_2 x + b_2 \quad \text{with error} \le \epsilon_2 \quad \text{over } [x_{start}, x_{end}], \tag{2}$$

the estimated number of keys in each line segment is:

$$\Delta y_1 = |a_1 (x_{end} - x_{start})| \tag{3}$$

$$\Delta y_2 = |a_2 (x_{end} - x_{start})| \tag{4}$$

Using these estimated numbers, the slope a_m and intercept b_m of the merged model in interval $[x_{start}, x_{end}]$ are calculated as the following.

$$a_m = \frac{\Delta y_1 + \Delta y_2 - 1}{x_{end} - x_{start}} \tag{5}$$

$$b_m = y_{start} - a_m \cdot x_{start} \tag{6}$$

Once the linear function parameters are calculated, the line segment is added to the output model (lines 29–31).

4.1.1 Example of Merging Two Models. Figure 8 illustrates an example of model merging. The input SSTables, T_1 and T_2, have learned models M_1 and M_2, respectively. Model M_1 consists of two segments: $Seg_{1(1)}$ ($y = 0.034 \cdot x - 0.05$) covering the interval $1 \le x \le 60$, and $Seg_{1(2)}$ ($y = 0.118 \cdot x - 5.09$) covering $70 \le x \le 95$. Similarly, M_2 contains segments $Seg_{2(1)}$ ($y = 0.166 \cdot x - 2.73$) for $17 \le x \le 35$, and $Seg_{2(2)}$ ($y = 0.084 \cdot x - 0.37$) for $44 \le x \le 80$.

To merge M_1 and M_2, the start key and the end key of each line segment are collected from both models. From M_1, we obtain 1, 60, 95, and from M_2, we obtain 17, 35, 80. After sorting, the resulting boundary list B becomes (1, 17, 35, 60, 80, 95).

These boundary elements define the segment intervals of the merged model M_{out}. For the interval $[1, 17]$, the number of estimated keys from $Seg_{1(1)}$ is $\Delta y_{Seg_{1(1)}} = |0.034 \cdot (17 - 1)| = |0.544| = 1$.

If the number of keys in an interval is too small, it is merged with the subsequent interval to avoid creating too many line segments with very small key counts. Although this may increase prediction error, having too many short segments actually degrades lookup performance because the more segments the PLR model manages, the longer it takes to locate the correct segment for a given key.

After the first interval is expanded from $[1, 17]$ to $[1, 35]$, the numbers of keys in $Seg_{1(1)}$ and $Seg_{2(1)}$ are estimated as $\Delta y_{Seg_{1(1)}} = |0.034 \cdot (35-1)| = |1.156| = 1$, and $\Delta y_{Seg_{2(1)}} = |0.166 \cdot (35-17)| = |2.988| = 3$, respectively. For the first interval that includes the start key, $\Delta y_{Seg_{1(1)}}$ and $\Delta y_{Seg_{2(1)}}$ need to be incremented by one. Therefore, the slope and intercept of the merged line segment, $Merged_{(1)}$, are calculated as $a_m = (2 + 4 - 1)/(35 - 1) = 0.147$ and $b_m = 0 - 0.147 \cdot 1 = -0.147$, respectively, according to Equations (5) and (6).

For the second interval $[36, 60]$, the numbers of estimated keys from $Seg_{1(1)}$ and $Seg_{2(2)}$ are $\Delta y_{Seg_{1(1)}} = |0.034 \cdot (60-36)| = |0.816| = 1$, and $\Delta y_{Seg_{2(2)}} = |0.084 \cdot (60-36)| = |2.016| = 2$, respectively. Then, the slope and the intercept of $Merged_{(2)}$ are computed as $a_m = (1+2-1)/(60-36) = 0.08333$, and $b_m = 6 - 0.08333 \cdot 36 = 3.00012$, respectively. $Merged_{(3)}$ of interval $[61, 80]$ and $Merged_{(4)}$ of interval $[81, 95]$ can be calculated in the same manner.

In this way, the Merger generates approximate models using simple computations. This method is faster but less accurate than Greedy-PLR. However, we observe that its accuracy still remains usable in practice. Unlike conventional methods that require scanning all keys with a time complexity of O(number of keys), the Merger runs in O(number of segments). Thus, the speedup is defined as the ratio of the number of keys to the number of line segments.

4.1.2 Excluding L0→L1 Compaction from Model Merging. Based on empirical observations, LearnedLessDB disables model merging during L0→L1 compaction by default.[2] This is because we observe that model merging at L1 provides little performance benefit. As merging eliminates learning tasks at upper levels, the learning thread can train models for nearly 99% L0 and L1 SSTables. Therefore, enabling model merging at L1 only marginally increases the fraction of merged-model lookups. Besides, it does not result in noticeable performance gains since merged models are less accurate

[2]Our implementation provides an option to enable it.

Algorithm 2 Compaction using the Merged Model

Input: Input SSTables SST_n of level n and its model M_n, SST_{n+1} of level $n+1$ and its model M_{n+1}
Output: mergedKeys[] for output SST

```
mergedSegments ← merge(M_n, M_{n+1})
Initialize mergedKeys[], mergedIdx ← 0, lastSegmentEnd ← 0
for each segment in mergedSegments do
    Clear bufKeys[]
    for each key in SST_n within the segment range do
        predicted ← floor(slope × key + b) - lastSegmentEnd
        while bufKeys[predicted] is occupied do
            predicted++
        end while
        bufKeys[predicted] ← key
    end for
    bufIdx ← 0
    for each key in SST_{n+1} within the segment range do
        predicted ← floor(slope × key + b)
        while bufIdx < predicted do                          ▷ Flush earlier buffered keys
            if bufKeys[bufIdx] then
                mergedKeys[mergedIdx++] ← bufKeys[bufIdx]
            end if
            bufIdx++
        end while
        if bufKeys[predicted] then                           ▷ collision, key comparison
            if key == bufKeys[bufIdx] then
                mergedKeys[mergedIdx++] ← bufKeys[bufIdx++]
            else if key < bufKeys[bufIdx] then
                mergedKeys[mergedIdx++] ← key
            else
                while bufKeys[bufIdx] and bufKeys[bufIdx] < key do
                    mergedKeys[mergedIdx++] ← bufKeys[bufIdx++]
                end while
                mergedKeys[mergedIdx++] ← key
            end if
        else                                                 ▷ No collision, insert directly
            mergedKeys[mergedIdx++] ← key
            bufIdx++
        end if
    end for
    while bufIdx < size of bufKeys[] do
        if bufKeys[bufIdx] then
            mergedKeys[mergedIdx++] ← bufKeys[bufIdx]
        end if
        bufIdx++
    end while
    lastSegmentEnd ← mergedIdx - 1
end for
return mergedKeys
```

than learned models. Specifically, the high degree of overlap among L0 SSTables and the sparse key

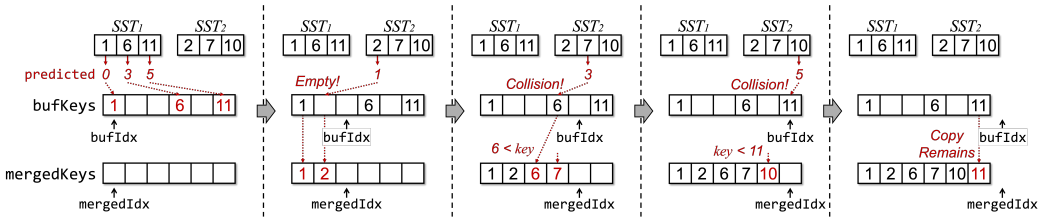


Fig. 9. An Example of Compaction using the Merged Model. *mergedSegments: y=round(0.5x-0.5) over [1,11]*

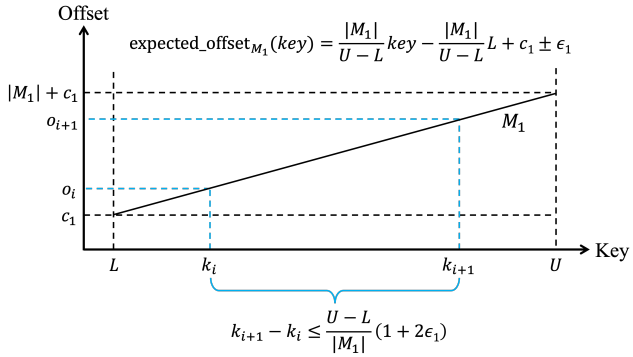
distribution often lead to highly fragmented key intervals and poor model accuracy. In addition, many intervals contain more than two line segments, which increases computational complexity, as multiple segment equations must be evaluated to estimate the number of keys.

4.2 Compare-Less Compaction

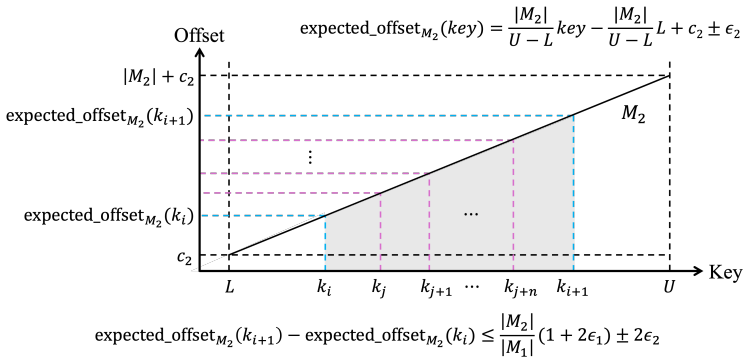
Since the Merger does not require sorted keys, it can generate merged models for output SSTables before compaction begins. This not only enables queries to immediately leverage the models as soon as the SSTables are created, but this also provides an opportunity for compaction threads to leverage the merged model during the write path. While conventional compaction incurs overhead from repeated key comparisons, our *Compare-Less* compaction approach minimizes these comparisons by leveraging the merged model, resulting in a faster and more efficient compaction process.

Algorithm 2 shows how a compaction thread merge-sorts two SSTables (SST_n and SST_{n+1}) using the merged model. The algorithm processes each merged line segment one at a time. Within each segment, it predicts the approximate index (i.e, offset) of each key from one SSTable SST_n using the merged model and places it in a temporary buffer array (`bufKeys[]`). If the predicted position is already occupied by another key, linear probing is used to find an empty slot for insertion (lines 5–11). Once all keys from SST_n within the segment are placed in the buffer array, keys from the other SSTable SST_{n+1} within the same segment interval are iterated through, and they are merged with the keys buffered in `bufKeys[]`. When inserting an SST_{n+1} key at its predicted position, there may be empty slots before that position. To address the issue, all buffered SST_n keys at positions less than the predicted position of the current key are written to the target merged array (lines 15–20). After writing all smaller keys, it checks if the predicted position is available for the current key (line 21). If the predicted position is empty, there is no conflict. Hence, the key is stored in the next position of the merged array. If the position is already occupied by a key from SST_n, the two keys are compared so that the smaller one is placed at an earlier position. If the keys are equal, the key from the lower level SSTable is treated as the updated version, and the other old duplicate is discarded. Once all keys from SST_{n+1} are processed, any remaining SST_n keys in the buffer array are copied to the target merged array. By repeating this process for each merged line segment, the fully sorted keys for the output SSTable are generated.

Figure 9 shows an example of how the algorithm merge-sorts the keys from two SSTables. In the example, the merged line segment $y = 0.5x - 0.5$ is given. The predicted offsets of SST_1's keys $(1, 6, 11)$ are $(0, 3, 5)$ according to the model, so they are inserted into the corresponding positions in a temporary buffer `bufKeys[]` without key comparisons (lines 5–11). Then a cursor `bufIdx` for `bufKeys[]` is initialized (line 12). Next, the predicted position (`predicted`) of each key from SST_2 is calculated (line 14), and the keys in `bufKeys[]` with indices smaller than the predicted position are copied into `mergedKeys[]` in order (lines 15–20). In the example, key 1 in `bufKeys[0]` is copied to `mergedKeys[0]`. If the predicted entry in `bufKeys[]` is already occupied, a conflict occurs and a key comparison is required. However, in the example, SST_2's key 2 has a predicted position of



(a) Input Model M_1. k_i and k_{i+1} are consecutive keys in a sorted array.



(b) Input Model M_2. k_j, k_{j+1}, ..., and k_{j+n} fall within the key range $[k_i, k_{i+1}]$

Fig. 10. Merged Model Error Bound Analysis.

1. Since `bufKeys[1]` is null, key 2 is directly copied to `mergedKeys[1]` (lines 32–34). Next, key 7's predicted position is 3. Because `bufIdx` is still behind, it is incremented to catch up. Then, it checks for whether there is a conflict with predicted position of 3 (line 21). Since `bufKeys[3]` is already occupied by key 6 from SST_1, a key comparison is needed. Since key 7 is larger, key 6 is inserted first. The algorithm also checks if more SST_1 keys follow due to linear probing, but finds the next slot empty. So key 7 is inserted in the next position of `mergedKeys[]` (lines 26–30). Key 10 from SST_2 is predicted to go to offset 5, which also already holds key 11 from SST_1. Since key 10 is smaller, it is inserted into `mergedKeys[]` first (lines 24–25). After all SST_2 keys are processed, `bufIdx` is incremented and the remaining key 11 from `bufKeys[]` is inserted into `mergedKeys[]` (lines 37–42). Finally, `mergedKeys[]` contains all keys from SST_1 and SST_2 in sorted order.

Lemma 4.2.1 (Sorted Output of Compare-Less Compaction). *Given two sorted SSTables SST_n and SST_{n+1}, the Compare-Less Compaction algorithm produces a merged output array* `mergedKeys[]` *in non-decreasing order.*

Proof. We prove the lemma by induction on the output position i. As a preliminary, note that the algorithm places keys from SST_n into their predicted positions in the array `bufKeys[]`, resolving collisions by linear probing to the right.

Base case ($i = 0$). We determine the first element to output into `mergedKeys[0]`. Let p_0 be the predicted position of the first key k_0 from SST_{n+1}. (1) If `bufKeys[]` has any elements with indices $< p_0$, those elements are all $< k_0$ since the model is monotonic. The leftmost among them is the global minimum. (2) Otherwise, if there is a collision at p_0, the smaller key among the colliding elements is the global minimum. (3) If neither of the above holds, the k_0 itself is the global minimum.

Inductive step ($i - 1 \rightarrow i$). Assume `mergedKeys[0..`$i-1$`]` is sorted and contains the i smallest keys. Let k denote the current key from SST_{n+1} and let p_k be its predicted position. To produce the next output at i: (1) If `bufKeys[]` has any elements with indices $< p_k$ that are not yet output, the leftmost such element is the next global minimum. (2) Otherwise, if a collision occurs at p_k, the smaller of the colliding keys is output. (3) If neither holds, k is the next global minimum. In all cases, the key written to `mergedKeys[`i`]` is the smallest among all remaining unmerged keys. □

4.2.1 String Key Conversion Overhead. Learned models require string keys to be converted into integers, which introduces non-negligible overhead. In LearnedLessDB, the Merger only needs the slope and intercept of segments, so once the Learner converts the keys, no further conversion is necessary. Unfortunately, the Compare-Less compaction still requires string keys to be converted during key comparisons. If this conversion is performed on every key access, YCSB throughput drops by up to 6% in our experiments. To eliminate repeated conversions and reduce the overhead, converted keys of each SSTable are cached along with its model in memory for faster access.

4.3 Error Bound of Merged Model

The error bound of the merged model depends on the ratio of the number of keys from the two input models. Figure 10 shows two input models, M_1 and M_2. The slope of a segment is determined by its key range, with the smallest key L, the largest key U, and the number of keys $|M_1|$ in that range. Using this slope, the expected offset of a key is given by:

$$\text{expected\_offset}_{M_1}(key) = \frac{|M_1|}{U-L}key - \frac{|M_1|}{U-L}L + c_1 \pm \epsilon_1 \tag{7}$$

where c_1 is the starting offset and ϵ_1 is the error bound. The error bound prevents keys from deviating from a uniform distribution. Therefore, by inverting the formula, the expected key k_i for a given offset o_i is:

$$k_i = \frac{U-L}{|M_1|}\left(o_i + \frac{|M_1|}{U-L}L - c_1 \pm \epsilon_1\right) \tag{8}$$

Using this formula, the distance between consecutive keys is given by:

$$k_{i+1} - k_i \leq \frac{U-L}{|M_1|}(o_{i+1} - o_i + 2\epsilon_1) = \frac{U-L}{|M_1|}(1 + 2\epsilon_1) \tag{9}$$

When two input models are merged, keys k_j, k_{j+1}, k_{j+2}, ..., from model M_2 may be inserted in between, which shifts the offsets and increases the error. The number of M_2's keys ($|M_2|$) within this range (k_i and k_{i+1}) is estimated as:

$$\text{expected\_offset}_{M_2}(k_{i+1}) - \text{expected\_offset}_{M_2}(k_i) \tag{10}$$

$$= \left(\frac{|M_2|}{U-L}k_{i+1} - \frac{|M_2|}{U-L}L \pm \epsilon_2\right) - \left(\frac{|M_2|}{U-L}k_i - \frac{|M_2|}{U-L}L \pm \epsilon_2\right) \tag{11}$$

$$= \frac{|M_2|}{U-L}(k_{i+1} - k_i) \pm 2\epsilon_2 \tag{12}$$

$$\leq \frac{|M_2|}{U-L}\left(\frac{U-L}{|M_1|}(1 + 2\epsilon_1)\right) \pm 2\epsilon_2 = \frac{|M_2|}{|M_1|}(1 + 2\epsilon_1) \pm 2\epsilon_2 \tag{13}$$

The offset of key k_{i+1} from M_1 may shift by this amount. Symmetrically, the offset of key k_j from M_2 may shift by

$$\frac{|M_1|}{|M_2|}(1 + 2\epsilon_2) \pm 2\epsilon_1. \tag{14}$$

Therefore, the error bound of the merged model is

$$max\left(\frac{|M_2|}{|M_1|}(1 + 2\epsilon_1) \pm 2\epsilon_2, \frac{|M_1|}{|M_2|}(1 + 2\epsilon_2) \pm 2\epsilon_1\right). \tag{15}$$

In LSM trees, compaction often removes duplicate and deleted keys, i.e., tombstones. Therefore, the key distribution in the resulting output SSTable may diverge further. To mitigate this issue, LearnedLessDB adjusts the slope of each segment's model by counting the actual number of keys during compaction. That is, during compaction, merging is performed using models based on the predicted number of keys, while after compaction, the models adjusted with the actual number of keys are used for query processing.

4.4 Lookup and Retraining

LearnedLessDB's lookup process is generally similar to that of Bourbon, but the error bound of a merged model can be orders of magnitude larger than that of the input PLR models, which can degrade lookup performance. Taking advantage of small fixed error bounds, Bourbon performs binary search. In contrast, LearnedLessDB approximates the error bound. Specifically, LearnedLessDB selects the largest error bound among the input models and uses it as the *approximate error bound* (AEB). However, the AEB of a merged model is not a strict error bound, so the predicted key may fall outside the expected range. If a binary search reveals that the current AEB does not contain the predicted key, the algorithm performs a linear scan from either the left or right boundary.

While linear scanning, LearnedLessDB counts how many data blocks are read from an SSTable. If a query reads more than a predefined number of blocks (2 in our implementation), the SSTable is marked for retraining such that the Learner constructs a more accurate Greedy-PLR model. The Learner manages two learning queues: a high-priority queue for regular learning tasks, i.e., learning for newly created SSTables from MemTable flush or L0→L1 compactions, and a low-priority *retraining queue* for tasks triggered due to large error bounds. By distinguishing learning priorities, LearnedLessDB ensures that models for recently written SSTables are trained promptly while improving accuracy and efficiency through retraining during read-intensive workloads.

Our implementation performs retraining at the SSTable level, but selectively retraining only the segments with large prediction errors could offer a better trade-off between model accuracy and training overhead. We leave this improvement as future work.

5 Evaluation

We evaluate LearnedLessDB to answer the following questions. (1) How does LearnedLessDB perform compared to other Learned-LSM trees? (Section 5.2) (2) How effectively does LearnedLessDB reduce learning overhead, and how much does the reduced learning overhead increase model utilization? (Section 5.3) (3) How efficiently does Compare-Less compaction improve compaction performance? (Section 5.4) (4) How accurate are merged models? (Section 5.5)

5.1 Experimental Setup

All experiments are performed on a testbed machine with 2× Intel Xeon Gold 5215 CPUs (20 cores per each CPU), 128 GB DRAM, and an Intel P5800X U.2 SSD. Unless stated otherwise, we run 32

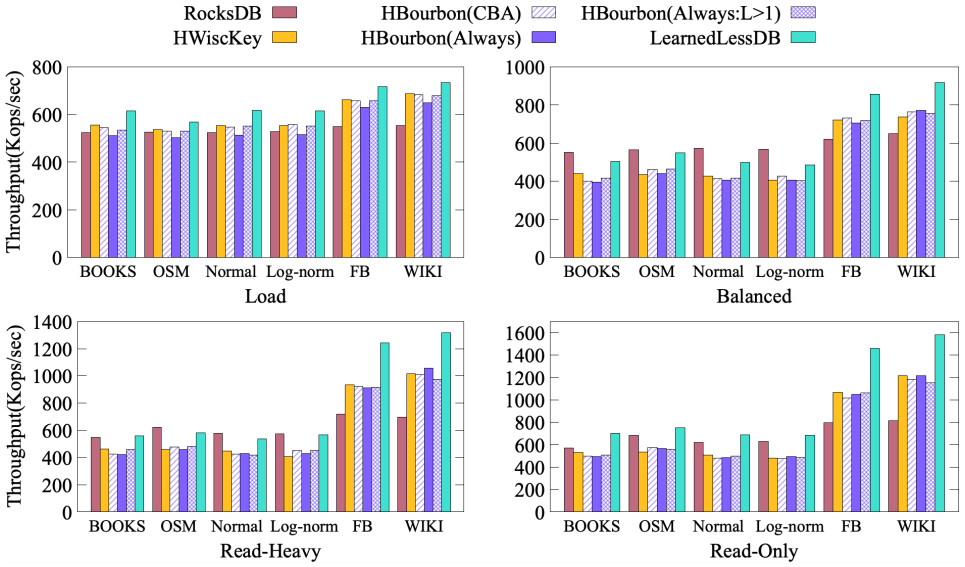


Fig. 11. Comparison of Learned-LSM Trees with SOSD Benchmark.

client and 8 compaction threads for all KVStores except RocksDB, which runs 9 compaction threads. For LearnedLessDB, a single learning thread is used, whereas HyperBourbon employs four.

We use db_bench [1], YCSB [9], and SOSD benchmarks [21, 28] for evaluation. SOSD is a collection of real-world datasets commonly used for the evaluation of learned indexes. Since the SOSD benchmark provides only read-only workloads, we ported the SOSD dataset to YCSB, as was done in [37], in order to generate read-write mixed workloads. For YCSB, the Load phase inserts 500 million key-value pairs into the database, and each workload performs 10 million queries in uniform distribution. We use the 16-byte keys and 64-byte values as in [10], where 16-byte numeric string keys are converted into 8-byte integers. The error bound of learned models is set to 8.

We compare the performance of LearnedLessDB against RocksDB, HyperWiscKey (denoted as HWiscKey), and HyperBourbon (denoted as HBourbon).[3] For a fair comparison, we set both the MemTable and SSTable sizes to 64 MB across all KV stores, consistent with the default configuration used in RocksDB.

Unfortunately, LeaderKV is not an open source, Therefore, we are unable to include its performance results. TridentKV is an open source, but it is highly unstable and works correctly only under specific workloads. Nevertheless, for completeness, we provide a basic performance comparison using db_bench.

5.2 Comparison with Other Learned-LSM trees

In the first set of experiments shown in Figure 11, we run the SOSD workloads using 32 clients to compare the performance of LearnedLessDB with other KVStores. In the Load phase, all workloads except FB and WIKI insert 600 million key-value pairs while the FB and WIKI workloads insert 200 million data. The WIKI dataset contains duplicate keys, so the number of unique keys is around 90.4 million. Balanced (read 50% and write 50%), Read-Heavy (read 90% and write 10%), and Read-Only (read 100%) workloads perform 10 million queries.

[3]Source codes are available at https://github.com/DICL/LearnedLessDB.

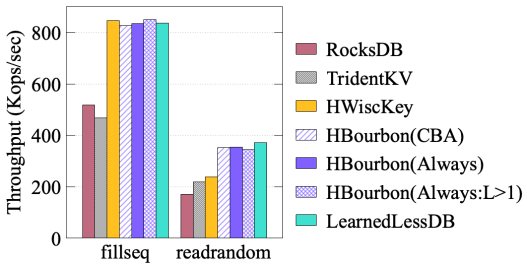


Fig. 12. db_bench Performance. (Single client)

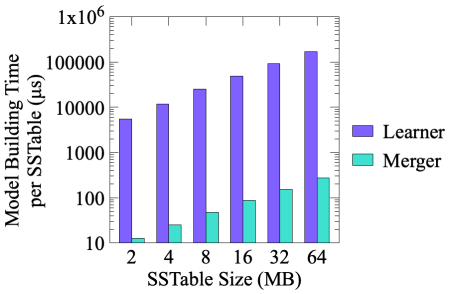


Fig. 13. Model Building Time.

LearnedLessDB consistently outperforms HyperWiscKey and HyperBourbon. In particular, it shows 22-43% and 25-42% higher throughputs than HyperBourbon(CBA and Always) under Read-Only and Read-Heavy workloads, respectively. With relatively small (200 million data) datasets - FB and WIKI, LearnedLessDB shows significantly better performance than all other KVStores, including RocksDB. However, for larger datasets with mixed read/write workloads, LearnedLessDB performs slightly worse than RocksDB. This is not due to learned indexing but rather differences in concurrency control. We plan to improve our implementation of multi-threaded compaction in HyperLevelDB for further optimization.

In the experiment shown in Figure 12, we compare the performance of TridentKV with Learned-LessDB using the `fillseq` and `readrandom` workloads from db_bench. TridentKV works correctly only when keys are inserted in a monotonically increasing order. Since TridentKV misplaces key-values during compaction, over 99% of queries fail to find them. Additionally, it works only when a single client submits queries despite its codebase is RocksDB. Therefore, for this experiments we use a single client thread.

In the `fillseq` workload, the key ranges of SSTables do not overlap, so compaction only updates the level information of SSTables without performing merge-sort. As a result, learned index construction is triggered only when MemTables are flushed to disk. Since merge-sort is not performed, HyperBourbon(CBA), HyperBourbon(Always), and LearnedLessDB all reuse the models learned from L0 SSTables in the upper levels without modification. As a result, they show similar write throughput. It is noteworthy that TridentKV performs worse than RocksDB because it creates learned models during compaction, i.e., learning blocks and slows down compaction tasks. Interestingly, in the single-client setting, HyperWiscKey outperforms RocksDB, which enables L0-to-L0 compaction, by a large margin. This is not related to learned index, but rather appears to stem from differences in the codebase. The write performance of HyperBourbon is comparable to HyperWiscKey, since the learning overhead of HyperBourbon is minimal in the single-client setting.

Since merge-sort does not occur in the `fillseq` workload, the learning overhead is minimized. As a result, in the subsequent `readrandom` workload, HyperBourbon utilizes learned models for most of the SSTables, and its read throughput is about 2× higher than HyperWiscKey. LearnedLessDB also does not provide significant performance advantages. Notably, despite the fact that the models generated via merging are less accurate than those constructed using the Greedy-PLR method, LearnedLessDB still delivers read performance comparable to HyperBourbon.

5.3 Model Construction Time

Figure 13 shows the average model construction time of the proposed Learned-Less method compared to the Greedy-PLR in the YCSB Load phase with varying the size of SSTables. This

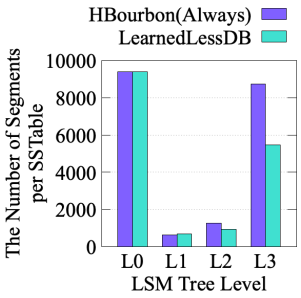


Fig. 14. Model Size.

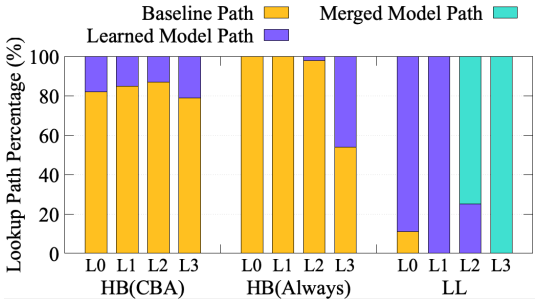
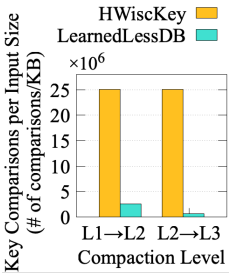
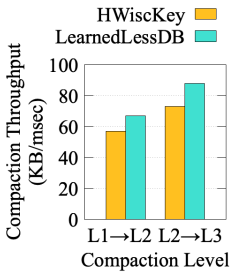


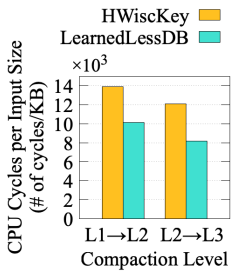
Fig. 15. Lookup Path Ratio. (HB: HyperBourbon, LL: LearnedLessDB)



(a) Key Comparisons.



(b) Compaction Throughput.



(c) CPU Cycles.

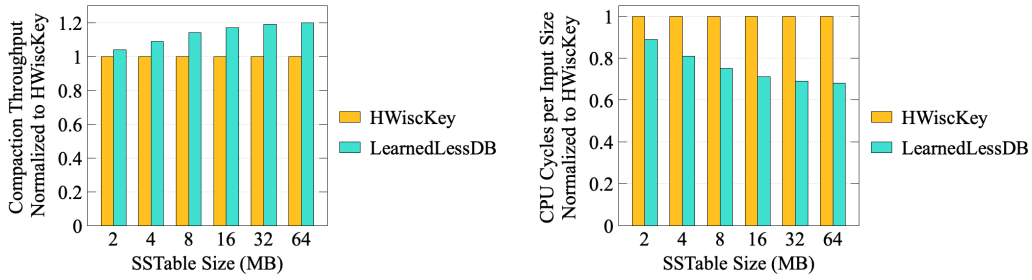
Fig. 16. Reduced Number of Key Comparisons.

measurement excludes merge-sorting and I/O time, and focuses solely on the time taken to construct models. The time to either train or merge learned models is generally proportional to the SSTable size. When increasing the SSTable size from 2MB to 64MB, the model construction time of Greedy-PLR increases by approximately 31×. In contrast, LearnedLessDB's construction time increases by about 22× starting from a significantly lower baseline. As the SSTable size increases, the performance gap between the Learned-Less Index and the Greedy-PLR method widens further. Specifically, LearnedLessDB is 434× faster at 2MB and 615× faster at 64MB.

Figure 14 shows the average number of segments per SSTable at each level, generated during the YCSB Load phase. Level 0 models are built using the Greedy-PLR in both HyperBourbon and LearnedLessDB. Since Level 0 SSTables span a wide key range, the key distribution tends to be highly skewed, resulting in a large number of segments. In contrast, Level 1 SSTables are created through merge-sort, and are split disjointly by key range, which leads to a more uniform key distribution and a significant reduction in the number of segments. In addition, in HyperLevelDB-based KVStores, the key range boundaries of Level-K SSTables (K>0) must be aligned with the key boundaries of Level-(K+1) SSTables. As a result, while the size of SSTables at the bottom level (e.g., L3) is 64MB, the SSTables at the lower levels (e.g., L1 and L2) are often smaller than 64MB.

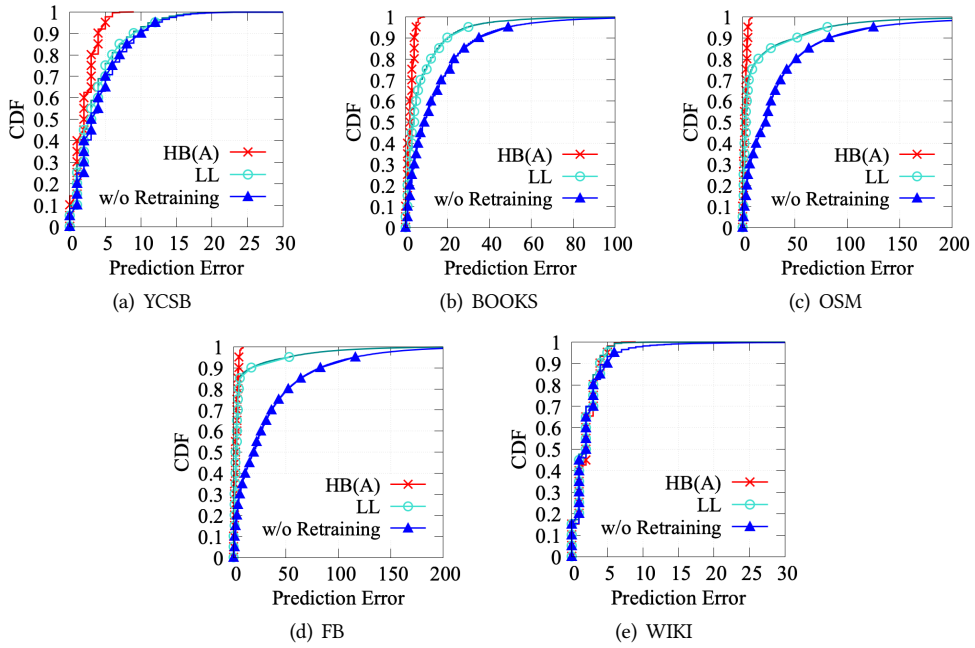
Merged models first appear at Level 2, where segments are generated by merging input intervals, resulting in approximately 1.36x as many segments as in Level 1. Despite having more segments than Level 1, merged models still produce significantly fewer segments than Greedy-PLR models in higher levels at the cost of higher prediction errors. For example, at Level 3, merged models have about 37% fewer line segments compared to Greedy-PLR models.

Model construction performance affects the time window during which the model can be used. Specifically, if learning is slow, more queries use conventional index blocks. In the experiments



(a) Compaction Throughput Normalized to HyperWiscKey. (b) CPU Cycles Normalized to HyperWiscKey.

Fig. 17. Normalized Compaction Throughput and CPU Cycles with Varying SSTable Sizes.



(a) YCSB (b) BOOKS (c) OSM

(d) FB (e) WIKI

Fig. 18. Prediction Error CDF. (HB(A): HyperBourbon(Always), LL: LearnedLessDB)

shown in Figure 15, we run YCSB C workload and measure the proportion of queries that use index blocks, learned models, and merged models. In HyperBourbon(CBA) and HyperBourbon(Always), model training falls behind. Even if learning threads construct PLR models, they are often quickly removed by compaction due to its long queueing delay. This causes most queries to follow the baseline path using index blocks. In contrast, LearnedLessDB constructs merged models for 75% and 100% SSTables in Level 2 and 3, respectively. As a result, the learning thread has more available CPU cycles and can spend them to construct PLR models for Level 0 and Level 1 SSTables without queueing delay. Therefore, LearnedLessDB prepares models for Level 0 and Level 1 SSTables in time, and nearly all lookups can be handled through the model path.

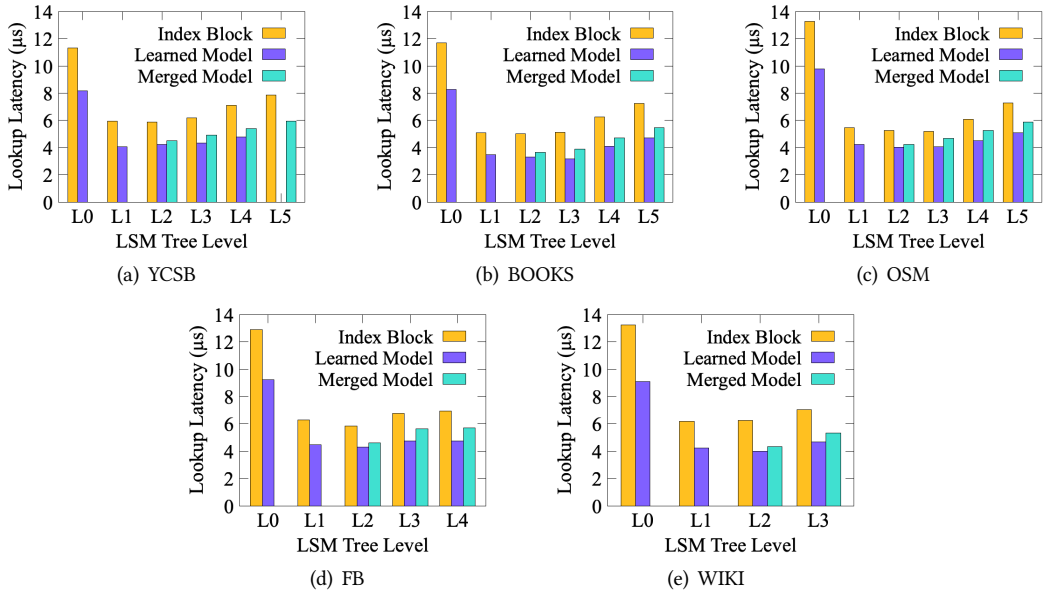


Fig. 19. Lookup Latency across Levels. (Index Block: HyperWiscKey, Model: LearnedLessDB)

5.4 Evaluation of Compare-Less Compaction

In Figure 16, we measure the number of key comparisons performed by compaction tasks during the YCSB Load phase. Since LearnedLessDB performs key comparisons only in the event of collisions, it reduces the number of comparisons by a factor of 9.86 and 39.47 during L1→L2 and L2→L3 compactions, respectively.

While the number of key comparisons drops dramatically, the improvement in *compaction throughput*, i.e., the number of bytes processed by compaction tasks per unit time, is relatively modest as shown in Figure 16(b). This is because a significant portion of the compaction time is spent on I/O operations. This is also because, even though key comparisons are reduced, additional computation is still required to use the models and check whether the predicted position is available or occupied. Figure 16(c) shows the CPU cycles consumed during compaction, and the reduction in CPU usage is roughly proportional to the compaction throughput improvement.

Figure 17(a) shows the normalized compaction throughput as the SSTable size is varied, with results normalized to the throughput of HyperWiscKey. Figure 17(b) presents the number of CPU cycles measured during the same experiments. LearnedLessDB reduces CPU cycles by a factor of 1.13 to 1.47 compared to the baseline. For both compaction throughput and CPU cycles, the performance gap between LearnedLessDB and the baseline widens as the SSTable size increases because the speedup is defined as the ratio of the number of keys to the number of line segments.

5.5 Accuracy of Merged Model

Figure 18 shows the distribution of *prediction errors*, defined as the differences between the predicted and actual positions, based on models generated for the default configuration of YCSB and SOSD workloads. YCSB Load workload stores uniformly distributed keys, whereas the key distribution in SOSD datasets is more complex.

In this experiments, we reduce the size ratio of adjacent levels in the LSM tree from 10 to 4. This smaller ratio increases the number of levels and allows us to observe how repeated model merging

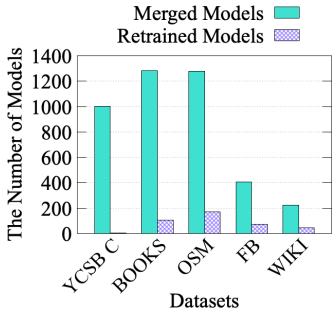


Fig. 20. The Number of Merged and Retrained Models.

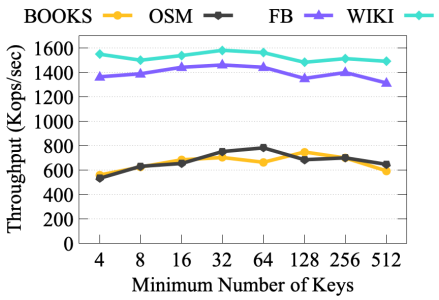


Fig. 21. Search Throughput under Varying Minimum Number of Keys per Merged Segment.

affects the prediction error. Using this configuration, YCSB, BOOKS and OSM workloads result in LSM trees with six levels, i.e., up to Level 5. The configuration labeled `w/o Retraining` refers to a setting where merged models are reused across all levels without retraining. For HyperBourbon(Always), the prediction error is bounded by a fixed threshold of 8. This ensures that its error does not exceed 8 across all workloads.

Interestingly, the WIKI workload exhibits even lower error than YCSB, and merged models still achieve prediction accuracy comparable to that of Greedy-PLR-trained models. This is because the WIKI workload has lots of duplicated keys, which results in fewer compaction and merge operations. However, for other SOSD workloads, particularly OSM, disabling retraining causes more than half of the queries to suffer from prediction errors greater than 10. It is noteworthy that retraining significantly improves model accuracy: about 70–90% of queries benefit from low prediction errors, comparable to those of learned models.

When the number of levels is small (e.g., four levels), retraining is rarely needed. However, as the number of levels increases, retraining becomes more necessary. Interestingly, prediction error does not correlate directly with the level depth. While Levels 0, 1, and 2 consistently show the lowest prediction errors, among the upper levels, the error distribution varies depending on the workload. Specifically, prediction error consistently increases from Levels 2 to 5 in YCSB, BOOKS, and WIKI workloads. In contrast, for OSM and FB, the error at the last level is similar to the penultimate level.

For the same experiments, we measure the average lookup latency at each level when index blocks, learned models, and merged models are used. Figure 19 shows that at L0, lookup latency is high due to the large key range overlaps. Learned models achieve the lowest latency due to their tight error bounds. Since merged models have larger error bounds, they have 5-23% higher latency compared to learned models, but still 11-44% faster than the baseline path. In the FB workload, there is little to no key overlap during L3→L4 compaction, so most SSTables move to the next level without merge-sort. As a result, the latencies at L3 and L4 are similar.

Figure 20 shows the number of merged models generated under the workloads of Figures 18 and 19, along with the number of models that were retrained. Since the YCSB Load workload populates keys in uniform distribution, the merged models achieve high accuracy, resulting in a low retraining ratio of 0.3%. In contrast, real-world datasets with more complex distributions requires retraining for 8–19% of the models.

Figure 21 shows the impact of adjusting the threshold for the minimum number of keys per segment on read throughput. If the minimum key count is too small, many short segments are created, which increases the overhead of locating the correct segment during a search. In contrast, if the minimum key count is too high, the error of the merged segments may become excessive.

Within a reasonable range, performance is insensitive to the threshold. We use 32 as the default value.

6 Related Work

Linear regression models have been extensively studied and widely applied across many domains including learned indexes [8, 15, 41]. Various linear regression model merging techniques such as ARM (Average Regression Model) [6], FIC (Focused Information Criterion) [16], and BMA (Bayesian Model Averaging) [30] have also been proposed over the years. These model merging techniques have been used primarily to improve the accuracy of linear regression models, enhance their stability by reducing variance, and minimize uncertainty in model selection by considering multiple plausible models rather than just one. That is, these model merging techniques that have evolved into ensemble methods in machine learning are fundamentally different from our Learned-Less Index.

To the best of our knowledge, no prior work has explored merging linear regression models for merge-sort, which is required in the context of log-structured learned indexes. The most closely related effort to reduce model construction time is the sampling-based approach [7]. Sample EB-PLA and Sample EB-Histogram are error-bounded sampling techniques that help reduce the index-building time while maintaining competitive lookup performance and index size. While our work focuses on model merging, sampling can be incorporated into L0 and L1 SSTables to further reduce model construction time.

7 Conclusion

In this work, we develop a lightweight merging technique for learned indexes in LSM tree-based KVStores. Merging existing linear regression models significantly reduces computational overhead and avoids the cost of training learned models. Despite being less accurate than fully trained learned models, merged models still provide much faster lookup performance than index block search, Moreover, by generating models prior to compaction, merged models enable the model-basedkcompaction, which leads to improved write performance.

We believe this work opens up new directions for future exploration. While our current approach targets simple linear regression models, one promising direction is to employ more advanced models, such as RadixSpline [22], LIPP [38], or ALEX [11], to further reduce prediction error. Another possible direction would be to apply the model merging technique to other domains where computational resources are constrained.

Acknowledgments

This research was supported in part by Samsung Electronics, and by the National Research Foundation of Korea (NRF) (Grant No. NRF2022R1A2C2091680), and by the Institute of Information & Communications Technology Planning & Evaluation (IITP) (Grant No. RS-2024-00461572, RS-2024-00459026, and RS-2021-0-1817), funded by the Korean government (MSIT).

References

[1] db_bench. https://github.com/facebook/rocksdb/wiki/Benchmarking-tools.

[2] Muhammad Yousuf Ahmad and Bettina Kemme. Compaction Management in Distributed Key-Value Datastores. *Proceedings of the VLDB Endowment (VLDB)*, 8(8):850–861, 2015.

[3] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *USENIX Annual Technical Conference (USENIX ATC)*, pages 753–766, 2019.

[4] Oana Balmau, Rachid Guerraoui, Vasileios Trigonakis, and Igor Zablotchi. FloDB: Unlocking Memory in Persistent Key-Value Stores. In *12th European Conference on Computer Systems (EuroSys)*, pages 80–94, 2017.

[5] Laurent Bindschaedler, Ashvin Goel, and Willy Zwaenepoel. Hailstorm: Disaggregated Compute and Storage for Distributed LSM-based Databases. In *25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 301–316, 2020.

[6] Leo Breiman. Bagging Predictors. *Machine Learning*, 24(2):123–140, 1996.

[7] Minguk Choi, Seehwan Yoo, and Jongmoo Choi. Can Learned Indexes be Built Efficiently? A Deep Dive into Sampling Trade-offs. *Proceedings of the ACM on Management of Data*, 2(3):1–25, 2024.

[8] Gerda Claeskens and Nils Lid Hjort. *Model Selection and Model Averaging*. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press, 2008.

[9] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing (SoCC)*, pages 143–154, 2010.

[10] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. From WiscKey to Bourbon: A Learned Index for Log-Structured Merge Trees. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 155–171, 2020.

[11] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David Lomet, and Tim Kraska. ALEX: An Updatable Adaptive Learned Index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, page 969–984, 2020.

[12] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Evolution of Development Priorities in Key-value Stores Serving Large-scale Applications: The RocksDB Experience. In *19th USENIX Conference on File and Storage Technologies (FAST)*, pages 33–49, 2021.

[13] Paolo Ferragina and Giorgio Vinciguerra. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proceedings of VLDB Endowment (VLDB)*, 13(8):1162–1175, 2020.

[14] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. FITing-Tree: A Data-aware Index Structure. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD)*, pages 1189–1206, 2019.

[15] Nils Lid Hjort and Gerda Claeskens. Frequentist Model Average Estimators. *Journal of the American Statistical Association*, 98(464):879–899, 2003.

[16] Nils Lid Hjort and Gerda Claeskens. Focused Information Criteria and Model Averaging for the Cox Hazard Regression Model. *Journal of the American Statistical Association*, 101(476):1449–1464, 2006.

[17] HyperDex. HyperLevelDB. http://hyperdex.org/performance/leveldb/, 2017.

[18] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young ri Choi. SLM-DB: Single-Level Key-Value Store with Persistent Memory. In *17th USENIX Conference on File and Storage Technologies (FAST)*, page 191–204, 2019.

[19] Dongui Kim, Chanyeol Park, Sang-Won Lee, and Beomseok Nam. BoLT: Barrier-optimized LSM-Tree. In *21st International Middleware Conference (Middleware)*, page 119–133, 2020.

[20] Wonbae Kim, Chanyeol Park, Dongui Kim, Hyeongjun Park, Young ri Choi, Alan Sussman, and Beomseok Nam. ListDB: Union of Write-Ahead Logs and Persistent SkipLists for Incremental Checkpointing on Persistent Memory. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 161–177, 2022.

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

[22] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. RadixSpline: A Single-Pass Learned Index. In *Proceedings of the 3rd International Workshop on Exploiting Artificial Intelligence Techniques for Data Management (aiDM)*, pages 1–5, 2020.

[23] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD)*, pages 489–504, 2018.

[24] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. KVell: The Design and Implementation of a Fast Persistent Key-Value Store. In *the 27th ACM Symposium on Operating Systems Principles (SOSP)*, page 447–461, 2019.

[25] Pengfei Li, Yu Hua, Pengfei Zuo, and Jingnan Jia. A Scalable Learned Index Scheme in Storage Systems. *arXiv preprint arXiv:1905.06256*, 2019.

[26] Kai Lu, Nannan Zhao, Jiguang Wan, Changhong Fei, Wei Zhao, and Tongliang Deng. TridentKV: A Read-Optimized LSM-Tree Based KV Store via Adaptive Indexing and Space-Efficient Partitioning. *IEEE Transactions on Parallel and Distributed Systems*, 33(8):1953–1966, 2021.

[27] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Wisckey: Separating Keys from Values in SSD-Conscious Storage. *ACM Transactions On Storage (TOS)*, 13(1):1–28, 2017.

[28] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. Benchmarking Learned Indexes. *Proceedings of the VLDB Endowment (VLDB)*, 14(1):1–13, 2020.

[29] Meta. RocksDB. https://rocksdb.org, 2023.

[30] Adrian E. Raftery, David Madigan, and Jennifer A. Hoeting. Bayesian Model Averaging for Linear Regression Models. *Journal of the American Statistical Association*, 92(437):179–191, 1997.

[31] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. PebblesDB: Building Key-Value Stores Using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, pages 497–514, 2017.

[32] Liana V. Rodriguez, Farzana Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan. Learning Cache Replacement with CACHEUS. In *19th USENIX Conference on File and Storage Technologies (FAST)*, pages 341–354, 2021.

[33] Sanjay Ghemawhat, Jeff Dean, Chris Mumford, David Grogan, and Victor Costan. LevelDB. https://github.com/google/leveldb, 2011.

[34] Jinghan Sun, Shaobo Li, Yunxin Sun, Chao Sun, Dejan Vucinic, and Jian Huang. LeaFTL: A Learning-Based Flash Translation Layer for Solid-State Drives. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 442–456, 2023.

[35] Chuzhe Tang, Youyun Wang, Zhiyuan Dong, Gansen Hu, Zhaoguo Wang, Minjie Wang, and Haibo Chen. XIndex: a scalable learned index for multicore data storage. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 308–320, 2020.

[36] Shengzhe Wang, Zihang Lin, Suzhen Wu, Hong Jiang, Jie Zhang, and Bo Mao. LearnedFTL: A Learning-Based Page-Level FTL for Reducing Double Reads in Flash-Based SSDs . In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 616–629, 2024.

[37] Yi Wang, Jianan Yuan, Shangyu Wu, Huan Liu, Jiaxian Chen, Chenlin Ma, and Jianbin Qin. LeaderKV: Improving Read Performance of KV Stores via Learned Index and Decoupled KV Table. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, pages 29–41, 2024.

[38] Jiacheng Wu, Yong Zhang, Shimin Chen, Jin Wang, Yu Chen, and Chunxiao Xing. Updatable Learned Index with Precise Positions. *Proceedings of VLDB Endowment (VLDB)*, 14(8):1276–1288, 2021.

[39] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with Matrix Container in NVM. In *USENIX Annual Technical Conference (USENIX ATC)*, pages 17–31, 2020.

[40] Jinghuan Yu, Sam H. Noh, Young ri Choi, and Chun Jason Xue. ADOC: Automatically Harmonizing Dataflow Between Components in Log-Structured Key-Value Stores for Improved Performance. In *21st USENIX Conference on File and Storage Technologies (FAST)*, pages 65–80, 2023.

[41] Zheng Yuan and Yuhong Yang. Combining Linear Regression Models: When and How? *Journal of the American Statistical Association*, 100(472):1202–1214, 2005.

Received July 2025; revised October 2025; accepted November 2025