
# Kirin: Efficient In-Storage Learned Compaction for LSM-Trees via System-Algorithm Co-Design

Guifeng Wang[1], Shengan Zheng[2*], Penghao Sun[1], Jin Pu[1], Kaijiang Deng[1], Bowen Zhang[1], Weihan Kong[1], Cong Zhou[1], Yifan Hua[3] and Linpeng Huang[1*]
[1]School of Computer Science, Shanghai Jiao Tong University
[2]MoE Key Lab of Artificial Intelligence, AI Institute, Shanghai Jiao Tong University [3]Peking University
[1]{wangguifeng, sunpenghao, grey-hibari, 2020-sjtu-dkj, bowenzhang, weihankong, cong258258, lphuang}@sjtu.edu.cn
[2]shengan@sjtu.edu.cn [3]huayifan@pku.edu.cn

## ABSTRACT

The log-structured merge-trees (LSM-trees) are widely used in modern Key-Value (KV) stores, offering strong write performance but facing significant inefficiencies in compaction and indexing. While recent researches have integrated learned indexes with LSM-trees to address these inefficiencies, their integration remains hindered by excessive cold data movement, limited parallelism in model training, and the decoupled nature of compaction and training. In this paper, we present Kirin, a hybrid KV store that synergistically integrates LSM-tree and learned index, and leverages computational storage devices (CSDs) to offload data-intensive tasks. Kirin introduces a novel learned compaction approach that embeds model training directly into the compaction process to conceal training latency and enable timely model updates. Kirin also employs a collaborative approach between the host and CSD to parallelize compaction and minimize storage access during indexing. Our experiments with DaisyPlus OpenSSD demonstrate that Kirin outperforms existing solutions in both read and write throughput by a large margin, while maintaining low read latency under heavy write workloads.

**PVLDB Reference Format:**
Guifeng Wang, Shengan Zheng, Penghao Sun, Jin Pu, Kaijiang Deng, Bowen Zhang, Weihan Kong, Cong Zhou, Yifan Hua, and Linpeng Huang. Kirin: Efficient In-Storage Learned Compaction for LSM-Trees via System-Algorithm Co-Design. PVLDB, 19(9): 1991 - 2004, 2026.
doi:10.14778/3819518.3819529

**PVLDB Artifact Availability:**
The source code, data, and/or other artifacts have been made available at https://github.com/SJTU-DDST/kirin-code.

## 1 INTRODUCTION

Key-value (KV) stores serve as foundational components in numerous domains, including internet services [11, 50], caching systems [14, 48], distributed stores [43, 79], and real-time analytics [4, 22]. Among these, log-structured merge-tree (LSM-tree) [52] stands out as the storage backbone for mainstream KV stores such as LevelDB [23], RocksDB [17], X-Engine [24], and Cassandra [36].

*Shengan Zheng and Linpeng Huang are corresponding authors.

Proceedings of the VLDB Endowment, Vol. 19, No. 9 ISSN 2150-8097.
doi:10.14778/3819518.3819529

A major challenge in these systems is the performance of random writes, particularly when faced with bursts or sustained random write traffic, common in online transaction processing (OLTP) workloads [45, 56]. LSM-trees address this by buffering KV pairs in memory and sequentially flushing them to storage, making them highly effective for write-intensive workloads and outperforming traditional tree-based indexes (e.g., B+-trees).

Despite their effectiveness, LSM-trees encounter performance bottlenecks, primarily due to inefficiencies in indexing and compaction. Key lookup operations in on-disk tables are inherently inefficient due to the absence of fine-grained mappings between keys and their respective storage locations. Consequently, retrieving a specific KV pair requires reading multiple storage blocks, leading to high read amplification [9, 69, 78]. Additionally, the compaction process, which moves KV pairs between storage levels to improve space utilization, adds further strain on system resources. Compaction demands substantial computational effort and storage bandwidth, as it involves reading, sorting, merging, and rewriting large quantities of KV pairs [44, 66, 77].

Machine learning-driven learned indexes have emerged as a promising solution for improving the indexing performance of LSM-trees [21, 34]. Unlike traditional tree-based indexing methods, learned indexes predict the position of key-value (KV) pairs by employing linear regression models that fit the distribution of sorted keys [13, 20, 62, 68, 75]. This simplifies the indexing process, which is a key contributor to read latency, particularly on high-speed storage devices such as SSDs [9]. Recent researches [1, 9, 46, 69] have explored integrating learned indexes into LSM-trees, aiming to reduce lookup latency and minimize the index's storage footprint. The integration typically involves training machine learning models on sorted KV pairs in SSTables and replacing conventional index blocks with space-efficient learned models. The synergy between LSM-trees and learned indexes offers an effective solution to improve read throughput without compromising the LSM-tree's inherent high write performance.

Nevertheless, existing hybrid indexes that integrate the LSM-tree and learned index are hindered by the inefficiencies in both compaction and training processes. Compaction plays a crucial role in LSM-trees by removing invalid KV pairs and maintaining storage efficiency [25, 77]. However, it involves loading large volumes of cold data in the lower levels from storage to memory, consuming substantial I/O bandwidth and leading to significant write stalls [15, 74]. As illustrated in Figure 1(a), once bandwidth reaches saturation, additional compaction threads provide negligible improvement, highlighting the dominance of the host-storage I/O bottleneck.

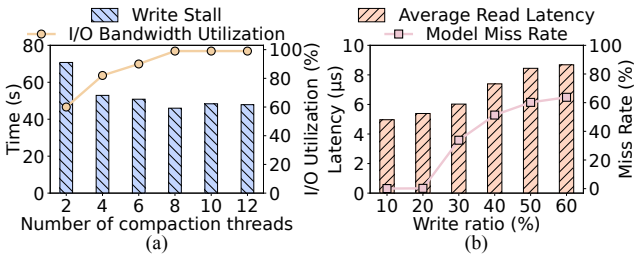


**Figure 1: (a) The impact of the number of compaction threads on write stall and I/O utilization in RocksDB. (b) Lookup latency and miss rate of the learned model under varying write percentages in Bourbon.**

Furthermore, the integration of LSM-tree and learned index is also undermined by the separate handling of compaction and training, resulting in additional data movement overhead. During compaction, key distribution in SSTables is changed, requiring the learned models to be retrained. Existing strategies [9, 46, 69] treat compaction and training as separate processes, extracting newly merged and sorted keys from the compaction flow to train the models in a separate thread. This separation introduces additional data movement overhead and prolongs the training process. Moreover, while models are being retrained, queries to newly compacted SSTables must bypass the outdated learned indexes and fall back to original search paths, which undermines the intended benefits of learned indexes. This results in suboptimal performance, particularly for write-intensive workloads where retraining is frequent. As shown in Figure 1(b), the miss rate of the learned model in Bourbon [9] increases significantly with the write ratio because the models cannot be retrained in time, resulting in a higher lookup latency.

Meanwhile, the training of learned indexes also faces severe scalability bottlenecks. The Piecewise Linear Regression (PLR) algorithm, commonly used in learned indexes [68, 72, 75], is notoriously difficult to parallelize on either multi-core CPUs or FPGAs [29] due to its iterative nature. Additionally, PLR models are unable to directly locate key-value pairs, forcing the host CPU to traverse all key-value pairs within a range, which leads to unnecessary storage accesses and exacerbates performance degradation [69]. These inefficiencies are further intensified by the lengthy communication path between the host and storage devices, leading to increased latency and power consumption.

Computational storage devices (CSDs) [35, 39, 73] provide an opportunity to overcome the above challenges. By integrating computational engines (e.g., embedded general-purpose processors, FPGA, etc.) near storage, CSDs offload the data-intensive tasks from the host and leverage the high internal bandwidth of the storage [27]. The near-data processing capability of CSDs enables operations like data compaction and searching to be performed directly within the CSD, reducing data movement overhead and improving bandwidth efficiency. Additionally, CSDs can accelerate model training by offloading tasks to the integrated FPGA, enabling parallel processing to boost efficiency while lightening the load on the host system.

In this paper, we present Kirin, a hybrid index that integrates LSM-tree with learned index, leveraging CSD to offload compaction, training, and searching tasks. Kirin features a novel approach called *learned compaction*, which integrates compaction and training into a unified process on the CSD. In contrast to conventional compaction, learned compaction directly uses keys generated during compaction for training, eliminating the need for separate key preparation. By exploiting the parallelism between compaction and training, Kirin minimizes total processing time by overlapping their execution. To further accelerate learned compaction and address write stalls, we categorize SSTables by data hotness: the host handles hotter SSTables in higher levels, while the CSD processes colder ones in lower levels.

On the CSD, we employ FPGA-based pipelining to accelerate learned compaction. Unlike prior FPGA-based approaches [64, 66, 77], we introduce a parallel PLR accelerator that speeds up model training by enabling parallel execution of PLR operations via system-algorithm co-design. To the best of our knowledge, this is the first FPGA-based accelerator of PLR-based learned index training. In addition, we mitigate the data movement overhead in indexing through a collaborative indexing framework, where the host computes position ranges for KV pairs, sends them to the CSD in batches, and the CSD performs the scan, returning results with minimal data transfer. Our main contributions are summarized as follows:

- We present Kirin, a hybrid index that synergistically combines LSM-tree and learned index, and leverages CSD to enhance hybrid indexing performance for both read and write operations.
- We propose a novel learned compaction approach that tightly integrates training into the compaction process, concealing training time while enabling rapid updates of outdated models.
- We introduce a collaborative indexing framework that leverages both the host and CSD, minimizing cold data movement and alleviating write stalls in write-heavy workloads.
- We build a learned compaction accelerator on the FPGA of the CSD, incorporating a parallel PLR accelerator to accelerate offloaded compaction tasks, resolving performance bottlenecks in both LSM-tree and learned index.
- We implement Kirin on a real FPGA-based CSD platform. Evaluation shows that Kirin substantially reduces the overhead of compaction and retraining, achieving 2.33× and 7.74× higher throughput compared to Bourbon and RocksDB, respectively.

## 2 BACKGROUND

### 2.1 LSM-tree Based KV Store

The Log-Structured Merge (LSM) tree is a data structure optimized for write-intensive workloads [52], commonly used in modern KV stores such as LevelDB [23] and RocksDB [17]. The LSM-tree improves write performance by buffering writes in memory and merging them into storage in batches. However, it constitutes a tradeoff between read and write operations, resulting in compromised read performance. It typically includes an in-memory MemTable, where incoming writes are initially stored. Once the MemTable reaches a predefined size, it becomes an Immutable MemTable and is flushed to Level 0 on disk as a Sorted String Table (SSTable). The SSTables are organized into multiple levels, with the capacity of each level growing exponentially.

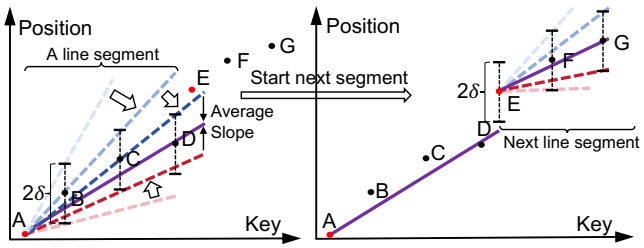


**Figure 2: Illustration of PLR Algorithm.**

Write operations in the LSM-tree involve appending data to the MemTable, which keeps all buffered KV pairs sorted by keys in a search-optimized structure such as a skiplist. A process called compaction is employed to maintain the structure's efficiency and prevent an unbounded growth of SSTables. During compaction, SSTables from a higher level are merged into those of lower levels, removing invalid entries and consolidating updates. This process reduces storage overhead and improves read performance by minimizing the number of SSTables that need to be searched. Nevertheless, compaction consumes significant computational resources and is generally bottlenecked by storage bandwidth.

For read operations, the LSM-tree first checks MemTable for the requested key. If the KV pair does not exist, it is searched in the on-storage tree from the highest to the lowest level. This multi-level lookup strategy can significantly increase latency if the target key is located at lower levels, as it requires checking multiple SSTables at these levels. Moreover, searching within SSTables involves binary lookups through index and data blocks, resulting in considerable disk access latency.

## 2.2 Learned Index and PLR Algorithm

Learned index employs machine learning techniques to improve read performance in traditional index structures, while also reducing memory usage [21, 34]. It functions as a regression model that approximates the relationship between sorted keys and their positions by estimating the cumulative distribution function (CDF) [13, 43, 65]. Given a key, the learned index predicts its position within a specified error range using the trained regression model. Several studies have integrated learned index with LSM trees [1, 9, 46, 69], significantly reducing indexing overhead and enhancing read throughput. However, these approaches fall short in balancing read and write performance, as they often overlook the increased training overhead of learned models and inefficiencies during data compaction. In addition, the read performance is also restricted by storage access overhead caused by inaccurate predictions from the learned index. Kirin tackles these challenges by co-locating compaction and model training on computational storage devices, eliminating redundant data movement and enabling low-latency, fine-grained updates to learned models through collaborative offloading.

Since the introduction of the linear segmentation algorithm in FITing-Tree [21], the Piecewise Linear Regression (PLR) algorithm has become the core model in many learned index applications [9, 43, 46, 62, 65, 68]. The key concept behind the PLR algorithm is to approximate data points within a specified error bound $\delta$ using multiple line segments, each representing a linear regression model. During training, for each new key, the PLR algorithm evaluates whether the key falls within the current segment's bounding region defined by upper and lower lines (bounds). If the key falls within the bounds, the algorithm computes a new pair of upper/lower boundary lines using the bounding points (the key's position plus/minus $\delta$). It then updates the segment's bounds by taking the minimum of the upper slopes and the maximum of the lower slopes between the existing and newly calculated bounds. These adjustments are incrementally updated as consecutive sequence points are processed [72]. Once a key falls outside the range (cannot be approximated within $[-\delta, \delta]$), the segment is finalized, and its slope is calculated as the average of the upper and lower slopes. We select the PLR algorithm for Kirin due to its stream-based iterative nature, which maps efficiently to FPGA pipelining, unlike multi-pass approaches like RadixSpline [32] and PGM-index [20] that require complex auxiliary structure construction.

Figure 2 illustrates the main procedure of PLR algorithm. It processes a sequence of points, each comprising a key and a position, iteratively calculates boundary slopes, and finalizes a line segment when a point falls outside the current bounds. Point $A$ is the beginning point of a line segment, referred to as the *support point* [72]. Point $B$ establishes the initial upper and lower boundary lines with point $A$ for the segment, represented by light blue and red lines. Point $C$ lies within the region enclosed by these boundary lines, and both the upper and lower boundary lines are updated. Point $D$ also resides within the current region, but its lower bounding point falls below the red line, so only the upper boundary is updated (now shown as the dark blue line). Point $E$ is above the dark blue line therefore the current line segment terminates. The slope of this segment is the average of the red and lower blue lines, and a new segment (Points $F$, $G$) begins with point $E$ as its support point.

Training learned models over large key sets is computationally expensive, making FPGA acceleration an attractive solution [29]. However, the inherent data dependencies in the PLR algorithm make FPGA parallelization challenging. In the PLR algorithm outlined in Figure 2, boundary lines are incrementally updated, and the support point is reset at the end of each segment. This process involves evaluating the positional relationship between the input point and the current boundary lines. It requires up-to-date slope values for the boundary lines and support point, introducing two types of data dependencies:

- **Slope dependency**. Before updating slopes in each iteration, the latest slopes from the previous iteration must be available. This introduces an inter-iteration dependency, where each iteration must wait for the preceding iteration's slopes to be determined. The slope dependency enforces a strict execution order, severely limiting the algorithm's scalability.
- **Point dependency**. If the current point falls outside the boundaries, it becomes the support point for the next segment. However, it is impossible to predict whether the current point will serve this role in advance. Thus, each iteration depends on the outcome of the previous iteration to determine the support point.

In summary, PLR's inter-iteration data dependencies enforce strict sequential execution, severely limiting scalability and preventing efficient utilization of hardware parallelism.

## 2.3 Computational Storage Device

Computational Storage Device (CSD) integrates processing capabilities directly within storage devices, allowing computation to occur where the data resides. CSD typically employs fast storage devices like SSDs and computational engines such as ARM cores and FPGA, leveraging the high internal bandwidth of SSD to reduce data movement and accelerate computation [63]. FPGAs, in particular, are well-suited for CSDs because of their ability to build reconfigurable pipelines and support fine-grained parallelism [77]. In the industry, several CSD products have been released, such as OpenSSD [8], SmartSSD [49], and ScaleFlux [59], and these products continue to evolve. Furthermore, the SNIA Computational Storage Technical Work Group (TWG) and NVM express have created a set of standards and defined the interfaces to promote the interoperability of CSDs [51, 60].

The CSD also attracts the attention of academia and is widely used to optimize data-intensive applications such as graph processing [35, 39, 47], neural network training [27, 30, 40], data analysis in HTAP [38] and key-value store [5, 25]. These researches leverage CSDs to minimize data movement between storage and host memory, alleviating PCIe bus congestion and freeing up host CPU for other application tasks. The computational engines on the CSDs also provide an opportunity to parallelize and accelerate the computation in the above applications. Despite these specific applications, some studies extensively research in-storage file systems [28, 33, 76] and generic frameworks for CSD [57, 73], paving the way for broader applicability.

# 3 DESIGN

This section introduces the design of Kirin, a hybrid key-value store that combines the efficiency of learned indexes with the structural advantages of the LSM-tree. We begin by outlining the hybrid architecture of Kirin and its data placement strategy, followed by the collaborative indexing and search workflow that minimizes data movement through workload partitioning. Then, we delve into the learned compaction engine, which features a parallelized PLR accelerator implemented on FPGA to accelerate in-storage model training and improve compaction throughput.

The design of Kirin is engineered to meet three primary design goals, enabling the efficient integration of learned indexes with LSM-tree on computational storage.

- **Concealing model training latency**. The core objective is to completely hide the latency of model retraining so that it adds no overhead to the LSM-tree's write path. This is achieved by tightly integrating the training process with the I/O-intensive compaction workflow and accelerating the unified operation in hardware (Section 3.2).
- **Minimizing host-storage data movement**. We offload the data-intensive operations including range scanning and the learned compaction process to the CSD, thereby avoiding costly data transfers over the PCIe bus (Section 3.3).
- **Maximizing systemic parallelism and resource utilization**. To fully leverage the CSD's potential, we redesign the training algorithm for hardware parallelism (Section 3.4) and orchestrate the host CPU and CSD engine to maximize the utilization of transmission bandwidth (Section 3.2).

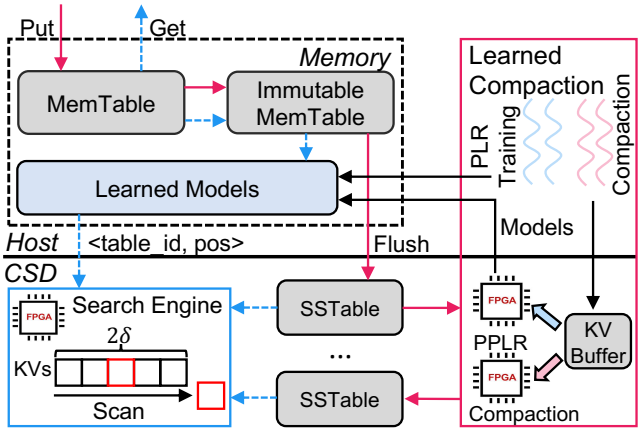


**Figure 3: The system overview of Kirin.**

## 3.1 System Overview

The overall architecture of Kirin, illustrated in Figure 3, combines the write-optimized structure of the LSM-tree with the read efficiency of the learned index. In the hybrid index structure, the memtable and learned index models reside in host memory, while flushed SSTables are stored on the Computational Storage Device (CSD). A learned model is trained for each SSTable via Piecewise Linear Regression (PLR) algorithm to approximate key positions.

To retrieve KV pairs, Kirin performs a memtable search first in host memory and, for absent keys, leverages the learned models to determine the key's position range within the corresponding SSTables. These ranges are then batched and sent to the CSD, where an FPGA-based search engine scans these ranges to retrieve the target KV pairs with minimal storage access overhead. As new KVs are inserted, higher-level SSTables are merged into lower levels during compaction, necessitating the retraining of learned models to keep indexing accurate for subsequent queries.

To mitigate the performance impact of model retraining, we propose *Learned Compaction*, a tightly integrated approach that embeds model training directly into the compaction pipeline. Learned compaction utilizes the inherent merge-sort operations in compaction to generate the sorted key sequences required for model training without altering the original compaction workflow. Since SSTable compaction and model training operate independently, they can run in parallel, effectively overlapping training time with the compaction process. The updated models, generated alongside the compacted SSTables, become immediately available for subsequent queries, ensuring that new models are available for queries to the SSTable.

To accelerate learned compaction across all levels, Kirin adopts a collaborative learned compaction framework (Section 3.2) that minimizes data movement by distributing tasks between the host and CSD. The host handles hot data in Levels 0 and 1, where frequent access and residency in the page cache facilitate efficient processing. Meanwhile, the CSD offloads and parallelizes compaction tasks for colder data in lower levels, capitalizing on its high internal bandwidth and minimizing data movement between storage and the host.

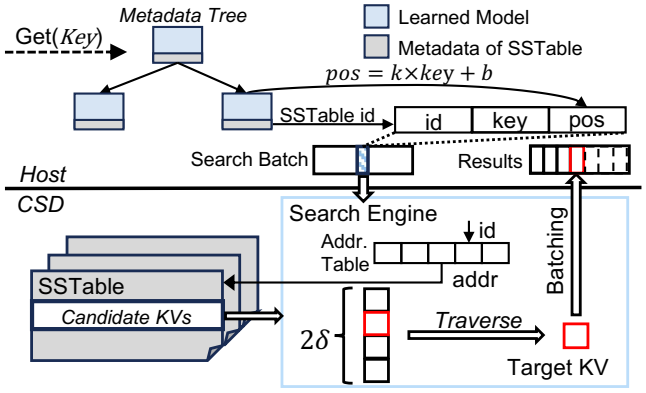


**Figure 4: The coordinated search scheme of Kirin.**

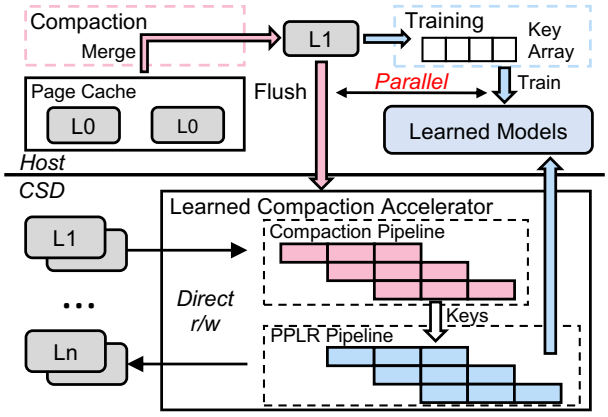


**Figure 5: The collaborative learned compaction framework of Kirin.**

Within the CSD, Kirin accelerates learned compaction by employing a dedicated offload engine on the FPGA to accelerate the offloaded learned compaction tasks (Section 3.3). The engine harnesses the FPGA's parallel processing capabilities to reduce data movement and accelerate the compaction process. To further expedite the training of learned models, we implement a Parallel PLR (PPLR) accelerator (Section 3.4) on the FPGA, which batch-trains keys in a pipelined manner, maximizing computational efficiency and concealing training latency. Upon completion, only the lightweight learned models are transferred to host memory, while SSTables remain on the CSD, significantly reducing data movement and improving overall performance.

## 3.2 Collaborative Indexing

Kirin employs a hybrid index structure that combines the LSM-tree and learned index to enhance index performance by reducing SSTable lookup latency. However, model training and retraining during compaction introduce non-negligible computational overhead, and a simple host-side parallelization is insufficient because it fails to mitigate the I/O bottleneck inherent to compaction. Moreover, the approximate nature of learned predictions also necessitates range scans within SSTables to locate exact key-value pairs, leading to additional storage accesses and degraded query efficiency. A naive offload strategy cannot maximize the system's parallelism and available bandwidth resources, as it risks creating a new bottleneck on the CSD while leaving powerful host resources idle.

To address these challenges, Kirin employs a collaborative mechanism using both the host and CSD. During searches, Kirin determines approximate position ranges within SSTables using the host-side hybrid index. These ranges are delegated to the CSD in batches, enabling parallel host range computation for subsequent queries. Kirin incorporates model training into compaction, forming a learned compaction workflow. This division optimizes resource utilization and minimizes training and compaction overhead.

**Coordinated KV Searching.** In Kirin, each SSTable is associated with a learned index model to expedite KV pair lookups. These models are trained concurrently as SSTables are flushed to the CSD. When their corresponding SSTables are later merged during compaction, the old models are deleted, and new models are generated alongside the lower-level SSTables via learned compaction.

The procedure of KV search is illustrated in Figure 4, Kirin begins by locating the relevant SSTable and its corresponding model through the upper metadata tree. The learned model, structured as a series of linear regression segments, undergoes a binary search to locate the segment encompassing the key's range. The key's approximate position within the SSTable is then estimated via linear regression model with an associated error bound $[pos - \delta, pos + \delta]$. Kirin finally retrieves the KV pairs within this range from storage and scans them to locate the target key. If performed on the host, such scans incur significant data movement over PCIe, potentially saturating bandwidth under heavy workloads.

To mitigate data movement and performance bottlenecks in learned index lookups, Kirin introduces a coordinated search scheme that integrates an FPGA-based search engine within the CSD. Kirin processes key-value (KV) position ranges calculated via learned index models, combines these ranges with the corresponding SSTable IDs and target keys, and transmits them to the CSD in batches. The FPGA-based search engine within the CSD first locates the SSTable's physical address by looking up the address list using the SSTable ID. Then, the search engine on CSD retrieves KV pairs from storage via its internal bus, and filters candidates within the estimated range to identify the target key. Successful lookups complete locally on the CSD, and the host issues a new query into the batch to keep the pipeline saturated. Otherwise, the key is re-queued for further search in lower levels. The coordinated search scheme minimizes host-CSD communication and leverages CSD's internal bandwidth to improve search efficiency.

**Collaborative Learned Compaction.** A key objective in Kirin is to reduce unnecessary host–device data movement during compaction. To this end, we design a collaborative framework (Figure 5) that partitions compaction tasks by data level: Levels 0 to 1, which deal with frequently accessed data, are handled by the host CPU, ensuring low-latency processing. Meanwhile, the FPGA on the CSD manages Levels 1 to n, processing large volumes of cold data and amortizing the PCIe communication overhead through efficient, in-storage computations. This division minimizes data movement overhead, leveraging the computational strengths of both the host and CSD to maximize the efficiency of learned compaction.

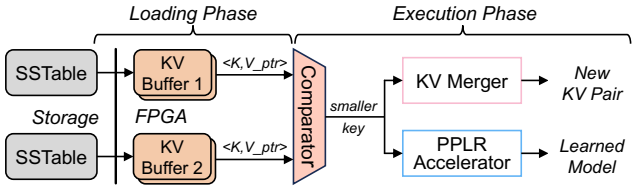


**Figure 6: The architecture of the learned compaction accelerator.**

For learned compaction on the host side, Kirin parallelizes the training and flushing stages. When Kirin converts an immutable memtable to an SSTable or merges SSTables in compaction, the valid keys are collected, sorted, and cached in host memory during table traversal. The new SSTable is flushed to the CSD, triggering a host thread to concurrently train the corresponding learned model using the sorted valid keys. Since model training typically completes faster than data flushing (details in Section 4.4 and Figure 14), the two operations overlap, introducing negligible additional latency for frequently updated SSTables in Levels 0 and 1. Upon completion, Kirin atomically switches visibility to the new SSTables and their corresponding models simultaneously. This strict lifecycle coupling eliminates model staleness due to the immutability of SSTables, ensuring that all lookups are consistently served by the learned index path without falling back to the original search path.

On the CSD, we build a learned compaction accelerator that integrates the learned model training into the compaction pipeline. As illustrated in Figure 5, the learned compaction accelerator directly retrieves high-level SSTables from storage, merges them into low-level SSTables and writes the results back to storage. Simultaneously, the PPLR accelerator concurrently trains learned models using keys output by the compaction pipeline. Once trained, the new models are sent to the host, minimizing the overhead of additional data transfers. We detail the design of the learned compaction accelerator in Section 3.3. The collaborative approach leverages the internal bandwidth of CSD and the data parallelism of FPGA to mask training latency, eliminate unnecessary data transfers, and avoid interference with host memory, thereby significantly improving the efficiency of learned compaction.

## 3.3 Learned Compaction Accelerator

Kirin introduces a learned compaction accelerator on the CSD's FPGA to speed up learned compaction tasks for levels below Level 1, which are offloaded from the host. However, a naive hardware implementation that separately executes compaction and retraining on the CSD would be critically inefficient. Such a two-stage process faces prohibitive bottlenecks: it demands an infeasible amount of on-chip RAM for intermediate key buffering and introduces a long latency chain, as training must wait for compaction to complete. Therefore, Kirin features a pipelined architecture that integrates model training directly into the compaction workflow, leveraging the FPGA's parallelism and the CSD's internal bandwidth to minimize data movement and enhance in-storage processing efficiency.

As shown in Figure 6, the learned compaction accelerator follows a two-phase workflow: a *loading phase* and an *execution phase*. In the loading phase, chunks of KV records are fetched from two adjacent levels of SSTables and stored in dedicated buffers. A key from each buffer is loaded into the comparator for merge-sorted processing. To enhance data locality and reduce overhead, keys are decoupled from values within the pipeline, avoiding unnecessary value duplication. This streaming approach enables continuous data flow that sustains both the compaction and model training pipelines on the FPGA.

The execution phase employs a pipelined approach to maximize computational resource utilization on the CSD. The execution pipeline begins by comparing two KV records, selecting the smaller key, and discarding duplicates. Then, the accelerator concurrently merges KV pairs and trains learned models. Afterwards, a KV entry is forwarded to the candidate array for the next iteration. Finally, both the merged KV pairs and their associated learned models are buffered and flushed back to storage.

Different from existing compaction pipelines [64, 66] which exclusively focus on compaction optimizations, we embed learned index training into the compaction process, forming a learned compaction pipeline that fully exploits pipeline parallelism. By leveraging a PPLR accelerator on the CSD's FPGA (detailed in Section 3.4), we enable the independent training of learned models using streamed keys. During compaction, the pipeline selects the smaller key, which is sent to the PPLR accelerator for slope calculations and incremental updates of line segment boundaries. Since the smaller key is determined in the initial steps of the pipeline, model training can run in parallel with subsequent compaction processes. As compaction generally takes longer than training, the training overhead is effectively concealed. Compared to host-side training during flush operations, the learned compaction pipeline on the CSD significantly shortens the data path between training and compaction, enhancing the effectiveness of learned compaction.

To ensure uninterrupted pipeline operations, we introduce a dual-buffer mechanism that guarantees a steady flow of KV pairs. The mechanism utilizes KV buffers to store data fetched from storage and output buffers to hold processed data ready for writing. Each KV buffer has a capacity twice that of the output buffer, divided into two equal-sized partitions. As soon as one partition is exhausted, the pipeline transitions immediately to the other, while the empty partition is concurrently refilled with new data from storage. This mechanism ensures that the KV buffer consistently supplies more KV pairs than the output buffer can accommodate during each stage, keeping the pipeline fully utilized. Consequently, the output buffer is consistently filled to capacity and flushed back to storage, optimizing the overall throughput.

## 3.4 Parallel PLR Accelerator

To accelerate the training of the learned index and enhance its performance under write-heavy workloads, we parallelize the PLR algorithm through a system-algorithm co-design approach. As outlined in Section 2.2, the original sequential PLR algorithm imposes strict inter-iteration dependencies due to its incremental slope and boundary updates, which severely limits its scalability. To address these challenges, we propose a parallel PLR (PPLR) algorithm that restructures the training process into parallelizable batches. Slope computations within a batch proceed independently, enabling efficient parallel execution while maintaining algorithm correctness.

**Algorithm 1:** Parallel PLR Algorithm

**Input:** $keys$: the input keys; $\delta$: the specified error bound
**Output:** $\delta$-bounded line segments

1 $segment\_finalized = false$; % **Initialization**
2 $sp$ = current segment's first point; % Fixed support point
3 % **The i-th iteration of the algorithm**
4 % **Get input point and boundary points**
5 **if** *reprocessing_queue is empty* **then**
6 | $p = (keys.read(), pos_{i-1} + 1)$;
7 **else**
8 | $p = reprocessing\_queue.dequeue()$;
9 $p_{upper} = (p.key, p.pos + \delta)$; $p_{lower} = (p.key, p.pos - \delta)$;
10 % **Compute slopes in parallel across iterations**
11 $cur\_sl = slope(sp, p)$;
12 $cur\_sl_{upper} = slope(sp, p_{upper})$;
13 $cur\_sl_{lower} = slope(sp, p_{lower})$;
14 % **Update slopes sequentially**
15 **if** $segment\_finalized$ **then**
16 | $reprocessing\_queue.enqueue(p)$;
17 **else if** $cur\_sl < sl_{upper}$ *and* $cur\_sl > sl_{lower}$ **then**
18 | $sl_{upper} = min(sl_{upper}, cur\_sl_{upper})$;
19 | $sl_{lower} = max(sl_{lower}, cur\_sl_{lower})$;
20 **else**
21 | $segment\_finalized = true$;
22 | % **Output line segment**
23 | $segments.add(line((sl_{upper} + sl_{lower})/2, sp))$;
24 | sp = p; % **Update support point**

As detailed in Algorithm 1, for each iteration in a batch, we anchor the support point (the first point in a segment) as a constant within each batch (Line 3). PPLR first gets the input key from the input stream or reprocessing queue (Line 6-13) and determines its upper and lower error-bounded points (Line 14). For these points, PPLR then computes the marginal line slopes relative to the current segment's support point (Line 16-18). It then evaluates whether the input point lies beyond the defined segment boundary (Line 23). If within the boundary, the upper and lower slopes are updated sequentially following the traditional PLR algorithm (Line 24-25). Otherwise, the line segment is finalized, and the input point becomes the support point for the next line segment (Line 28-33).

To maintain correctness across batch transitions, PPLR utilizes a reprocessing queue. When a key finalizes a segment, all subsequent keys arriving within that same batch are temporarily enqueued in this queue (Line 20-22). At the start of the next batch, the algorithm gives priority to points in the reprocessing queue before advancing to new input points (Line 12). This ensures these carry-over keys are correctly re-calculated against the updated support point.

We briefly prove the correctness of PPLR. Let $I_t$ denote the feasible slope interval after processing key $k_t$. The sequential algorithm recursively defines this state as $I_t = I_{t-1} \cap R(sp_t, k_t)$, where $sp_t$ is the support point at iteration $t$ and $R(sp_t, k_t)$ is the admissible slope range between support point $sp_t$ and current key $k_t$. PPLR computes $\{R(sp, k_t)\}$ for a batch in parallel and then updates $I_t$ sequentially. Since intersection is associative (e.g., $(A \cap B) \cap C = A \cap (B \cap C)$), the result is identical to sequential execution. If $I_t$ becomes empty, subsequent keys are invalidated and reprocessed in the next batch. This rollback mechanism resets the support point for the new segment, ensuring the output remains deterministic and identical to the sequential algorithm.

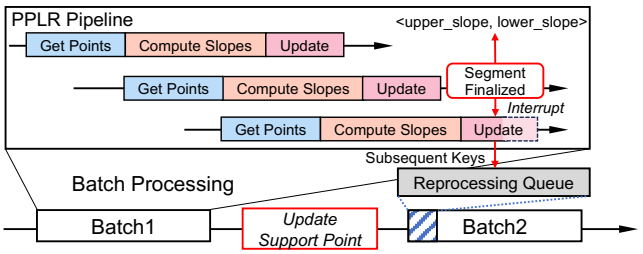


**Figure 7: The processing pipeline of the PPLR accelerator.**

The optimized PPLR procedure effectively mitigates the point and slope dependencies inherent in the original PLR algorithm. The constant support point ensures that slope calculations depend solely on the input points within the current batch and addresses the point dependency, enabling parallel slope computation of all iterations within the batch and minimizing training overhead. As for slope dependency, we restructure the algorithm steps by shifting slope computation to precede the update step. This facilitates parallel slope computation across iterations without compromising the algorithm's correctness. Ultimately, we significantly enhance the performance of PLR algorithm by fully parallelizing the slope computation which is the most time-consuming step.

To expedite learned index training using the PPLR algorithm, we design a PPLR accelerator on the CSD's FPGA, as illustrated in Figure 7. The PPLR accelerator employs a pipelined architecture that overlaps the execution phases of each iteration, thereby increasing the concurrency of the PPLR algorithm. The pipeline begins by fetching current point from reprocessing queue or input stream, then identifies error-bounded points, computes slopes, and updates segment boundaries. By integrating the PPLR algorithm with FPGA, Kirin can efficiently train models in CSD, while maintaining adaptability to other learned-index-based storage engines.

While batch training effectively resolves the inherent data dependencies in the PLR algorithm, its effectiveness hinges on the selection of an appropriate batch size. We propose an adaptive method to select batch sizes that align with the average distance between adjacent support points. For instance, an oversized batch triggers rollbacks when a segment terminates early. By predicting segment length via host sampling, we align batch boundaries to minimize these invalidated parallel computations. The method involves sampling distances between support points using the original PLR algorithm on the host and incorporating the calculated average as the batch size in the learned compaction request sent to the CSD. This value is updated after each learned compaction task, enabling the system to dynamically adapt to changes in data distribution. By dynamically adjusting batch size based on input key distribution, the PPLR accelerator seamlessly embeds learned index training within the compaction pipeline, significantly boosting both training speed and resource efficiency.

**Table 1: Resource utilization of Kirin on the FPGA.**

| Modules | LUT | FF | BRAM | DSP |
|---|---|---|---|---|
| Learned Compaction Accelerator | 95980 | 74085 | 198.5(3573KB) | 6 |
| Search Engine | 4391 | 6806 | 114(2052KB) | 0 |
| NVMe Controller | 17875 | 35950 | 80.5(1449KB) | 0 |
| Interconnect | 25351 | 30442 | 0 | 0 |
| Others | 55149 | 64782 | 5(90KB) | 0 |
| FPGA Total | 423403 | 846806 | 796(14328KB) | 1590 |
| **Utilization** | 46.94% | 25.04% | 50% | 0.38% |

**Table 2: Evaluation system configurations.**

| Platform | Item | Configuration |
|---|---|---|
| Host | CPU | 2× Intel Xeon Gold 6240 @2.6GHz |
| | Memory | 375GB DDR4 |
| | OS | Ubuntu 22.04 LTS (Linux 5.10.186) |
| Device | SoC | Xilinx Zynq Ultrascale+ ZU17EG |
| | Memory | 2GB LPDDR4 |
| | Host interface | NVMe over PCIe 3.0 ×8 |
| | Storage | 2× 32GB DDR4 DIMM |
| | Flash layout | 16KB page, 256 pages per block, 2 dies per channel, 8 channels |
| | Flash latency | 100$\mu$s read, 200$\mu$s write, 4ms erase |

# 4 EVALUATION

## 4.1 Hardware Implementation

We implement Kirin on DaisyPlus OpenSSD [7], an OpenSSD-based CSD platform widely adopted in prior computational-storage studies [54, 70, 73]. It integrates a Xilinx Zynq Ultrascale+ SoC with FPGA logic, quad-core ARM cores, 2GB DRAM, and exposes an NVMe interface over PCIe. Since the OpenSSD does not come with real NAND flash, we follow prior works [53, 73] and equip OpenSSD with two 32GB DDR4 DIMMs as the storage media. To more realistically reflect the performance of NAND flash-based SSDs, we implement a flash emulation module based on the FEMU emulator [42] to simulate the behavior of NAND flash, including read/write latencies and garbage collection. The ARM cores handle host NVMe requests and communicate with the FPGA, where the learned compaction accelerator and search engine are synthesized with Xilinx Vitis HLS 2023 at 300MHz. Table 1 reports FPGA resource utilization.

## 4.2 Experiment Setup

**Testbed.** Table 2 shows our detailed configurations of the testbed. We conduct our experiment on a server featuring two Intel Xeon Gold 6240 2.6 GHz 18-core CPUs with 375 GB memory, running Ubuntu 22.04 LTS with Linux kernel 5.10.186. The device is connected to the host via PCIe Gen3 ×8. The 2GB LPDDR4 memory acts as the device controller memory. We add flash latency to emulate the performance of real-world NAND flash, with the write and read bandwidths between the host and device set to 1.5GB/s and 1.8GB/s, respectively. The internal bandwidth between the storage media and the device controller memory is set to 4.4GB/s, which is comparable to real SSDs [15, 40, 73].

**Datasets.** We evaluate Kirin using a variety of datasets. We construct two synthetic datasets, uniform and normal, containing 100M KV pairs. Additionally, we utilize four real-world datasets: fb and wiki from [31], as well as libio and genome from [71]. Across these datasets, keys are 8B integers, and values are 64B strings. To demonstrate Kirin's applicability to diverse key types beyond 8B integers, we further evaluate it using uniform-16B (16B integers) and MemeTracker [41] (variable-length strings).

**Baselines.** We compare Kirin with three representative schemes:

- Bourbon [9]. An LSM-Tree based KV store that integrates a learned index to accelerate lookups.
- RocksDB [17]. A high-performance KV store optimized for fast storage.
- HotRAP [55]. An LSM-tree design specifically optimized for tiered storage, leveraging hot/cold data separation.
- STEM [66]. The state-of-the-art work that accelerates compaction using the FPGA.

Both Bourbon and RocksDB adopt host-only architectures without CSD offloading. In this case, the OpenSSD serves as a conventional storage device for them. For STEM, since it is not open-source, we reimplement it on the CSD's FPGA to the best of our ability and other components align with those of Kirin. Additionally, we incorporate an unoptimized PLR module which is implemented using native serial PLR algorithm into STEM to quantify the performance enhancements achieved attributed to hardware acceleration.

**Configurations.** We conduct evaluations using the latest versions of the comparison systems and refer to the official tuning manual for experimental verification [18]. For Kirin and all the baselines, we set the MemTable size and SSTable size to 64MB. For RocksDB, we use 6 compaction threads and 4 flush threads to fully utilize the bandwidth between the host and CSD. Compression and WAL functionalities are turned off. For Bourbon and STEM, we follow the default configurations in their papers.

## 4.3 End-to-End Performance

**Write-Only.** We begin by evaluating the write performance of Kirin under write-only workloads. Since Bourbon builds on LevelDB which supports only one write thread, we configure Kirin and all baselines with a single write thread. We load 100M KV pairs from eight datasets into the tested systems in random order, measuring throughput to represent write performance. As shown in Figure 8(a), Kirin outperforms all baselines across eight datasets. Kirin exhibits an average write speedup of 1.47× over RocksDB, 1.54× over Bourbon, 1.24× over HotRAP and 1.39× over STEM.

These improvements stem from the combined effects of the in-storage learned compaction accelerator and the collaborative framework. As illustrated in Figure 8(b) and (c), Kirin and STEM further reduce write stall time by offloading compaction tasks to the CSD's FPGA, significantly decreasing data movement between the host and device. Unlike STEM, which solely offloads compaction to FPGA, Kirin's collaborative framework assigns learned compaction tasks between Level 0 and Level 1 to the host side, leveraging the page cache to speed up the compaction process for levels 0 and 1. Moreover, since SSTables in level 1 and lower levels contain non-overlapping key ranges, Kirin achieves higher compaction efficiency through its pipelined design than STEM.

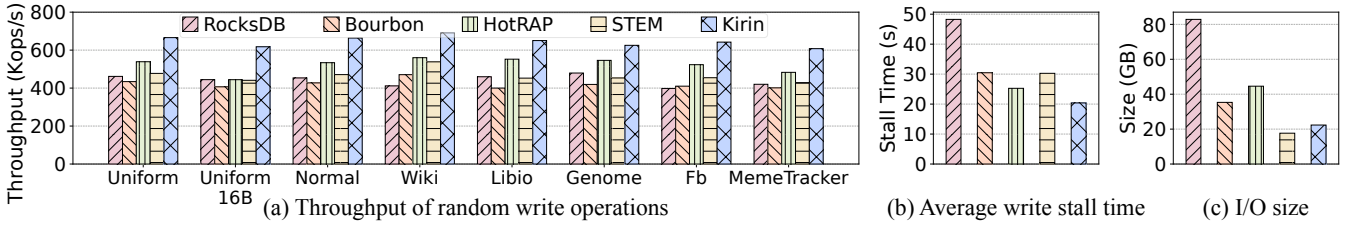


**Figure 8: Performance of Kirin and baselines under write-only workloads.**

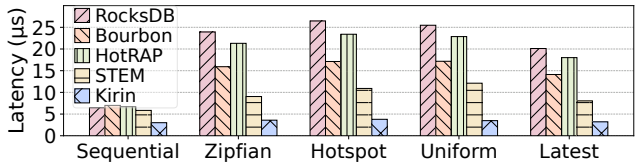


**Figure 9: Average read latency of Kirin and baselines under read-only workload.**

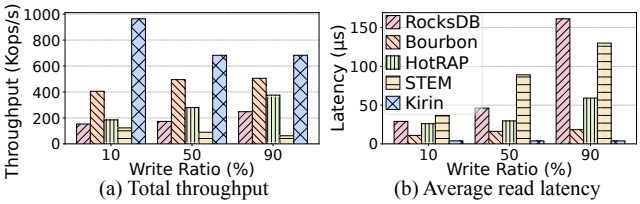


**Figure 10: Total throughput and read latency of Kirin and baselines under mixed workloads.**

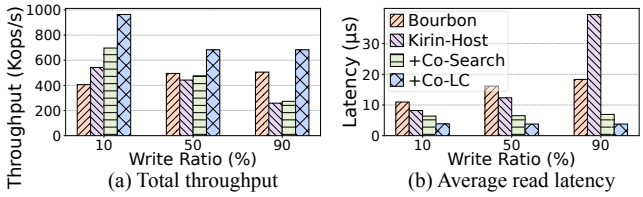


**Figure 11: The impact of each component of Kirin.**

**Read-Only.** We proceed to assess the read performance of Kirin under read-only workloads. To evaluate the impact of request distributions on read performance, we execute 20M lookup operations with four threads using the fb dataset across five distributions. STEM is not equipped with FPGA-based search engine, we compare it with Kirin to represent the efficiency of the search engine. As shown in Figure 9, Kirin achieves an average reduction in lookup latency of 5.87×, 4.12×, 5.25× and 2.66× compared to RocksDB, Bourbon, HotRAP and STEM, respectively.

This exceptional performance is driven by two key factors. First, Kirin builds learned models for all SSTables through learned compaction, including those in Level 0, where Bourbon does not apply learned models due to the short lifetime of SSTables. Our analysis indicates that by enabling learned index lookups in Levels 0 and 1, Kirin reduces the query latency in these layers by approximately 82%. This comprehensive use of learned models allows Kirin to effectively accelerate all lookups within SSTables. Second, the search engine on the CSD performs the scanning step, minimizing storage access overhead compared to the host. By overlapping search engine execution with position calculation on the host, Kirin further reduces lookup latency. As a result, Kirin outperforms Bourbon and STEM, achieving significant gains over RocksDB.

**Mixed Workload.** We next evaluate the read performance of Kirin in the presence of write operations. Write operations modify the key distribution within SSTables, requiring model retraining, which makes learned compaction efficiency crucial to Kirin's performance. To assess this, we execute 50M operations consisting of reads and writes after loading 100M KV pairs from the fb dataset. By varying the percentage of write operations, we measure and analyze the total throughput and average latency of lookups.

As shown in Figure 10, Kirin achieves the highest total throughput and the lowest read latency across all baselines. Specifically, Kirin delivers up to 42.61×, 4.84×, 15.61× and 34.34× lower latency for read operations compared to RocksDB, Bourbon, HotRAP and STEM, respectively. Moreover, as the write percentage increases from 10% to 90%, read latency rises by 456%, 67% and 260% for RocksDB, Bourbon and STEM, respectively, while Kirin consistently maintains significantly lower read latency. This is attributed to efficient model updates and batched searching, which reduce PCIe data transfers. We measure the average I/O size across varying write ratios, the results indicate that Kirin reduces the average data transfer size by approximately 61.8% compared to the baselines. Furthermore, algorithmic design plays a critical role in performance. Although STEM utilizes FPGA's parallel capabilities, it exhibits the lowest total throughput. This is because the data dependencies in its original PLR algorithm severely constrain performance. Our latency breakdown of the STEM accelerator reveals that the serial PLR computation consumes approximately 90% of the total processing time, creating a major pipeline bottleneck that throttles the faster merge-sort logic. This disparity highlights the importance of optimizing both hardware and algorithm design, which balances the training speed with compaction throughput.

## 4.4 Impact of Individual Techniques

**Ablation Study.** Figure 11 shows the impact of each component of Kirin on the total throughput and lookup latency. We use Bourbon and Kirin-Host, which is the host-only implementation of Kirin that does not utilize the processing capabilities of the CSD, as the baseline. *+Co-Search* enables the FPGA-based search engine, and

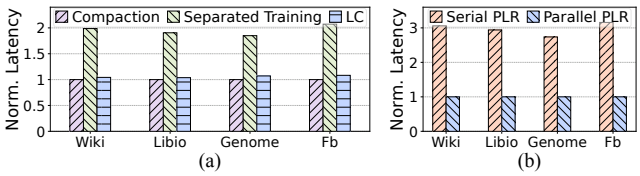


**Figure 12: (a) The impact of learned compaction accelerator. (b) The impact of PPLR accelerator.**

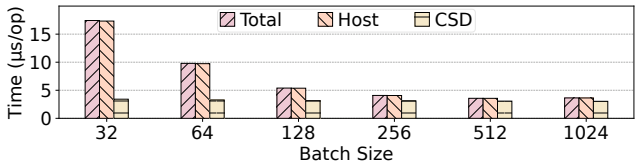


**Figure 13: Search time with different search batch sizes.**

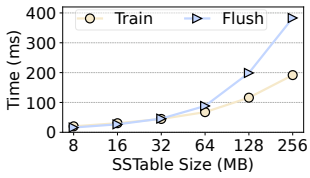


**Figure 14: Results of various SSTable sizes.**

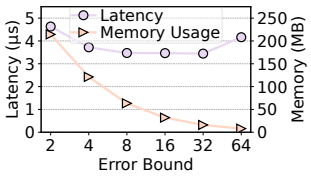


**Figure 15: Results of various error bounds.**

*+Co-LC* adds the learned compaction accelerator and collaborative learned compaction based on *+Co-Search*. We observe that the search engine reduces the read latency by 1.92× compared to Kirin-Host. The learned compaction accelerator significantly reduces the read latency and improves the total throughput by more efficiently retraining the learned index and processing compaction.

**Impact of Collaborative Learned Compaction.** Figure 12(a) presents the performance comparison of the learned compaction accelerator against compaction-only and the separated compaction and training approach on the CSD's FPGA. The results demonstrate that our learned compaction mechanism effectively overlaps training with compaction, thereby hiding most of the training time. This integration significantly improves overall throughput compared to the approach that performs compaction and training separately.

**Impact of PPLR.** We further evaluate the performance of the PPLR accelerator compared to the original PLR implementation. As shown in Figure 12(b), our design effectively mitigates the data dependency issues inherent in the serial PLR. As a result, the PPLR accelerator achieves an average speedup of 2.97× across various datasets compared to the baseline FPGA implementation of PLR.

## 4.5 Sensitivity Analysis

**Search Batch Size.** To evaluate the effectiveness of the search engine and the coordinated search mechanism between the host and the CSD, we vary the search batch size and measure the elapsed time for both sides. As depicted in Figure 13, the total search time per lookup operation predominantly depends on the host side, since

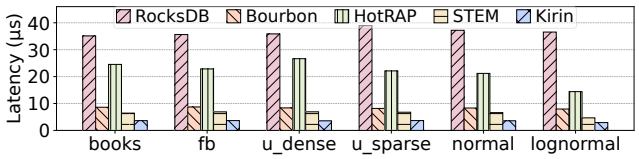


**Figure 16: Read latency of Kirin and baselines under SOSD.**

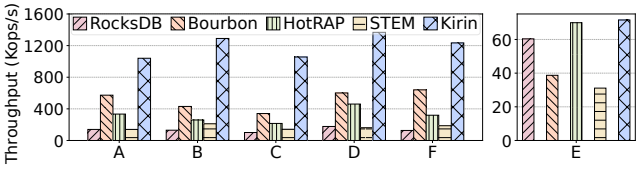


**Figure 17: Total throughput of Kirin and baselines under YCSB workloads.**

the CSD's search engine runs in parallel and thus its execution period is overlapped in our design. Meanwhile, the batch size has a negligible effect on the CSD's search engine because its random access pattern fully exploits the available internal bandwidth, even with relatively small batches.

However, the total search time decreases with larger batch sizes and stabilizes after exceeding 512. This trend occurs because larger batches amortize the communication time during offloading, while the time of collecting results and prediction becomes the dominant factor, scaling linearly with the batch size. Consequently, we choose a batch size of 512 for our experiments to maximize bandwidth utilization and achieve the best overall search efficiency.

**SSTable Size.** We evaluate the performance of Kirin using varying SSTable sizes, ranging from 8MB to 256MB. As illustrated in Figure 14, the training time becomes shorter than the flushing time as the SSTable size increases. Based on this observation, we set the SSTable size to 64MB, where the flushing time just overlaps with the training time, ensuring efficient learned compaction.

**Error Bound.** We vary the error bound of the learned index to evaluate the lookup latency and memory usage of Kirin. As shown in Figure 15, the lookup latency initially decreases with increasing error bound but starts to rise after the error bound exceeds 32. This trend occurs because a larger error bound results in fewer line segments, reducing the time to locate the appropriate segment. However, it also increases the number of candidate KVs that must be scanned to find the target key. Based on this trade-off, we set the error bound to 32 in Kirin to achieve the optimal balance between lookup latency and memory usage. The models occupy only 20 MB for 100 million keys (approx. 0.2 bytes per key), representing a negligible overhead compared to the raw data size.

## 4.6 Macrobenchmarks

We next analyze the performance of Kirin using two widely recognized macrobenchmarks: SOSD [31] and YCSB [6].

**SOSD.** We evaluate Kirin's performance using the SOSD benchmark, which is specifically designed to measure the efficiency of the learned index. Figure 16 presents the average lookup latency results, where Kirin achieves the lowest latency across all datasets.

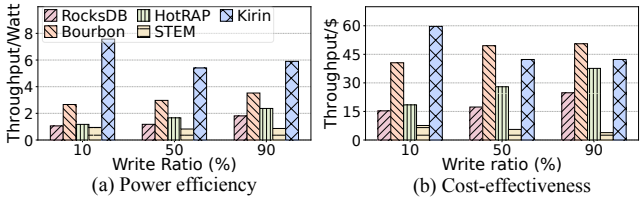


**Figure 18: Power efficiency and cost-effectiveness of Kirin and baselines.**

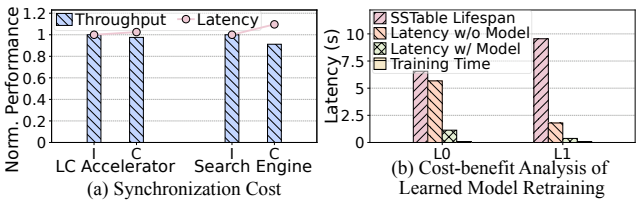


**Figure 19: Synchronization cost and cost-benefit analysis.**

Precisely, Kirin delivers latencies that are 2.23× to 2.73× lower than Bourbon, 9.69× to 12.56× lower than RocksDB, 4.95× to 7.49× lower than HotRAP, and 1.60× to 1.94× lower than STEM. These results highlight the efficiency of Kirin's learned index and its superior performance under read-intensive workloads, outperforming RocksDB and Bourbon. The search engine further alleviates the storage access bottleneck in search operations, enabling Kirin to significantly surpass STEM.

**YCSB.** We evaluate Kirin using six YCSB workloads.We load the YCSB default dataset generated by ycsb-load in random order. Figure 17 illustrates the total throughput of Kirin compared to other schemes across these workloads. It can be observed that Kirin achieves the highest throughput in all cases. For write-heavy workloads (A and F), Kirin delivers 2.24×, 9.37×, 3.79× and 7.54× higher throughput than Bourbon, RocksDB, HotRAP, and STEM, respectively. This improvement illustrates the effectiveness of our collaborative learned compaction framework in real-world benchmarks, enabling Kirin to handle write-heavy workloads more efficiently than baselines.

For read-intensive workloads B and D, Kirin outperforms the baselines by up to 9.92×. The advantage of Kirin is particularly pronounced in B, as the zipfian query distribution leads to more searches within SSTables, where the search engine substantially reduces storage access overhead. In the read-only workload C, Kirin achieves 1.55-4.51× higher throughput compared to the baselines. In E, which is a scan-heavy workload, Kirin achieves a modest improvement of 1.51× over RocksDB, as the primary bottleneck for scan operations is storage access overhead during value retrieval. In summary, Kirin excels in both read and write workloads, with extreme performance under write-heavy scenarios.

## 4.7 Overhead Analysis

**Power Efficiency.** We measure system power using PowerTop [19] and report throughput per watt ($\frac{\text{Throughput}}{Power_{Host}+Power_{Device}}$) in Figure 18(a). Kirin demonstrates the highest energy efficiency across all write ratios. Although its FPGA logic increases device dynamic power, the shorter execution time lowers total energy cost. Specifically, Kirin improves energy efficiency by up to 1.82× over Bourbon, 4.59× over RocksDB, 3.25× over HotRAP, and 6.55× over STEM.

**Cost-Effectiveness.** We also analyze the cost-effectiveness of each system, measured as throughput-per-dollar ($\frac{Throughput}{Cost_{Host}+Cost_{Device}}$). Our analysis is based on the approximate market prices of the system components: the host CPU (Intel Xeon Gold 6240), DRAM (Samsung DDR4 32GB), DaisyPlus OpenSSD at $2855 [26], $79 [58], and $6150 [7], respectively. All systems use two CPUs and 12 DRAM modules, while Kirin and STEM additionally use OpenSSD. As shown in Figure 18(b), Kirin is more cost-effective than the baselines, exceeding RocksDB by 2.44×, HotRAP by 1.51×, and STEM by 7.82× on average. Compared with Bourbon, the high cost of our DaisyPlus OpenSSD prototype limits Kirin's advantage in write-heavy scenarios, but this gap is expected to shrink as CSDs mature.

**Synchronization Cost.** To quantify the overhead of hardware synchronization, we evaluate the throughput and latency of the learned compaction accelerator and search engine under isolated ('I') and concurrent ('C') execution modes. As shown in Figure 19(a), Kirin demonstrates robust concurrency control. Under concurrency, the learned compaction accelerator experiences negligible performance fluctuation (approximately 2% variance), while the search engine retains over 91% of its peak throughput with a moderate 10% increase in latency. This is because the two accelerators share memory through AXI arbitration, avoiding software locks and major pipeline stalls.

**Cost-Benefit Analysis of Retraining.** To verify whether retraining learned models is justified for short-lived SSTables (Levels 0 and 1), we analyze the time breakdown of these operations under YCSB-A. Figure 19(b) presents the average SSTable lifespan, the accumulated lookup time with and without models, and the model training latency. The results indicate that the training overhead is negligible, occupying less than 1% of the SSTable's lifespan, whereas the absence of models leads to significantly higher overall lookup latency. Consequently, the substantial reduction in read latency significantly outweighs the minimal training cost, confirming that maintaining up-to-date models is both necessary and highly beneficial, even for short-lived L0/L1 SSTables.

## 4.8 Discussion

**Impact of Real NAND Flash.** Deploying Kirin on physical NAND flash introduces distinct physical behaviors compared to emulation. First, while real NAND exhibits latency variance due to Garbage Collection (GC), Kirin mitigates this via its sequential LSM-tree write patterns. This alignment with NAND's block-erasure property reduces internal fragmentation and GC frequency. Second, the high access cost on real media amplifies Kirin's benefits. By using learned predictions to bypass intermediate index block reads, Kirin eliminates entire flash read commands, yielding larger absolute latency savings on real hardware than in emulation. Finally, regarding contention on limited flash channels, Kirin's in-storage architecture offers distinct advantages. The accelerator's proximity to the flash controller enables tighter coordination of compaction and queries, improving resource management over host-side systems lacking device visibility.

**Impact of PCIe Evolution.** Although our evaluation employs PCIe 3.0, Kirin's collaborative architecture is designed to scale with newer standards such as PCIe 4.0 and 5.0. Higher bandwidth accelerates host-side data flushing and L0-L1 compaction migration, enabling tighter synchronization with the CSD, reducing idle time, and improving parallelism between host processors and in-storage accelerators. However, increased raw bandwidth does not resolve the granularity mismatch between block-based storage and small key-value accesses. Kirin's in-storage search improves effective goodput by ensuring that the high-speed interface transports only relevant data rather than excessive block overhead. Ultimately, as data volumes continue to outpace bandwidth growth, the PCIe link remains a contended resource, rendering Kirin's near-data processing strategy essential for scalability.

# 5 RELATED WORK

## 5.1 Learned Index

**Algorithm.** Since Kraska et al. [34] proposed learned index, linear models have been widely adopted for their efficiency. FITing-Tree [21] and RadixSpline [32] propose efficient linear segmentation models. PGM-index [20] introduces the optimal piecewise linear regression (PLR) algorithm, which guarantees error bounds [72] and is widely used in learned index applications [9, 43, 62, 68, 75]. However, PLR's $O(n)$ complexity limits training efficiency on large datasets. Existing methods have not addressed PLR acceleration. We exploit parallelism in PLR and develop a parallel FPGA accelerator to overcome this bottleneck.

**Retraining.** Learned indexes struggle with updates as key changes invalidate models. To address this, some studies buffer inserts [13] or periodically retrain [43]. Others like XIndex [65] and PGM-index [20] use LSM-like compaction to decouple writes from training. However, these incur training costs and data preparation overhead. In-place update mechanisms [21, 68] reduce training overhead but increase update complexity and degrade performance under write-heavy workloads. Unlike these, we integrate retraining into LSM-tree compaction, concealing training time and enabling model utilization. Our training module directly retrieves keys from compaction, eliminating data preparation and reducing data movement.

## 5.2 Optimizations of LSM-tree

LSM-trees are widely used in key-value stores because they efficiently handle write-heavy workloads. However, they also introduce challenges, particularly in terms of read performance and compaction overhead. To enhance search efficiency within SSTables, prior works have adopted machine learning techniques that can be categorized into learned indexes and learned filters. Bourbon [9], Google Bigtable [1], TridentKV [46] and LeaderKV [69] use space-efficient learned index models to replace the original index blocks, enabling faster KV position location. Meanwhile, Oasis [3] and SNARF [67] present learned range filters to exclude unnecessary searches efficiently, thereby enhancing query performance. Despite their advantages, these learned approaches impose model training overhead, which can degrade overall performance, especially under write-heavy workloads.

For compaction, several studies like REMIX [78] and Spooky [10] attempt to reduce the compaction size through a partial merge or tiered compaction strategy. Beyond software-level optimizations, hardware-level approaches have also been explored. Zhang et al. [77], Sun et al. [64] and STEM [66] employ FPGA to offload compaction. The gLSM [61] leverages a parallel compaction system on heterogeneous GPGPUs for KV stores, and DComp [12] utilizes a Data Processing Unit to offload compaction. While these offloading strategies alleviate the host's workload and accelerate compaction, they do not fully address the data movement overhead between host memory and storage.

Different from these studies, Kirin combines learned index to enhance query performance while leveraging CSD to accelerate compaction and retraining. By utilizing the FPGA on the CSD, we not only speed up these processes but also significantly reduce data movement by exploiting the CSD's internal bandwidth.

## 5.3 In-Storage Key-Value Systems

Current research on in-storage key-value stores primarily focuses on leveraging the computational capabilities of CSD controllers to minimize storage access overhead. A prominent direction in this field is KVSSD, which implements key-value storage directly on the CSD. LightStore [5], iLSM-SSD [37] and PinK [25] present LSM-tree based in-storage key-value engines, while Dotori [16] develops a novel B+ tree based KVSSD. Other works explore offloading specific tasks from the host to the CSD. For instance, KEVIN [33] proposes an LSM-tree-based in-storage engine for efficient indexing of files and directories within the file system, and POLARDB [2] utilizes FPGA-equipped CSDs to offload table scan operations.

These studies harness the computational potential of hardware like CSDs and FPGAs, but they fall short in optimizing task division between the host and CSD. This results in suboptimal use of system resources and an inability to fully capitalize on the strengths of both host and storage. In our design for key-value store systems, we adopt a collaborative approach to enable the host and device sides to work in parallel. This coordination maximizes resource utilization and ensures efficient execution across the host and CSD.

# 6 CONCLUSION

In this paper, we present Kirin, a hybrid index KV store born from a system-algorithm co-design that integrates LSM-tree with learned indexes, leveraging computational storage device (CSD) to offload and accelerate critical tasks. Kirin features learned compaction, embedding the training process into compaction to reduce data movement and conceal training overhead. Moreover, it adopts a collaborative framework between the host and CSD, parallelizing learned compaction tasks across levels and reducing storage access during searches. Our evaluations on the DaisyPlus OpenSSD platform demonstrate that Kirin significantly enhances both read and write performance of the hybrid index.

# ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful feedback. This work is supported by National Key R&D Program of China (Grant No. 2024YFB4505203), National Natural Science Foundation of China (NSFC) (Grant No. 62227809, 62332012, 62302290), Natural Science Foundation of Beijing (Grant No. 4264104) and the Fundamental Research Funds for the Central Universities.

# REFERENCES


[1] Hussam Abu-Libdeh, Deniz Altınbüken, Alex Beutel, Ed H Chi, Lyric Doshi, Tim Kraska, Andy Ly, Christopher Olston, et al. 2020. Learned indexes for a google-scale disk-based database. *arXiv preprint arXiv:2012.12501* (2020).

[2] Wei Cao, Yang Liu, Zhushi Cheng, Ning Zheng, Wei Li, Wenjie Wu, Linqiang Ouyang, Peng Wang, Yijing Wang, Ray Kuan, et al. 2020. POLARDB meets computational storage: Efficiently support analytical workloads in Cloud-Native relational database. In *18th USENIX conference on file and storage technologies (FAST 20)*. 29–41.

[3] Guanduo Chen, Zhenying He, Meng Li, and Siqiang Luo. 2024. Oasis: An Optimal Disjoint Segmented Learned Range Filter. *Proceedings of the VLDB Endowment* 17, 8 (2024), 1911–1924.

[4] Guoqiang Jerry Chen, Janet L Wiener, Shridhar Iyer, Anshul Jaiswal, Ran Lei, Nikhil Simha, Wei Wang, Kevin Wilfong, Tim Williamson, and Serhat Yilmaz. 2016. Realtime data processing at facebook. In *Proceedings of the 2016 International Conference on Management of Data*. 1087–1098.

[5] Chanwoo Chung, Jinhyung Koo, Junsu Im, Arvind, and Sungjin Lee. 2019. Lightstore: Software-defined network-attached key-value drives. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 939–953.

[6] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.

[7] CRZ Technology. 2021. DaisyPlus OpenSSD Platform. Retrieved May 22, 2026 from https://www.crz-tech.com/crz/article/DaisyPlus/

[8] CRZ Technology. 2021. OpenSSD projects. Retrieved May 22, 2026 from https://github.com/CRZ-Technology/OpenSSD-OpenChannelSSD

[9] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2020. From WiscKey to Bourbon: A Learned Index for Log-Structured Merge Trees. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 155–171.

[10] Niv Dayan, Tamar Weiss, Shmuel Dashevsky, Michael Pan, Edward Bortnikov, and Moshe Twitto. 2022. Spooky: granulating LSM-tree compactions correctly. *Proceedings of the VLDB Endowment* 15, 11 (2022), 3071–3084.

[11] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS operating systems review* 41, 6 (2007), 205–220.

[12] Chen Ding, Jian Zhou, Jiguang Wan, Yiqin Xiong, Sicen Li, Shuning Chen, Hanyang Liu, Liu Tang, Ling Zhan, Kai Lu, et al. 2023. Dcomp: Efficient offload of lsm-tree compaction with data processing units. In *Proceedings of the 52nd International Conference on Parallel Processing*. 233–243.

[13] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, et al. 2020. ALEX: an updatable adaptive learned index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 969–984.

[14] Dormando. 2019. memcached: A Distributed Memory Object Caching System. Retrieved May 22, 2026 from https://memcached.org/

[15] Zhuohui Duan, Hao Feng, Haikun Liu, Xiaofei Liao, Hai Jin, and Bangyu Li. 2025. AegonKV: A High Bandwidth, Low Tail Latency, and Low Storage Cost KV-Separated LSM Store with SmartSSD-based GC Offloading. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*. 321–335.

[16] Carl Duffy, Jaehoon Shim, Sang-Hoon Kim, and Jin-Soo Kim. 2023. Dotori: A key-value ssd based kv store. *Proceedings of the VLDB Endowment* 16, 6 (2023), 1560–1572.

[17] Facebook. 2022. RocksDB. Retrieved May 22, 2026 from http://rocksdb.org

[18] Facebook. 2023. RocksDB Tuning Guide. Retrieved May 22, 2026 from https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide

[19] fenrus75. 2022. PowerTop. Retrieved May 22, 2026 from https://github.com/fenrus75/powertop.git

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

[21] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. Fiting-tree: A data-aware index structure. In *Proceedings of the 2019 international conference on management of data*. 1189–1206.

[22] Anil K Goel, Jeffrey Pound, Nathan Auch, Peter Bumbulis, Scott MacLean, Franz Färber, Francis Gropengiesser, Christian Mathis, Thomas Bodner, and Wolfgang Lehner. 2015. Towards scalable real-time analytics: An architecture for scale-out of OLxP workloads. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1716–1727.

[23] Google. 2021. LevelDB. Retrieved May 22, 2026 from https://github.com/google/leveldb

[24] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. 2019. X-Engine: An optimized storage engine for large-scale E-commerce transaction processing. In *Proceedings of the 2019 International Conference on Management of Data*. 651–665.

[25] Junsu Im, Jinwook Bae, Chanwoo Chung, Sungjin Lee, et al. 2020. PinK: High-speed In-storage Key-value Store with Bounded Tails. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 173–187.

[26] Intel. 2019. Xeon Gold 6240 @2.6GHz. Retrieved May 22, 2026 from https://www.intel.com/content/www/us/en/products/sku/192443/intel-xeon-gold-6240-processor-24-75m-cache-2-60-ghz/specifications.html

[27] Hongsun Jang, Jaeyong Song, Jaewon Jung, Jaeyoung Park, Youngsok Kim, and Jinho Lee. 2024. Smart-Infinity: Fast Large Language Model Training using Near-Storage Processing on a Real System. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 345–360.

[28] Sudarsun Kannan, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, Yuangang Wang, Jun Xu, and Gopinath Palani. 2018. Designing a true Direct-Access file system with DevFS. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*. 241–256.

[29] Minsu Kim, Jinwoo Hwang, Guseul Heo, Seiyeon Cho, Divya Mahajan, and Jongse Park. 2024. Accelerating String-Key Learned Index Structures via Memoization-Based Incremental Training. *Proceedings of the VLDB Endowment* 17, 8 (2024), 1802–1815.

[30] Shine Kim, Yunho Jin, Gina Sohn, Jonghyun Bae, Tae Jun Ham, and Jae W Lee. 2021. Behemoth: a flash-centric training accelerator for extreme-scale DNNs. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 371–385.

[31] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2019. SOSD: A benchmark for learned indexes. *arXiv preprint arXiv:1911.13014* (2019).

[32] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2020. RadixSpline: a single-pass learned index. In *Proceedings of the third international workshop on exploiting artificial intelligence techniques for data management*. 1–5.

[33] Jinhyung Koo, Junsu Im, Jooyoung Song, Juhyung Park, Eunji Lee, Bryan S Kim, and Sungjin Lee. 2021. Modernizing file system through in-storage indexing. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. 75–92.

[34] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *Proceedings of the 2018 international conference on management of data*. 489–504.

[35] Miryeong Kwon, Donghyun Gouk, Sangwon Lee, and Myoungsoo Jung. 2022. Hardware/Software Co-Programmable framework for computational SSDs to accelerate deep learning service on Large-Scale graphs. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. 147–164.

[36] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS operating systems review* 44, 2 (2010), 35–40.

[37] Chang-Gyu Lee, Hyeongu Kang, Donggyu Park, Sungyong Park, Youngjae Kim, Jungki Noh, Woosuk Chung, and Kyoung Park. 2019. iLSM-SSD: An intelligent LSM-tree based key-value SSD for data analytics. In *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 384–395.

[38] Kitaek Lee, Insoon Jo, Jaechan Ahn, Hyuk Lee, Hwang Lee, Woong Sul, and Hyungsoo Jung. 2023. Deploying Computational Storage for HTAP DBMSs Takes More Than Just Computation Offloading. *Proceedings of the VLDB Endowment* 16, 6 (2023), 1480–1493.

[39] Yunjae Lee, Jinha Chung, and Minsoo Rhu. 2022. Smartsage: training large-scale graph neural networks using in-storage processing architectures. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 932–945.

[40] Yunjae Lee, Hyeseong Kim, and Minsoo Rhu. 2024. PreSto: An In-Storage Data Preprocessing System for Training Recommendation Models. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 340–353.

[41] Jure Leskovec, Lars Backstrom, and Jon Kleinberg. 2009. Meme-tracking and the dynamics of the news cycle. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. 497–506.

[42] Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Matias Bjørling, and Haryadi S Gunawi. 2018. The CASE of FEMU: Cheap, accurate, scalable and extensible flash emulator. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*. 83–90.

[43] Pengfei Li, Yu Hua, Pengfei Zuo, Zhangyu Chen, and Jiajie Sheng. 2023. ROLEX: A Scalable RDMA-oriented Learned Key-Value Store for Disaggregated Memory Systems. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. 99–114.

[44] Yongkun Li, Zhen Liu, Patrick PC Lee, Jiayu Wu, Yinlong Xu, Yi Wu, Liu Tang, Qi Liu, and Qiu Cui. 2021. Differentiated Key-Value storage management for balanced I/O performance. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 673–687.

[45] Hyeontaek Lim, Bin Fan, David G Andersen, and Michael Kaminsky. 2011. SILT: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 1–13.

[46] Kai Lu, Nannan Zhao, Jiguang Wan, Changhong Fei, Wei Zhao, and Tongliang Deng. 2021. 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 (2021), 1953–1966.

[47] Kiran Kumar Matam, Gunjae Koo, Haipeng Zha, Hung-Wei Tseng, and Murali Annavaram. 2019. GraphSSD: graph semantics aware SSD. In *Proceedings of the 46th international symposium on computer architecture*. 116–128.

[48] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S Berger, Nathan Beckmann, and Gregory R Ganger. 2021. Kangaroo: Caching billions of tiny objects on flash. In *Proceedings of the ACM SIGOPS 28th symposium on operating systems principles*. 243–262.

[49] Pankaj Mehra. 2019. Samsung smartssd: Accelerating data-rich applications. *Flash Memory Summit* (2019).

[50] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. 2013. Scaling memcache at facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. 385–398.

[51] NVM Express. 2025. NVM Express. Retrieved May 22, 2026 from https://nvmexpress.org/

[52] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33 (1996), 351–385.

[53] Xiurui Pan, Endian Li, Qiao Li, Shengwen Liang, Yizhou Shan, Ke Zhou, Yingwei Luo, Xiaolin Wang, and Jie Zhang. 2025. InstAttention: In-Storage Attention Offloading for Cost-Effective Long-Context LLM Inference. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 1510–1525.

[54] Junhyeok Park, Chang-Gyu Lee, Soon Hwang, Soonyeal Yang, Jungki Noh, Woosuk Chung, Junghee Lee, and Youngjae Kim. 2024. BandSlim: A Novel Bandwidth and Space-Efficient KV-SSD with an Escape-from-Block Approach. In *Proceedings of the 53rd International Conference on Parallel Processing*. 1187–1196.

[55] Jiansheng Qiu, Fangzhou Yuan, Mingyu Gao, and Huanchen Zhang. 2025. HotRAP: hot record retention and promotion for LSM-trees with tiered storage. In *Proceedings of the 2025 USENIX Conference on Usenix Annual Technical Conference*. 497–511.

[56] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 497–514.

[57] Zhenyuan Ruan, Tong He, and Jason Cong. 2019. INSIDER: Designing In-Storage computing system for emerging High-Performance drive. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 379–394.

[58] Samsung. 2019. DDR4 32GB DRAM. Retrieved May 22, 2026 from https://www.amazon.com/-/zh_TW/ARS948/dp/B0B6GR5WMB

[59] ScaleFlux. 2025. ScaleFlux's CSD. Retrieved May 22, 2026 from https://scaleflux.com/products/csd-5000/

[60] SNIA. 2024. SNIA Computational Storage Technical Work Group (TWG). Retrieved May 22, 2026 from https://www.snia.org/computationaltw

[61] Hui Sun, Jinfeng Xu, Xiangxiang Jiang, Guanzhong Chen, Yinliang Yue, and Xiao Qin. 2024. gLSM: Using GPGPU to Accelerate Compactions in LSM-tree-based Key-value Stores. *ACM Transactions on Storage* 20, 1 (2024), 1–41.

[62] Jinghan Sun, Shaobo Li, Yunxin Sun, Chao Sun, Dejan Vucinic, and Jian Huang. 2023. 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, Volume 2*. 442–456.

[63] Penghao Sun, Litong You, Shengan Zheng, Wanru Zhang, Ruoyan Ma, Jie Yang, Guanzhong Wang, Feng Zhu, Shu Li, and Linpeng Huang. 2023. Learning-based Data Separation for Write Amplification Reduction in Solid State Drives. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[64] Xuan Sun, Jinghuan Yu, Zimeng Zhou, and Chun Jason Xue. 2020. FPGA-based compaction engine for accelerating LSM-tree key-value stores. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1261–1272.

[65] Chuzhe Tang, Youyun Wang, Zhiyuan Dong, Gansen Hu, Zhaoguo Wang, Minjie Wang, and Haibo Chen. 2020. XIndex: a scalable learned index for multicore data storage. In *Proceedings of the 25th ACM SIGPLAN symposium on principles and practice of parallel programming*. 308–320.

[66] Dongdong Tang, Weilan Wang, Yu Mao, Jinghuan Yu, Tei-Wei Kuo, and Chun Jason Xue. 2024. STEM: Streaming-Based FPGA Acceleration for Large-Scale Compactions in LSM KV. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 3893–3905.

[67] Kapil Vaidya, Subarna Chatterjee, Eric Knorr, Michael Mitzenmacher, Stratos Idreos, and Tim Kraska. 2022. SNARF: a learning-enhanced range filter. *Proceedings of the VLDB Endowment* 15, 8 (2022), 1632–1644.

[68] Shengzhe Wang, Zihang Lin, Suzhen Wu, Hong Jiang, Jie Zhang, and Bo Mao. 2024. 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)*. IEEE, 616–629.

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

[70] Mark Wilkening, Udit Gupta, Samuel Hsia, Caroline Trippel, Carole-Jean Wu, David Brooks, and Gu-Yeon Wei. 2021. RecSSD: near data processing for solid state drive based recommendation inference. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 717–729.

[71] Chaichon Wongkham, Baotong Lu, Chris Liu, Zhicong Zhong, Eric Lo, and Tianzheng Wang. 2022. Are updatable learned indexes ready? *Proceedings of the VLDB Endowment* 15, 11 (2022), 3004–3017.

[72] Qing Xie, Chaoyi Pang, Xiaofang Zhou, Xiangliang Zhang, and Ke Deng. 2014. Maximum error-bounded piecewise linear representation for online stream approximation. *The VLDB journal* 23 (2014), 915–937.

[73] Zhe Yang, Youyou Lu, Xiaojian Liao, Youmin Chen, Junru Li, Siyu He, and Jiwu Shu. 2023. $\lambda$-IO: A Unified IO Stack for Computational Storage. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. 347–362.

[74] Jinghuan Yu, Sam H Noh, Young-ri Choi, and Chun Jason Xue. 2023. 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 23)*. 65–80.

[75] Ce Zhang, Cheng Xu, Haibo Hu, and Jianliang Xu. 2024. COLE: A Column-based Learned Storage for Blockchain Systems. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*. 329–345.

[76] Jian Zhang, Yujie Ren, and Sudarsun Kannan. 2022. FusionFS: Fusing I/O Operations using CISCOps in Firmware File Systems. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. 297–312.

[77] Teng Zhang, Jianying Wang, Xuntao Cheng, Hao Xu, Nanlong Yu, Gui Huang, Tieying Zhang, Dengcheng He, Feifei Li, Wei Cao, et al. 2020. FPGA-Accelerated Compactions for LSM-based Key-Value Store. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 225–237.

[78] Wenshao Zhong, Chen Chen, Xingbo Wu, and Song Jiang. 2021. REMIX: Efficient Range Query for LSM-trees. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 51–64.

[79] Zeying Zhu, Yibo Zhao, and Zaoxing Liu. 2024. In-Memory Key-Value Store Live Migration with NetMigrate. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*. 209–224.