RecDB: An LSM-Tree based Storage System for Training Large Recommendation Model in Low-Resource Scenarios

Ming Gao
Sun Yat-Sen University
Guangzhou, Guangdong, China
gaom59@mail2.sysu.edu.cn

Qingyin Lin
Sun Yat-Sen University, Peng Cheng Laboratory
Guangzhou, Guangdong, China
linqy35@mail2.sysu.edu.cn

Zhitao Chen
Sun Yat-Sen University
Guangzhou, Guangdong, China
chenzht37@mail2.sysu.edu.cn

Yunling Chen
Sun Yat-Sen University
Guangzhou, Guangdong, China
chenyling65@mail2.sysu.edu.cn

Zhiguang Chen
Sun Yat-Sen University
Guangzhou, Guangdong, China
chenzhg29@mail.sysu.edu.cn

Abstract

Deep learning based recommendation models have achieved significant commercial success. However, as the size of embedding tables in these models grows dramatically, how to store these massive embedding tables cost-effectively becomes a major concern for practitioners equipped with limited resources. Offloading these tables to external storage (e.g., solid-state drives) during training has emerged as a promising approach to reduce resource expenditures. Unfortunately, the training process brings significant random data accesses to these tables, which are not friendly for most external storage devices, resulting in suboptimal training performance.

Therefore, we propose RecDB, a storage engine for effectively storing embedding tables in solid-state drives. Basically, RecDB leverages the Log-Structured Merge-Tree (LSM-Tree) to convert random writes to sequential writes. While achieving good write performance, simply applying the LSM-Tree brings about issues in terms of poor read performance and wasted storage space, due to its out-of-place update mechanism. Firstly, a request preprocessor is designed to effectively gather frequently-accessed data and avoid loading data repeatedly, reducing random reads and improving the read performance. Then, a compaction picker is applied to select files that contain much invalid data to remove invalid data as much as possible, saving storage space. Furthermore, the background compaction operations of the LSM-Tree interfere with read requests due to resource contention. RecDB utilizes a compaction scheduler to carefully trigger compaction operations and avoid overlap between reads and compactions. Experimental results demonstrate that RecDB achieves $1.03\times - 2.94\times$ end-to-end training speedup compared to RocksDB, a well-known LSM-Tree based storage system.

Keywords

Recommendation Model, Log-Structured Merge-Tree, Resource-Constrained Training

1 Introduction

Deep recommendation models have become an integral part of numerous online services, including social media, e-commerce, and search engines [17, 19, 35, 41]. Deep recommendation models usually contain large embedding tables, which stores massive information of items interacted with users. Therefore, the size of embedding tables is largely related to the performance of the models. Recent researches of deep recommendation models reveal a common trend: the size of embedding tables is continuing to grow[21]. Taking Facebook's DLRM[32] as an example, the size of its embedding tables has progressively scaled from a few gigabytes to several terabytes. This trend makes deep recommendation model training a costly and challenging task for small vendors and researchers with limited memory resources.

To compensate for the lack of GPU memory, many works manage to offload embedding tables to CPU memory [9, 23, 28, 29, 36] or to persistent storage devices [13, 25, 38, 45, 47, 48]. Compared to general persistent storage devices, such as Solid-State Drives (SSDs) and Hard Disk Drives (HDDs), CPU memory provides acceptable bandwidth for recommendation model training. However, due to the limited capacity, high cost and volatility of the CPU memory, offloading to SSDs is a more cost-efficient solution for small vendors and researchers. For example, a server with 1.4TB of CPU memory costs $10.6 per hour in Google cloud, while a server with a 1.4TB SSD and limited 32GB of CPU memory only costs $0.6 per hour. In recent years, SSDs have been offering faster read and write speeds, making it a more popular choice for storing embedding tables. In this case, embedding parameters needed for training are loaded on demand to reduce the I/O and CPU memory overhead. However, the embedding table accesses exhibit strong randomness, which is unfriendly to block devices like SSDs. As shown in Table 1, the random read and write bandwidths of our SSD are significantly lower than the sequential read and write bandwidths. Therefore, it is crucial to carefully schedule read and write requests and optimize data placement on the SSDs.

Many efforts have been made to bridge the gap between the SSD bandwidth and model training requirements, attempting to reduce data accesses in SSDs. Firstly, near-data processing offloads data-related computation to SSDs [22, 26, 38, 40, 47]. NDRec [26] offloads both the parameters and the computation of the embedding layer to computational storage device. Jang [22] performs the reduce operation within the near-data processing unit (e.g. smartSSD) to minimize the data that move across the PCIe interface. Secondly, CPU memory acts as cache to buffer selected embedding parameters. EVStore [25] replaces embedding vectors in SSDs with approximate vectors cached in memory, which largely improves cache hit rate. Ehsan k [10] leverages the access skewness of embedding vectors and caches hot (frequently-accessed) embedding vectors in memory. RecRT [14] improves cache hit rate by looking ahead and evicting the latest accessed data. MTrainS [24] further introduces mixed-precision cache to

EDBT '26, Tampere (Finland)

10.48786/edbt.2026.37

Table 1: SSD access bandwidth comparison of different workload [6].

Block size	Workload	Read (GB/s)	Write (GB/s)
128KB	Sequential	2.30	1.55
4KB	Random	1.91	1.34

improve cache efficiency. In summary, these works improve cache hit rate, but either sacrificing accuracy or ignoring the random access pattern of the embedding parameters.

In addition to limited bandwidth, storing embedding tables in SSDs also faces two additional challenges. Basically, we assume that updated parameters are written to SSDs in place of their old values. Firstly, SSDs' inherent write amplification increases the update latency. Typically, the write granularity of SSDs is 4KB and updating an embedding vector of 144 bytes requires writing the entire 4KB flash block. The updated vectors fall in random blocks, exacerbating the impact of write amplification. One way to mitigate this problem is to perform out-of-place updates. AIbox [48] maps each embedding vector to its address in SSDs through hash indexing, flushes updated vectors to new addresses, and updates the hash index. However, hash indexing incurs large memory footprint. For example, indexing 10^{10} embedding vectors demands 149 GB of CPU memory. Secondly, the read amplification of SSDs reduces reading performance. Since the read granularity of SSDs is a block, loading a single embedding vector requires reading an entire block, wasting SSD bandwidth. OC-DLRM [39] gathers hot embedding vectors in the same flash block to reduce overall SSD accesses. However, OC-DLRM relies on Open-Channel SSD to handle the internal changes of physical address in the SSD while keeping the logical address unchanged.

The Log-Structured Merge-Tree (LSM-Tree) offers a good chance to address these challenges with small CPU memory overhead. LSM-Tree buffers random updates in memory and flushes them into SSDs through sequential writes, which significantly improves the efficiency of embedding updates. Previous works [24, 46] adopt LSM-Tree to index embedding parameters in SSDs, while ignoring the challenges brought by LSM-Tree. Though with high write performance and small memory footprint, the LSM-Tree is yet unable to meet model training requirements. Firstly, the LSM-Tree lacks the ability to reorder the random accessing requests given by the training work, which reduces the efficiency of LSM-Tree's block cache. Secondly, LSM-Tree introduces the accumulation of outdated data, which increases the reading path in LSM-Tree. Thirdly, frequent compaction operations inside the LSM-Tree interfere with both read and write performances. Therefore, it is important to carefully improve LSM-Tree to provide high throughput for model training.

In this paper, we present RecDB, a LSM-Tree based key-value store specifically designed for recommendation model training, which largely mitigate the impacts of random writes and reads to SSDs due to parameter update and lookup. RecDB consists of three key components: a Read/Write Request Preprocessor and Scheduler, a Compaction Picker and a Compaction Scheduler. These components work together to reduce random accesses, remove useless outdated values, and mitigate the interference of compactions to read and write requests. RecDB improves read and write performance with small memory footprint, making it more cost-effective for small vendors and researchers to offload embedding parameters to SSDs during model training.

The contributions of this work are summarized as follows:

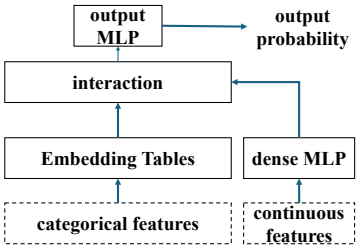


Figure 1: The architecture of DLRM.

- We design a Request Processor to gather hot embedding vectors together in data blocks and reduce the number of data block accesses, by creatively adding a prefix to the keys and reordering the read requests.
- We propose a Compaction Picker to save storage space and reduce reading paths in the LSM-Tree by selecting the most outdated file to perform compaction, which is identified during key lookup.
- We further design a Compaction Scheduler to prevent the background compaction from blocking the read requests by carefully excluding the compaction operation from the prefetching operation.
- Our evaluations indicate that RecDB significantly improves read and write performance when offloading embedding tables to SSDs during recommendation model training, achieving 1.03× ∼ 2.94× end-to-end speedup compared to the baseline approach (RocksDB). And RecDB has better tail latency and space consumption than SILK and SplinterDB.

The rest of the paper is organized as follows. Section 2 introduces the background on embedding table lookup and update for recommendation model training, the data access characteristics of SSDs and Log-Structured Merge-Tree. Section 3 illustrates the challenges and opportunities for embedding table lookup and update on SSDs. Section 4 describe the design of the proposed RecDB. Section 5 shows the experimental results for RecDB. Section 7 presents the conclusion.

2 BACKGROUND

2.1 Recommendation Model Training with SSDs

Recommendation models are widely used by modern online services to rank products, recommend advertisements according to users' behavior and preferences. Recently studied recommendation models basically comprise multi-layer perceptron (MLP) and embedding tables [13, 15, 20, 32, 43]. Taking widely studied DLRM [32] in Fig. 1 as an example, first it maps categorical features (e.g. user ID, item ID) to correspond embedding vector using embedding tables, while continuous features (e.g. users' age, rating and income) are processed by a multi-layer perceptron (called dense MLP). Next DLRM takes the feature interaction between all pairs of embedding vectors and processed continuous features and then passes the result to another MLP (called output MLP). Generally, the recommendation system contains billions of product items and embedding tables occupy substantial space. However, these tables require little computational resources, making it an ineffective choice to store all the embedding tables in memory during training. More and more works

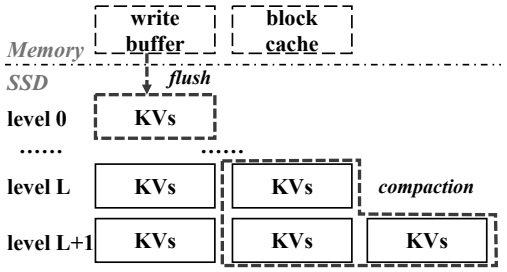


Figure 2: The architecture of LSM-Tree.

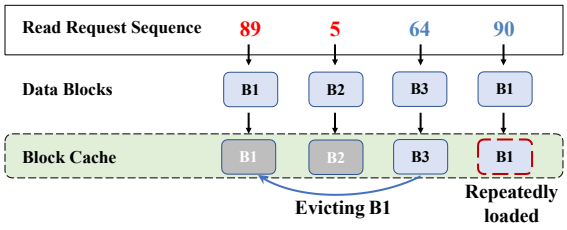


Figure 3: Repeated block loading induced by random reads.

aim to offload embedding tables to SSDs when training these tables [25, 38, 45, 47, 48]. When training embedding tables in SSDs, the training process can be divided into three key phases: embedding lookup, computation , and embedding update. Computation phase stresses computer capabilities to calculate the gradients of each parameter. On the other hand, embedding lookup and embedding update phases stress bandwidth by introducing random read and random write to embedding tables, because the purchased product lists or clicked advertisement lists generated by different users are highly random and varied.

However, these random access patterns significantly delay lookup and update embedding vector in SSDs. Specifically, the update embedding vectors are randomly scattered across different blocks mixed with unused data. Even if we only update 144-byte vector in-place within a flash block, the entire block must be read and rewritten. Similarly, when reading a small embedding vector from SSDs, an entire block must be read. This inefficiency not only increases the latency of embedding lookup but also wastes SSD bandwidth, further slowing down the training for recommendation model.

2.2 Log-Structured Merge-Tree (LSM-Tree)

Log-Structured Merge-Tree (LSM-Tree) [33] is a tree-like key-value store, which transforms multiple random writes into one sequential write and provides a promising opportunity to manage the embedding tables offloaded to SSDs. To store embedding tables, the index of embedding vector serves as the key, while the embedding vector itself is stored as value. As shown in Fig. 2, when we update an embedding vector, LSM-Tree first buffers it in the write buffer. And then multiple buffered update vectors are flushed into SSDs as an immutable file. These files are organized as a tree structure, where the newer files are stored in the shallow levels of tree. The files at shallow levels are gradually merged into the deep levels through a process named compaction. When we look up an embedding vector, LSM-Tree starts from the write buffer and traverses the tree from shallow level to deep level. When LSM-Tree finds the first matching embedding vector, the corresponding data block is loaded into a block cache, which replaces data with LRU policy [1]. To further reduce the cost of reading, LSM-Tree maintains bloom filters in the block cache, allowing a reading to skip probing a SSD file altogether, if the filter returns negative [18].

Generally speaking, LSM-Tree prioritizes write performance by sacrificing read efficiency. Data updates and insertions are performed by writing to new files , resulting in data disorder among files and the accumulation of outdated data. This pattern complicates the reading path of the LSM-Tree. To improve read performance and save space, the LSM-Tree sorts and merges data among different files and purges the outdated data through compactions. For example, when data at level L accumulates to a specific extent, the LSM-Tree picks a file at L and merges it with files at next level $L+1$ whose key range overlaps with the selected file. Typically, the file with the largest size at L is selected to perform compaction. During compactions, key-value pairs among all the participating files are sorted in key order and only the newly written pair is retained in case of key duplication. After compaction, the sorted key-value pairs are written to new files at level $L+1$. In this way, read efficiency is improved. However, to meet the requirements of recommendation model training, we find several challenges for the LSM-Tree to store embedding tables.

3 CHALLENGES & MOTIVATION

3.1 Challenges

Considering the inherent behaviors of the LSM-Tree, several challenges emerge when performing lookup and update operations on the embedding tables.

3.1.1 Read Amplification. Firstly, one block can be repeatedly loaded into memory cache even within single iteration due to random read workloads. Prior work has explored various intervention methods. For example, cache compression [12] achieves larger caches without the main drawbacks of physically increasing the caches themselves. Active and inactive lists [30, 37] protect valuable data from premature eviction. Nevertheless, prior research has largely focused on exploiting inter-block temporal and spatial locality, while the spatial locality within individual cache blocks remains underexplored. Fig. 3 gives an extreme example to illustrate this problem. We assume that the block cache size is 2 and each block can contain 144 embedding vectors, include one frequently accessed (e.g. 89) vector and 133 cold vectors (e.g. 64). Given the read request sequence issued by recommendation model training, data blocks containing these keys are loaded into memory and stored in the block cache. Firstly, $B1$ and $B2$ are loaded into cache one after another, reaching the full size of the cache. Then, when $B3$ is loaded into cache, $B1$ is evicted from the cache according to the LRU principle. Subsequently, $B1$ is loaded to cache once again, demonstrating the inefficiency of the cache. This is because the read request sequence is disordered while data in the same block is ordered, and the block cache lacks the ability to perceive read requests. Clearly, if the frequently accessed vectors 89 and 5 were co-located in the same block (e.g., $B1$), the intra-block spatial locality would be significantly enhanced and read amplification can be reduced. However, as shown in Fig. 5, up to 47% of data blocks are loaded repeatedly within a single iteration in our experiments.

Secondly, the read amplification from loading useless data is intensified by the access pattern. Based on our investigation,

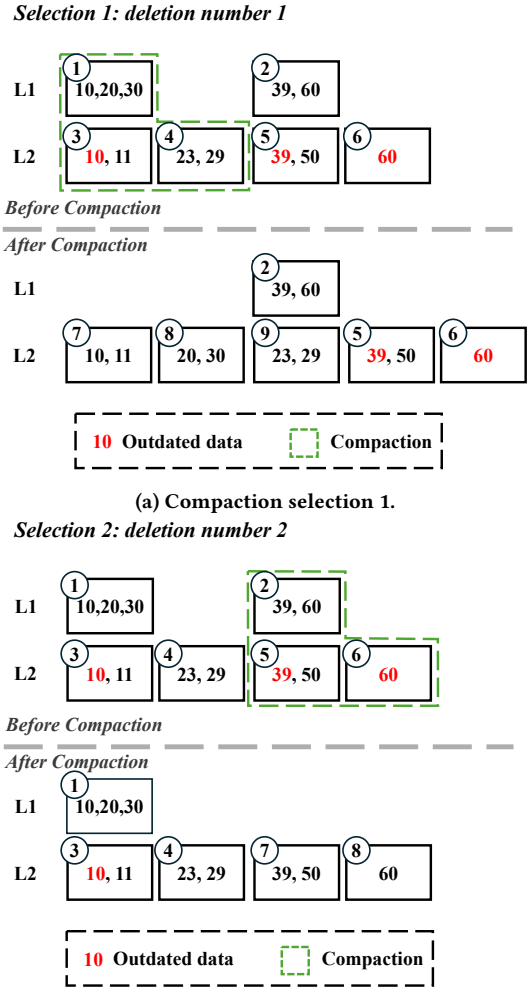


Figure 4: Different compaction selections lead to different deletion efficiency.

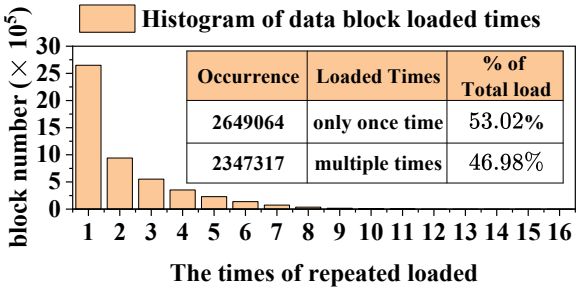


Figure 5: Data blocks loaded pattern.

the access to embedding vectors during model training exhibits a skewed distribution. More than 99% of the accesses are contributed to 1% of the embedding vectors. These hot embedding vectors are accessed frequently in adjacent iterations, while the other cold embeddings are rarely repeatedly accessed within a relatively long interval, for example, 200 iterations. The LSM-Tree lacks the ability to identify hot embedding vectors and mixes hot and cold embedding vectors in the same block. In this way, blocks containing hot embeddings are frequently loaded, carrying with the recently unused cold embeddings, wasting SSD bandwidth and cache space. Therefore, the block cache is unable to cache all the hot embedding vectors under memory-constrained environments, further intensifying the read amplification.

In order to illustrate this intuitively, we conduct experiments to evaluate the extent of read amplifications. Here we obtain the read amplification degree by dividing the volume of actually loaded data with that of the useful data. The read amplification of the SSD is 28, while that of the LSM-Tree is 124, increased by more than 4 times.

3.1.2 Accumulation of outdated data. The accumulation of outdated data wastes SSD space, especially under the update-intensive workload during recommendation model training. However, the original compaction selection strategy results in inefficient outdated data deletion. Fig. 4 compares the efficiency of deleting outdate data across different compaction selection strategies. As shown in the left of Fig. 4, the strategy selects the largest file ① at $L1$ as compaction input, merges it with files ③ and ④ at $L2$, and ultimately deletes one outdated data. On the right side of Fig. 4, the strategy selects the smaller file ② at $L1$ to merge with files ⑤ and ⑥ at $L2$ and finally deletes two outdated data. Fig. 4 indicates the fact that different compaction selections result in varying efficiencies of outdated data deletion. However, original LSM-Tree cannot make optimal compaction selections without the location information of outdated data, leading to significant space amplification.

3.1.3 Compaction interference. The compaction threads can interfere with the read threads generated by embedding lookup, reducing embedding lookup efficiency [42]. In low-resource scenarios, CPU cores and SSD bandwidth are highly valuable. When performing embedding lookup, we prefer that only the read threads about embedding lookup utilize these limited resources. Unfortunately, the LSM-Tree blindly schedules compaction whenever the size of a specific level exceeds its threshold, leading to resource contention between compaction tasks and reading requests. Fig. 6 shows the impact of compactions on reading latency. The x-axis (compaction size) indicates varying interference extent. A larger compaction size implies that more data are loaded to perform compaction, meaning a higher degree of compaction interference. With compaction, we track the compaction size within single iteration and record the read latency. Without compaction, although the read latency is affected by total data amount in the LSM-Tree, it is not affected by compaction size and we present average read latency here. Evaluation results show that the read latency can be increased by up to 258% compared to the case without compaction.

3.2 Motivation

Through the comprehensive analysis of the embedding table access pattern and the working process of the LSM-Tree, we identify several opportunities to improve the performance of the key-value store for storing embedding tables.

3.2.1 Predictable Training. During recommendation model training, the future embedding indices in subsequent iterations are predictable in both offline and online training scenarios. This gives us an opportunity to overlap embedding lookup latency by prefetching embedding vectors from SSDs. In offline model training, it is straightforward to foresee training data, given a

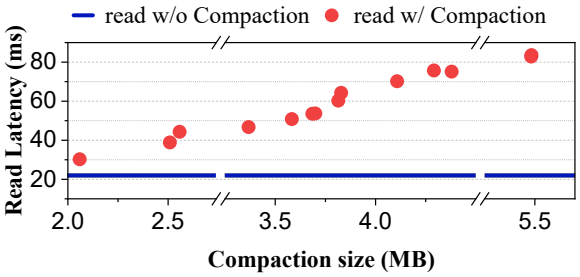


Figure 6: Reading latency under different compaction files size.

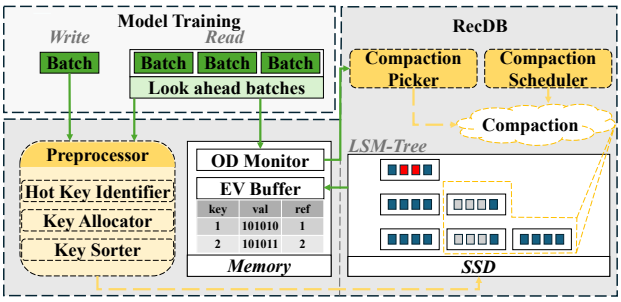


Figure 7: RecDB for Recommendation Model Training.

static dataset. Similarly, in online training, user behaviours accumulated throughout a day are collected as a dataset firstly, and then applied to incrementally train the recommendation model, making it predictable. Therefore, an intuitive way is to overlap the latency of embedding lookup with the computation and embedding update time.

Although many works also leverage prefetching to reduce end-to-end training time [28, 48], they only prefetch embeddings for the next batch. When offloading embedding tables to SSDs, the prefetching latency is hard to be fully overlapped. For example, AIbox [48] uses 3-stage pipeline to prefetch next batch embedding vectors. However, 22% prefetch latency is exposed although it uses a large memory to buffer embedding vectors. This is because the previous prefetching methods look a short distance ahead, missing the chance to reorder the embedding accesses into a more SSD-friendly manner.

3.2.2 Skewed Access to Embedding Vectors. LSM-Tree mixes hot and cold embedding vectors [27] in the same block, wasting the valuable bandwidth and leading to inefficient cache utilization. This can be mitigated by aggregating hot data into the same block. Many previous works make efforts to improve read efficiency of hot data in the LSM-Tree. For example, HotKey-LSM [44] puts hot keys and cold keys in two separate subtrees. When reading hot keys, it only needs to access a small hot-key LSM-Tree to obtain low latency compared to traversing the hot keys in a large LSM-Tree. SplitDB [11] identifies hot keys and moves them to the upper levels through a larger-scale compaction. Consequently, it can reach hot data faster because the upper levels are at the beginning of the reading path of the LSM-Tree. However, these works focus on keys that are frequently read and gather these keys by performing additional compactions targeting hot keys, while overlooking an important characteristic of the embedding table access pattern. That is, hot data in our LSM-Tree is not only frequently read but also frequently updated. This indicates that the hot data can be gathered when they are inserted to the tree, instead of being scattered among files and gathered afterwards.

3.2.3 Symmetric Read and Write Requests. In recommendation model training, only the embedding vectors accessed during the forward pass are updated in the backward pass [2, 5, 7]. Such a symmetric access pattern gives us an opportunity to identify the location of outdated data and remove it efficiently. Although the updated data is appended to the LSM-Tree and the outdated data is hidden thereby, the embedding lookup gives a hint about the outdated data. Specifically, if a file returns the value of the requested key, this file will contain a outdated value of this key at the end of current iteration, when an updated value is written to a new file of the LSM-Tree. Therefore, as the training progresses, the number of outdated data contained in each file can be easily obtained. Leveraging this opportunity, compactions can target at files containing much outdated data to mitigate its negative impacts.

4 DESIGN

4.1 Overview

RecDB is a storage system designed specifically for recommendation model training. It is built on the LSM-Tree based key-value store, aiming at providing acceptable read and write performance for resource-constrained model training, where embedding tables are offloaded to SSDs. In addition to leverage the inherent advantage of flushing write requests in an SSD-friendly way, RecDB also incorporates novel approaches to enhance the read performance, addressing the traditional shortcoming of the LSM-Tree based key-value store. Notably, here we focus on storing embedding tables in RecDB and other training parameters are stored in memory as they only account for a very small proportion. Basically, we take the index of each embedding vector in the embedding tables as the key, and the vector itself as the value, when storing embeddings in RecDB.

Fig. 7 illustrates an overview of RecDB, which shows the way of how RecDB interacts with the model training. Specifically, it is composed of three key designs: 1) a Request Preprocessor, 2) a Compaction Picker, and 3) a Compaction Scheduler. Additionally, two memory components, the EV Buffer (Embedding Vector Buffer) and the OD Monitor (Outdated Data Monitor), help with the above designs. With these designs, RecDB intends to address the following three shortcomings of the original LSM-Tree: 1) read amplification due to random reads, 2) accumulated outdated data, 3) interference due to the background compactions. Firstly, the Request Preprocessor works as a middleware to intercept read and write requests issued by the model training. In RecDB, the write batch contains the updated embedding vectors at the end of each iteration, while read batches contain multiple lookup requests by looking ahead future iterations. The preprocessor identifies hot keys in these requests, allocates hot keys in a specific way and sorts keys to suit the reading route. In this way, the preprocessor can gather hot data into the same block leveraging the nature of the LSM-Tree and improve read efficiency. The embedding vectors returned by the LSM-Tree for these look-ahead batches are stored in the EV Buffer for future usage. Secondly, the Compaction Picker picks files that contain much outdated data to perform compactions with the help of the OD Monitor. The OD Monitor cleverly identifies outdated data through read requests and records the distribution of outdated data. Finally, the Compaction Scheduler excludes compaction threads from

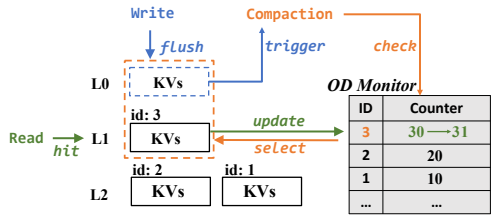


Figure 8: LSM-Tree read flow for recommendation model training.

reading threads and carefully determines the timing of launching compaction tasks to avoid blocking the model training.

4.2 Request Preprocessor

As shown in the left of Fig. 7, the Request Preprocessor comprises following three components. We introduce how each component works.

The Hot Key Identifier tracks embedding access patterns by buffering multiple batches of read requests and counting each embedding's access times. According to the access counts, it identifies top-k frequently accessed embeddings within look-ahead batches as hot embeddings. These newly identified hot embeddings are rewritten into SSDs with special keys during updates.

Notably, the "top-k" configuration is critical to the performance of RecDB. To determine an optimal top-k, we analyze the access patterns of the dataset offline. Specifically, we identify 'hot' keys with an access frequency exceeding 1%. The 1% threshold is chosen because, with a typical batch size (B) greater than 128, such keys are statistically expected to recur within almost every mini-batch (i.e., $B \times frequency > 1.28$). Let p denote the hot-key ratio. We then derive the top-k value using the following formula: top-k = $lookahead\_num \times p$.

The Key Allocator assigns above hot embeddings prefixed keys ("special prefix + original key") before prefetch/update operations. This special prefix is typically the character with the highest or lowest lexicographical order. In LSM-Tree, hot data with special prefix can group together smoothly and be differentiated from cold embedding vectors.

The Key Sorter optimizes SSDs access by reordering read requests based on their key. It avoids repeatedly loading the same block under a resource-constrained environment.

It is notable that these look-ahead embedding vectors are then stored into the EV Buffer. The embeddings are organized by their indices and the EV Buffer records the future reference counts (ref) of each embedding. When a training lookup request is issued, the EV Buffer indexes the key, returns the value, and decreases its ref.

4.3 Compaction Picker

The Compaction Picker enables efficient outdated embedding vector deletion. To achieve this goal, we leverage the symmetric read-write property of embedding vectors. That is, the embedding vector read at the beginning of an iteration will be updated soon after the backward pass. It means that data in the LSM-Tree can be safely removed after being read. Based on such a data access pattern, Fig. 8 gives an example of how the compaction picker works. Firstly, when an embedding e hits in file 3 at Li, we update the outdated data counter of file 3 in the OD Monitor. For example, we increase it from 30 to 31. It means that we can delete at least 31 outdated embedding vectors when we use file 3 as a compaction input. After training, the new embedding vector e' is flushed into $L0$.

Notably, we define a compaction score C_{score} to quantify the deletion efficiency of each file. The C_{score} can be estimated by Eq. (1). In this equation, fid represents the LSM-Tree file stored in SSDs, and $outdated\_num$ is equal to the outdated data counter of fid. T_{merge} refers to the compaction time when file fid is selected as compaction input, while the V_{merge} represents the time required to merge 1 byte of data during compaction. The function $size(k)$ denotes the size of file k, and the $comp_{files}(fid)$ represents the set of files that need to be merged when file fid is chosen. Finally, the file with highest C_{score} is selected for compaction.

$$C_{score}(fid) = \frac{outdated\_num(fid)}{T_{merge}(fid)} \tag{1a}$$

$$T_{merge}(fid) = V_{merge} \times \sum size(k) \quad k \in comp_{files}(fid) \tag{1b}$$

We integrate the outdated data deletion of file 3 into the $L0$ compaction. The overall procedure is depicted in Algorithm 2. The green-shaded region of the algorithm follows the standard RocksDB compaction logic, utilizing the original interfaces *CompactionLevels* and *CompactionFiles* to identify target levels and files. For instance, the $L0$ compaction migrates the embedding vectors from $L0$ to $L1$. Building upon this, the yellow-shaded region implements our specialized deletion mechanism. Specifically, we employ Algorithm 1 to include the target file (file 3) in the compaction inputs, enabling the seamless removal of its outdated embedding vectors e. After the deletion process, the updated file is flushed back to its original level L_i.

Algorithm 1: Delete File Picker

Input : L_{in}, L_{out}, C_{score}, $F_{\text{meta}}[]$
Output: $file\_to\_delete$

```
file_to_delete ← ∅, score ← 0;
if L_in ≠ 0 then
    return file_to_delete
end
for fm ∈ F_meta do
    if fm.level ≥ 2 & score ≤ C_score[fm.ID] then
        score ← C_score[fm.ID], file_to_delete ← fm;
    end
end
return file_to_delete;
```

4.4 Compaction Scheduler

To avoid interference from compaction threads, we prevent concurrency between compaction threads and prefetch threads.

To determine whether a compaction task should be executed, we estimate both the training time supported by the EV Buffer and the latency of the compaction task. For the first one, our Compaction Scheduler interacts with the EV Buffer to obtain the maximum consumption rate of embedding vectors during training (denoted as V_{buf}). Based on V_{buff}, we can estimate the supported training time using Eq.(2a), where $Size(Buff, t)$ represents the number of look-ahead embedding vectors stored in

Algorithm 2: Compaction with delete file picker

Input : C_{score}, $F_{\text{meta}}[]$

$F_{\text{meta}}[]$; // (level and ID) for each SSTable

$L_{\text{in}}, L_{\text{out}} \leftarrow CompactionLevels(F_{\text{meta}}[])$;

$F_{\text{in}}, F_{\text{out}} \leftarrow CompactionFiles(F_{\text{meta}}[])$;

$F_{\text{comp}} \leftarrow merge\_and\_sort(F_{\text{in}}, F_{\text{out}})$;

dump($F_{\text{comp}}, L_{\text{out}}$);

$F_{\text{delete}} \leftarrow DeleteFilePicker(F_{\text{in}}, F_{\text{out}}, C_{\text{score}}, F_{\text{meta}}[])$;

if $L_{in} == 0$ & $F_{delete} \notin (F_{in} \cup F_{comp})$ **then**

  for $key \in Files[F_{delete}.ID]$ **do**

    if $key \in F_{comp}$ **then**

      delete $Files[F_{\text{delete}}.ID][key]$;

    end

  end

  dump($Files[F_{\text{delete}}.ID], F_{\text{delete}}.level$);

end

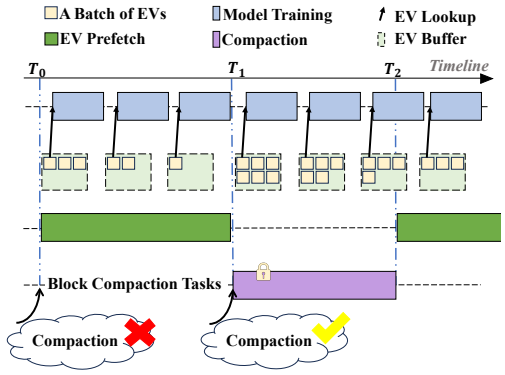


Figure 9: An example of the Compaction Scheduler schedules a compaction task.

the EV Buffer at time t. For the compaction latency at time t, we estimate it in the same way as described in section 4.3, using Eq.(2b). Here, $comp_{files}(t)$ denotes the size of the files selected for compaction at time t. Finally, the compaction task is scheduled at time t only if $T_{train}(t)$ is greater than $T_{merge}(t)$.

$$T_{train}(t) = V_{buff} \times Size(Buff, t) \tag{2a}$$

$$T_{merge}(t) = V_{merge} \times \sum size(k) \quad k \in comp_{files}(t) \tag{2b}$$

Fig. 9 illustrates an example of how the Compaction Scheduler works. At T_0. there are three embedding batches in the EV Buffer and the latency of compaction task is equivalent to the duration of 3 training iterations. As a result, the Compaction Scheduler delays the execution of the compaction task at T_0. Subsequently, the Compaction Scheduler periodically interacts the EV Buffer and attempts to schedule the compaction task again. At T_1, the EV Buffer accumulates 6 look-ahead embedding batches, and its supported training time $T_{training}(T_1)$ exceeds $T_{merge}(T_1)$. Therefore, the compaction task runs at T_1, while temporarily blocking the prefetching of embedding vectors.

Table 2: Model training configurations.

Model	Dataset	Model size
DeepFM [20]	Kaggle	14G
	Terabyte	136G
PNN [34]	Kaggle	22G
	Terabyte	241G
DLRM [32]	Kaggle	40G
	Terabyte	446G

5 EVALUATION

This section first illustrates the effectiveness of RecDB by presenting overall performance in terms of end-to-end training time. Next, we show detailed improvement of RecDB by breaking down the end-to-end training time and show the two main parts, the read time and the update time. We conduct an ablation study to show the impacts of our optimization solutions on RecDB.

5.1 Evaluation Setup

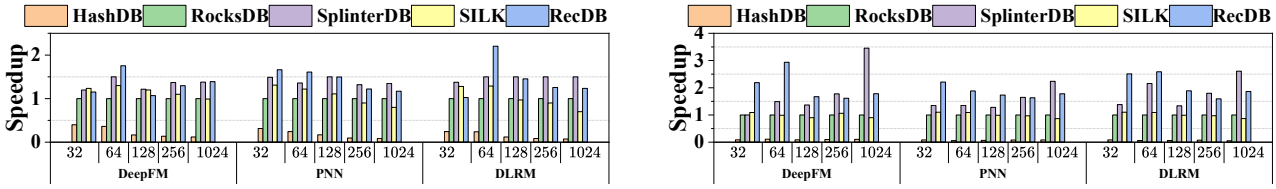
5.1.1 ***Implementation.*** We implemented RecDB in C++ as an extension to RocksDB, providing Python bindings via pybind11. Notably, we utilize a Radix Tree [31] to monitor modified embedding keys. This approach allows for the efficient identification of hot data and the seamless eviction of keys once their access frequency diminishes.

5.1.2 ***Testbed.*** All the following evaluations are conducted on a server that has Intel(R) Xeon(R) Gold 6230N CPU @2.30GHz with 40 cores, 187GB DRAM. We enforced a strict DRAM limit of 16 GB via Linux cgroups The server runs Ubuntu 22.04 and PyTorch 2.6.0. For persistent storage, the server is equipped with an Intel DC P3600 1.6TB NVMe SSD. To simulate a memory-constrained environment, we flush the operating system page cache every 0.02 seconds to avoid too many embedding vectors being buffered by the operating system.

5.1.3 ***Models and training configurations.*** For our evaluations, we configure three widely studied recommendation models, DeepFM [20], PNN [34], and DLRM [32], using embedding dimensions of 9, 18, and 36, respectively, as summarized in Table 2. Additionally, the look-ahead window is set to 512 for all experiments. For the overall performance evaluation, we train these three models using two datasets, the Criteo Kaggle dataset [3] and the Criteo Terabyte dataset [4], leading to different model sizes. We fully demonstrate the effectiveness of RecDB with these diverse models. For further evaluation of the optimization breakdown and the ablation study, we simply present experimental results based on the Criteo Kaggle dataset.

5.1.4 ***Baseline Methods.*** We conduct evaluations comparing to the following four baselines:

- **HashDB**: The storage engine based on hash indexing introduced in AIbox[48]. It contains two layers of hash indexing. The first layer maps indexes of embedding vectors to their corresponding SSD files. The second layer maps indexes to corresponding hash buckets. Each hash bucket is equipped with two memory caches for caching raw embedding vectors, an LRU cache and an LFU cache. When



(a) Kaggle dataset.

(b) Terabyte dataset.

Figure 10: End-to-end training speedup with different batch sizes and models using Kaggle dataset and Terabyte dataset.

the LRU cache or the LRU cache is full, the key-values contained in the cache will be written into SSD files in place according to the first-layer mapping.

- **RocksDB**: The storage engine based on the LSM-Tree, which stores indexes of embedding vectors as keys and serialized vectors as values. It is equipped with an LRU-based block cache, and multiple write buffers.
- **SplinterDB [16]**: The storage engine based on a novel data structure (STBϵ-tree, which combines ideas from log-structured merge trees and Bϵ-trees). It designs a garbage collection to delete inactive data.
- **SILK**: The LSM-Tree based storage engine. It opportunistically allocates I/O bandwidth to compaction operations according to client load.

To simulate a memory-constrained environment and for fair comparison, we allocate an equal 1GB of CPU memory for each method. For HashDB, we configure it 15MB memory for first-layer mapping, 1008MB memory for second-layer mapping and its caches. For LSM-Tree based methods, we configure them 856MB memory for write buffer and 168MB memory for the LRU block cache.

Since RecDB prefetches embedding vectors to overlap the embedding lookup latency with other training operations, we also equip baselines with the same prefetching process and the same size of prefetch buffer for fair comparison.

5.1.5 ***Metrics.*** For the overall comparison, we evaluate the effectiveness of RecDB, HashDB and RocksDB with the following metrics. And all latency metrics are normalized against the RocksDB baseline. 1)**Training Latency:** The average latency of each training. It includes the time for reading current embeddings from the prefetch buffer, model computation, and current embedding updates. 2)**Read Latency:** The average latency for the storage engine to load a batch of embedding vectors, measured from when the storage engine receives a batch of look-ahead requests to when it stores these embedding vectors into the prefetch buffer. 3)**Block Latency:** The average latency during which the prefetch thread blocks the training process, referring to the load time that cannot be overlapped with computation and update time of current iteration. 4)**Update Latency:** The average latency of persisting a batch of updated embedding vectors, starting from the training process finishes updating parameters to the storage engine returns a success signal of the write requests.

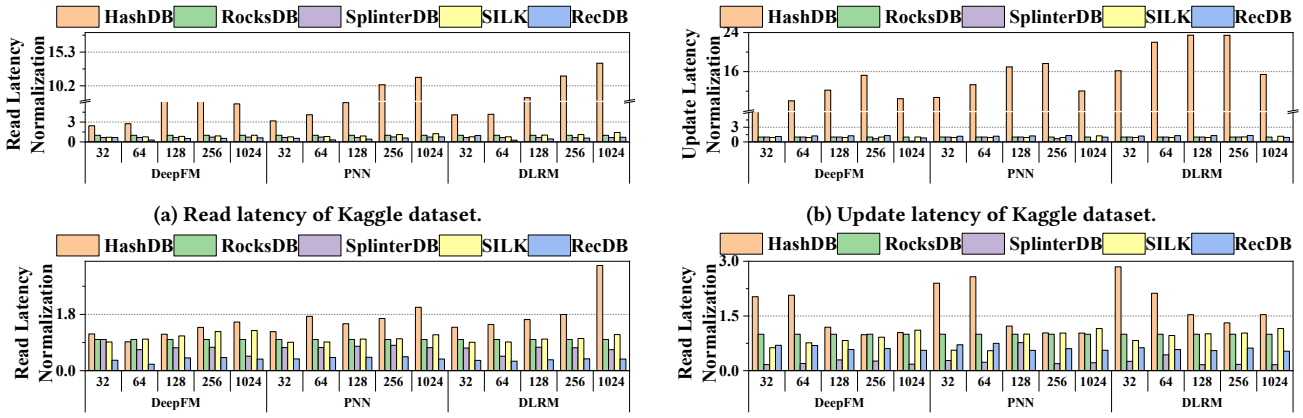
5.2 Overall Performance

This section evaluates the overall performance of HashDB, RocksDB, and RecDB by conducting the end-to-end model training experiments. These experiments train different models across different batch sizes using Kaggle dataset and Terabyte dataset.

To intuitively illustrate the overall acceleration of our approach, Fig. 10a and Fig. 10b present the training speedup of HashDB, SplinterDB, SILK and RecDB, where their latencies are normalized relative to RocksDB latency. For the Kaggle dataset as shown in Fig. 10a, RecDB achieves an average training speedup of 1.70× over RocksDB and 5.54× over HashDB. In other words, RecDB significantly outperforms the naïve LSM-Tree method and hash method on all three models. This aligns with our expectations. Specifically, RecDB achieves 1.33×, 1.43×, and 1.44× training speedup compared to RocksDB when training DeepFM, PNN, and DLRM respectively. It means that RecDB successfully overcomes the inherent bottleneck of the naive LSM-Tree design. However, HashDB has poor performance compared to RecDB. It achieves only 25.10% of RocksDB's performance in terms of training latency on average. The suboptimal training latency of HashDB can be explained by the fact that it faces a severe hash conflict issue in memory-constrained environment, where over 45KB of embedding vectors are mapped to the same physical address. This results in inefficient access patterns, requiring read 45KB of unused data when accessing a cold embedding vector that is not present in either the LRU or LFU caches.

It is worth noting that SplinterDB occasionally outperforms RecDB. This is because SplinterDB utilizes Bϵ-trees to optimize write requests; however, this advantage comes at the cost of higher space amplification compared to RecDB. As for SILK, its performance even falls below to that of RocksDB when peak-load periods are sustained. While SILK attempts to improve bandwidth by deferring compactions during peak loads, it fails to maintain this advantage when it cannot distinguish essential workloads or when the high-load duration exceeds its capacity to manage the resulting compaction backlog.

To test the overall performance in larger models, we further train HashDB, SplinterDB, SILK and RecDB with the Terabyte dataset. Apart from the Kaggle, the Criteo Terabyte is a widely used dataset. When training with Terabyte, the size of models scales to 136G, 241G, and 446G. As depicted in Fig. 10b, RecDB still outperforms RocksDB. Specifically, RecDB achieves average 2.09×, 1.84×, and 2.08× when training DLRM, PNN, and DeepFM respectively. Notably, these speedups with the Terabyte dataset are even higher than those observed with the Kaggle datasets. This is because, in large models, the distribution of hot data becomes more sparse in SSDs under original LSM-Tree design, whereas our RecDB effectively groups hot data. Similar to the Kaggle dataset, HashDB exhibits much poorer training latency in large models when using the Terabyte dataset. Specifically, the training latency of HashDB becomes 13.21× higher compared to RocksDB in the worst case. This is because the hash conflict issue of HashDB becomes more severe in the Terabyte dataset, where more than 443 KB of embedding vectors are mapped to the same SSD file.



(a) Read latency of Kaggle dataset.

(b) Update latency of Kaggle dataset.

(c) Read latency of Terabyte dataset.

(d) Update latency of Terabyte dataset.

Figure 11: Normalized read and update latency across different batch sizes and models.

5.3 Breakdown Study

Table 3: Training time breakdown when training three models with a batch size of 64, shown in percentage. Note that the actual computation time of one model is the same despite of different storage engines.

Model		DeepFM	PNN	DLRM
HashDB	block by read(%)	9.39	16.89	19.19
	computation(%)	10.92	8.53	6.51
	update(%)	**79.69**	**74.57**	**74.30**
RocksDB	block by read(%)	**48.02**	**41.89**	**58.68**
	computation(%)	30.06	35.09	27.21
	update(%)	21.92	23.02	14.11
RecDB	block by read(%)	0.00	0.00	0.00
	computation(%)	**57.83**	**60.38**	**65.85**
	update(%)	42.17	39.62	34.15

To further investigate the factors that contribute to the reduction in training iteration latency, we conduct the following experiments. First, we present the training latency breakdown in end-to-end model training to analyze the different training time components using various storage engines (RecDB, RocksDB, and HashDB). Next, we further focus on the impact of RecDB on two key parts, the read time and the update time.

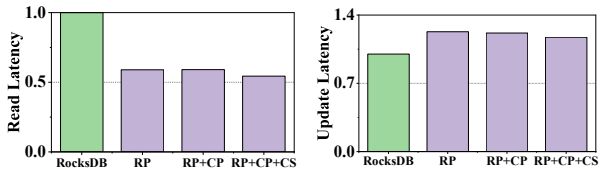
5.3.1 Training time breakdown. To demonstrate the bottlenecks of different storage engines and RecDB's optimization of these bottlenecks, we divide the training time of each iteration into three parts: the block time, the computation time, and the update time. Given that the computation time is unaffected by the storage engine, it remains the same across different storage engines for the same model. Therefore, a higher proportion of computation time indicates better storage engine performance. That is, the model training system spends most of time on computation and requires less time waiting for parameter reading and updates. As shown in Table 3, RecDB spends most of its iteration time on computation, improving training efficiency. When training models using HashDB, its update operation takes up the majority of the training time. It means that the bottleneck of HashDB is its update operations. Although HashDB can achieve relatively fast reads leveraging its LRU and LFU caches which saves much I/O, its write performance is poor due to its update mechanism. Specifically, HashDB performs actual update operations when one of its caches is full. At this time, all the key-values contained in the cache will be written to multiple places of the same file, as these keys share the same first-layer mapping. This results in significant I/O overhead because it requires rewriting almost the entire SSD file. As a result, the training system spends a significant amount of time waiting for parameter updates to complete. When training using RocksDB, the read operation blocks the training process, introducing the significant block time. This indicates that RocksDB suffers from poor read performance. It is because the original LSM-Tree design fails to group frequently accessed embeddings and loads the same data block repeatedly, resulting in high read amplification. In contrast, RecDB improves read performance, allowing its read latency to be fully hidden. Its block time accounts for 0% of the training time. Meanwhile, RecDB maintains a low update latency similar to RocksDB, benefiting from the LSM-Tree's out-place update approach. As a result, its update time accounts for only a small portion of total iteration time.

5.3.2 Read latency and Update latency. We further demonstrate the specific optimization of RecDB in terms of read latency and update latency, as shown in Fig. 11. Note that the measured latencies are normalized to RocksDB.

The Fig. 11c presents the read latency reduction achieved by RecDB over the compared baselines. Overall, RecDB obtains significant improvement in reading performance across all model training. Specifically, the read time of RecDB is reduced by 2.53% ~ 72.96% compared to RocksDB. It is because RecDB achieves lower read amplification and compaction interference than original LSM-Tree design, benefiting from the request preprocessor, the compaction picker and the compaction scheduler. However, HashDB exhibits higher read latencys compared to the LSM-Tree methods, as it requires read the entire SSD file when reading cold embedding vectors. When training with larger model (like DLRM), we observe a substantial increase in the read latency of HashDB. These performance drop is mainly caused by the embedding table's growth. As the embedding table grows, more embeddings are hashed into the same file, increasing hash conflict and read amplification. Finally, HashDB exhibits an average of 6.20× higher read latency than LSM-Tree methods.

Table 4: Optimization configurations for different optimization options

Abbreviation	Description
RP	Request Preprocessor
KS	Key Sorter
KA	Key Allocator
CP	Compaction Picker
CS	Compaction Scheduler



(a) Read latency. (b) Update latency.

Figure 12: Read and update latency of RocksDB, RP, RP+CP, and RP+CP+CS(Normalized to RocksDB).

The Fig. 11d illustrates the reduction in update latency. RecDB and RocksDB deliver lower update latencies compared to HashDB. This is because LSM-Tree methods (RecDB and RocksDB) use an append-only approach to update embedding vector in SSDs, which effectively mitigates write amplification. Notably, the update latency of RecDB is slightly higher than that of RocksDB (17.48% average). This is attributed to the preprocessing overhead introduced by the request preprocessor of RecDB for update requests. However, compared to the significant overall training latency optimization achieved by RecDB (1.71× average), such additional overhead is acceptable. In contrast, HashDB exhibits at least 2.17× update latency compared to LSM-Tree methods. Its poor performance can be explained by the fact that HashDB employs in-place updates, requiring a complete rewrite of the SSD file when updating an embedding vector. As a result, HashDB exhibits higher update latencies, especially when its LFU cache is full and waiting to be written to SSDs.

SILK delivers superior read/write performance over RocksDB under light workloads. For instance, during DeepFM training on the Kaggle dataset with a batch size of 32, SILK reduces read latency by 32.65%. However, as the batch size increases and the workload intensifies, SILK's performance gains diminish, eventually converging to or even falling below the RocksDB baseline. In contrast, SplinterDB is highly optimized for write-intensive scenarios, achieving an average reduction of 21.11% and 73.48% in write latency compared to RocksDB, respectively for Kaggle dataset and Terabyte dataset.

5.4 Ablation Evaluation

To evaluate the effectiveness of each RecDB design, we incrementally incorporate the "request preprocessor" (inlcuding "key sorter" and "key allocator"), "compaction picker" and "compaction scheduler" to RecDB. For the convenience of expression, the abbreviations of each component are shown in Table 4.

5.4.1 Request Preprocessor. As shown in Fig. 12a, the request preprocessor (RP) reduces the read latency by 41.06% compared to RocksDB. It is because the key sorter (KS) and key allocator

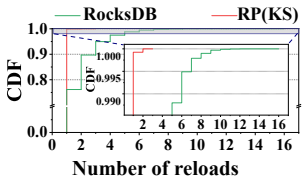


Figure 13: CDF of block reload counts.

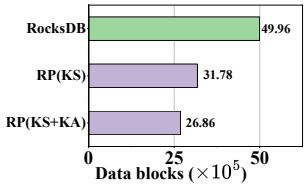


Figure 14: Number of data blocks loaded per batch.

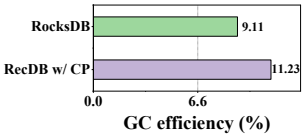


Figure 15: GC efficiency when applying the compaction picker to RecDB, compared to RocksDB.

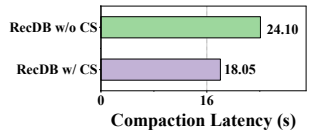


Figure 16: Compaction latency reduction of RecDB when applying the compaction scheduler.

(KA) in request preprocessor successfully reduce the read amplification during prefetch embedding vectors. Fig. 13 and Fig. 14 illustrate the reason behind this efficient reading performance. First, the key sorter avoids repeatedly loading the same SSD block. As depicted in Fig. 13, more than 23.85% data blocks in RocksDB are repeatedly loaded from SSDs to the cache. Some of them have been loaded 16 times. However, the key sorter reorders the access order, significantly reducing the number of reloads. Specifically, more than 99.93% of the data block is loaded only once after applying the key sorter. When applying the key sorter, the number of loaded data blocks decreases to 31.78 $\times 10^5$, representing 36.39% reduction compared to the 49.96$\times 10^5$ in RocksDB. Secondly, the key allocator further reduces the number of data block, through grouping the hot but sparse embedding vector into a small set of data blocks in SSD. In Fig. 14, when applying both the key sorter and the key allocator, the number of loaded data blocks is further reduces by 15.48% compared to simply applying the key sorter. Notably, the request preprocessor introduces an additional update overhead as shown in Fig. 12b. However, compared to the significant optimization achieved by the request preprocessor in read time, such overhead is acceptable.

5.4.2 Compaction Picker. Fig.15 illustrates that the compaction picker enhances the efficiency of outdated data deletion. We use garbage collection (GC) efficiency to measure the ability to delete outdated embedding vectors. GC efficiency is defined as the ratio of the number of deleted outdated vectors to the number of vectors that participate in compactions. Fig. 15 presents the GC efficiency of RocksDB and RecDB equipped with the compaction picker. For the original LSM-Tree compaction design, its GC efficiency is 9. 11%. When RecDB is equipped with the compaction picker, the efficiency of outdated embedding vector deletion improves by 23.30%. This is because the compaction picker helps select the compaction file that removes the most outdated embeddings efficiently.

5.4.3 Compaction Scheduler. The compaction scheduler (CS) avoids overlap between the compaction threads and the prefetch threads, mitigating resource contention. This design leads to reduced latency in prefetch and compaction tasks. As illustrated in Fig. 12a, we observe that when applying the request reprocessor,

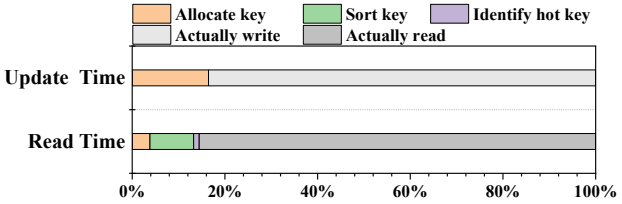


Figure 17: Overhead of the request processor, showing the execution time breakdown of read and update operations in RecDB.

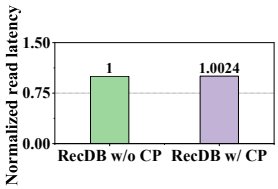


Figure 18: Overhead of the compaction picker, showing the normalized read latency of RecDB.

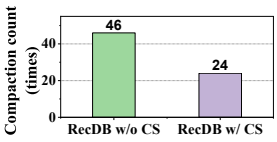


Figure 19: Overhead of the compaction scheduler, showing the frequency of compaction of RecDB.

the compaction picker and the compaction scheduler, the read latency is decreased by 11.53% compared to RecDB equipped with the request reprocessor and the compaction picker. This reduction can be explained by the fact that compaction scheduler allows prefetching to eliminate compaction interference.

Apart from prefetch, compaction itself also benefits from the compaction scheduler. Fig. 16 illustrates the compaction latency comparison between RecDB and RecDB without the compaction scheduler. Specifically, RecDB achieves a 25.10% reduction compared to RecDB without the compaction scheduler. This is because the prefetch thread content less resource with the compaction thread.

The compaction latency reduction further improves the update performance. When compaction finishes early, more write bandwidth can be provided for the flush operation to persist updated parameters from memory to the SSDs. As shown in Fig. 12b, when applying compaction scheduler (e.g. RP+CP+CS), the update latency is decreased by 4.80% compared to those without the compaction scheduler (e.g. RP+CS).

5.4.4 Overhead analysis. We further analyze the overhead introduced by the request preprocessor, the compaction picker, and the compaction scheduler. For the overhead of the request preprocessor, Fig. 17 shows the read and update time breakdown of RecDB. The read execution of RecDB includes allocating hot keys, identifying hot keys, sorting keys, and finally reading the embedding vectors. Similarly, the update execution of RecDB includes allocating hot keys and writing embedding vectors. Overall, the request preprocessor adds little overhead. Specifically, the execution time of key allocation during updates accounts for less than 16.50%. Similarly, the execution time for allocating hot keys, identifying hot keys, and sorting keys account for approximately 3.83%, 9.43%, and 1.19% of total read time, respectively. To evaluate the compaction picker's overhead, we compare the read latency between RecDB without the compaction picker and RecDB with the compaction picker. As shown in Fig. 18, when applying the compaction picker, the embedding read time is increased by 2.50%. It is because picker manages the OD monitor and locks during reading. Specifically, the compaction picker only introduces an average latency of 65.71 ns, which is negligible. For the compaction scheduler's overhead, Fig. 19 compares the number of compaction times between RecDB and RecDB without the compaction scheduler. Compaction accelerates reading by reorganizing the LSM-Tree data. This figure shows that when applying the compaction scheduler, the number of compactions decreases by 47.83%. It can be explained by the fact that the compaction scheduler delays the compaction execution. Although the compaction scheduler employs the lazy compaction policy, it enhances the GC efficiency. Given that the request preprocessor significantly improves the reading performance and overall training performance, the overhead introduced by the compaction scheduler becomes negligible.

6 DISCUSSION

While RecDB is optimized for SSD-based storage to accommodate models that exceed DRAM capacity, we recognize that real-world deployments often leverage tiered storage (combining memory, SSDs, and HDDs) to balance performance and cost. RecDB's LSM-tree architecture is inherently extensible to such heterogeneous environments through the following strategies:

1) **Fine-grained Key-level Caching:** Beyond the current block-granularity cache, RecDB can incorporate a key-granularity caching layer to store the most frequently accessed ("hot") embeddings. Given the power-law distribution typical of recommendation workloads, this design could bypass disk I/O for the majority of requests, significantly reducing tail latency.

2) **Temperature-Aware Data Placement:** RecDB can implement a multi-path storage backend where LSM-tree levels are placed according to data "temperature." Specifically, upper levels (e.g., L_0 to L_2), which handle the bulk of hot read requests, can be pinned to DRAM or NVMe SSDs. Lower levels (L_{max}), containing colder historical data, can be offloaded to high-capacity HDDs to optimize storage costs [8].

3) **Data-Promotion Compaction:** To keep hot key in SSDs, we can integrating data promotion into the compaction process. Specifically, during a compaction from level L_k to L_{k+1}, RecDB can identify hot keys within the input range and "promote" them to L_k instead of moving them to the lower level. This "hotspot-conscious" compaction ensures that frequently accessed embeddings migrate toward faster storage tiers (e.g., from SSD to DRAM) with minimal additional overhead.

7 Conclusion

This paper presents RecDB, an LSM-Tree based storage system designed specifically for training recommendation models in low-resource scenarios. It fully leverages the unique access patterns to address issues of random queries, space overhead caused by obsolete data, and compaction interference during training. Experimental results demonstrate that RecDB successfully eliminates the inherent disadvantages of the naive LSM-Tree design. For instance, it reduces read latency by 72.96% and improves the efficiency of outdated embedding vector deletion by 23.30% compared to RocksDB, a well-known LSM-Tree based storage system.

Acknowledgments

This work is Supported by Guangdong S&T Program under Grant NO. 2024B0101040005, and the National Natural Science Foundation of China (NSFC) under Grant NO. 62272499, 62332021.

8 Artifacts

The source code, datasets, and experimental results for this work are publicly available at: https://github.com/magicalcloud/RecDB_Paper. The repository includes a detailed README.md file with instructions for setting up the environment, installing RecDB, and reproducing the key experiments in Sections 5.

References

[1] [n. d.]. Block Cache. https://github.com/facebook/rocksdb/wiki/Block-Cache
[2] [n. d.]. bojone/keras_lazyoptimizer: Keras implement of Lazy optimizer. https://github.com/bojone/keras_lazyoptimizer
[3] [n. d.]. Display Advertising Challenge. https://kaggle.com/criteo-display-ad-challenge
[4] [n. d.]. Download Criteo 1TB Click Logs dataset. https://ailab.criteo.com/download-criteo-1tb-click-logs-dataset/
[5] [n. d.]. pytorch/torch/optim/_functional.py at main · pytorch/pytorch. https://github.com/pytorch/pytorch/blob/main/torch/optim/_functional.py
[6] [n. d.]. SSD (Solid State Drive) | Samsung Semiconductor Global. https://semiconductor.samsung.com/ssd/
[7] [n. d.]. TensorFlow Addons Optimizers: LazyAdam. https://tensorflow.google.cn/addons/tutorials/optimizers_lazyadam
[8] 2025. facebook/rocksdb. https://github.com/facebook/rocksdb original-date: 2012-11-30T06:16:18Z.
[9] Bilge Acun, Matthew Murphy, Xiaodong Wang, Jade Nie, Carole-Jean Wu, and Kim Hazelwood. 2021. Understanding Training Efficiency of Deep Learning Recommendation Models at Scale. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, Seoul, Korea (South), 802–814. doi:10.1109/HPCA51647.2021.00072
[10] Ehsan K. Ardestani, Changkyu Kim, Seung Jae Lee, Luoshang Pan, Jens Axboe, Valmiki Rampersad, Banit Agrawal, Fuxun Yu, Ansha Yu, Trung Le, Hector Yuen, Dheevatsa Mudigere, Shishir Juluri, Akshat Nanda, Manoj Wodekar, Krishnakumar Nair, Maxim Naumov, Chris Petersen, Mikhail Smelyanskiy, and Vijay Rao. 2022. Supporting Massive DLRM Inference through Software Defined Memory. In *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*. IEEE, Bologna, Italy, 302–312. doi:10.1109/ICDCS54860.2022.00037
[11] Miao Cai, Xuzhen Jiang, Junru Shen, and Baoliu Ye. 2024. SplitDB: Closing the Performance Gap for LSM-Tree-Based Key-Value Stores. *IEEE Trans. Comput.* 73, 1 (Jan. 2024), 206–220. doi:10.1109/TC.2023.3326982
[12] Daniel Rodrigues Carvalho and André Seznec. 2021. Understanding cache compression. *ACM Transactions on Architecture and Code Optimization (TACO)* 18, 3 (2021), 1–27.
[13] Cheng Chen, Yilin Wang, Jun Yang, Yiming Liu, Mian Lu, Zhao Zheng, Bingsheng He, Weng-Fai Wong, Liang You, Penghao Sun, Yuping Zhao, Fenghua Hu, and Andy Rudoff. 2023. OpenEmbedding: A Distributed Parameter Server for Deep Learning Recommendation Models using Persistent Memory. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, Anaheim, CA, USA, 2976–2987. doi:10.1109/ICDE55515.2023.00228
[14] Cheng-Yu Chen, Jui-Nan Yen, You-Ru Lai, Yun-Ping Lin, and Chia-Lin Yang. 2024. RecTS: A Temporal-Aware Memory System Optimization for Training Deep Learning Recommendation Models. In *Proceedings of the 17th ACM International Systems and Storage Conference*. 104–117.
[15] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, and et al. 2016. Wide & Deep Learning for Recommender Systems. doi:10.48550/arXiv.1606.07792 arXiv:1606.07792 [cs].
[16] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. 2020. {SplinterDB}: closing the bandwidth gap for {NVMe}{Key-Value} stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 49–63.
[17] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep Neural Networks for YouTube Recommendations. In *Proceedings of the 10th ACM Conference on Recommender Systems*. ACM, Boston Massachusetts USA, 191–198. doi:10.1145/2959100.2959190
[18] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Stumm. [n. d.]. Optimizing Space Amplification in RocksDB. ([n. d.]).
[19] Carlos A. Gomez-Uribe and Neil Hunt. 2016. The Netflix Recommender System: Algorithms, Business Value, and Innovation. *ACM Transactions on Management Information Systems* 6, 4 (Jan. 2016), 1–19. doi:10.1145/2843948
[20] Huifeng Guo, Ruiming Tang, Yunming Ye, Zhenguo Li, and Xiuqiang He. 2017. DeepFM: a factorization-machine based neural network for CTR prediction. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI'17)*. AAAI Press, Melbourne, Australia, 1725–1731.
[21] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, David Brooks, Bradford Cottel, Kim Hazelwood, Bill Jia, Hsien-Hsin S. Lee, and et al. 2020. The Architectural Implications of Facebook's DNN-based Personalized Recommendation. doi:10.48550/arXiv.1906.03109 arXiv:1906.03109 [cs].
[22] Yongjoo Jang, Sejin Kim, Daehoon Kim, Sungjin Lee, and Jaeha Kung. 2021. Deep Partitioned Training From Near-Storage Computing to DNN Accelerators. *IEEE Computer Architecture Letters* 20, 1 (2021), 70–73. doi:10.1109/LCA.2021.3081752
[23] Biye Jiang, Chao Deng, Huimin Yi, Zelin Hu, Guorui Zhou, Yang Zheng, Sui Huang, Xinyang Guo, Dongyue Wang, Yue Song, Liqin Zhao, Zhi Wang, Peng Sun, Yu Zhang, Di Zhang, Jinhui Li, Jian Xu, Xiaoqiang Zhu, and Kun Gai. 2019. XDL: an industrial deep learning framework for high-dimensional sparse data. In *Proceedings of the 1st International Workshop on Deep Learning Practice for High-Dimensional Sparse Data*. ACM, Anchorage Alaska, 1–9. doi:10.1145/3326937.3341255
[24] Hiwot Tadese Kassa, Paul Johnson, Jason Akers, Mrinmoy Ghosh, Andrew Tulloch, Dheevatsa Mudigere, Jongsoo Park, Xing Liu, Ronald Dreslinski, and Ehsan K Ardestani. 2023. MTrainS: Improving DLRM training efficiency using heterogeneous memories. *arXiv preprint arXiv:2305.01515* (2023).
[25] Daniar H. Kurniawan, Ruipu Wang, Kahfi S. Zulkifli, Fandi A. Wiranata, John Bent, Ymir Vigfusson, and Haryadi S. Gunawi. 2023. EVStore: Storage and Caching Capabilities for Scaling Embedding Tables in Deep Recommendation Systems. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 281–294. doi:10.1145/3575693.3575718
[26] Shiyu Li, Yitu Wang, Edward Hanson, Andrew Chang, Yang Seok Ki, Hai Li, and Yiran Chen. 2024. NDRec: A Near-Data Processing System for Training Large-Scale Recommendation Models. *IEEE Trans. Comput.* 73, 5 (May 2024), 1248–1261. doi:10.1109/TC.2024.3365939
[27] Haifeng Liu. 2023. Accelerating Personalized Recommendation with Cross-level Near-Memory Processing. (2023).
[28] Haifeng Liu, Long Zheng, Yu Huang, Jingyi Zhou, Chaoqiang Liu, Runze Wang, Xiaofei Liao, Hai Jin, and Jingling Xue. 2024. Enabling Efficient Large Recommendation Model Training with Near CXL Memory Processing. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, Buenos Aires, Argentina, 382–395. doi:10.1109/ISCA59077.2024.00036
[29] Zhuoran Liu, Leqi Zou, Xuan Zou, Caihua Wang, Biao Zhang, Da Tang, Bolin Zhu, Yijie Zhu, Peng Wu, Ke Wang, and Youlong Cheng. 2022. Monolith: Real Time Recommendation System With Collisionless Embedding Table. doi:10.48550/arXiv.2209.07663 arXiv:2209.07663 [cs].
[30] Nimrod Megiddo and Dharmendra S. Modha. 2003. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *2nd USENIX Conference on File and Storage Technologies (FAST 03)*. USENIX Association, San Francisco, CA. https://www.usenix.org/conference/fast-03/arc-self-tuning-low-overhead-replacement-cache
[31] Donald R Morrison. 1968. PATRICIA—practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM (JACM)* 15, 4 (1968), 514–534.
[32] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, and et al. 2019. Deep Learning Recommendation Model for Personalization and Recommendation Systems. http://arxiv.org/abs/1906.00091 arXiv:1906.00091 [cs].
[33] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (June 1996), 351–385. doi:10.1007/s002360050048
[34] Yanru Qu, Han Cai, Kan Ren, Weinan Zhang, Yong Yu, Ying Wen, and Jun Wang. 2016. Product-based Neural Networks for User Response Prediction. doi:10.48550/arXiv.1611.00144 arXiv:1611.00144 [cs].
[35] Mohammad R Rezaei. [n. d.]. Amazon Product Recommender System. ([n. d.]).
[36] Haidong Rong, Yangzihao Wang, Feihu Zhou, Junjie Zhai, Haiyang Wu, Rui Lan, Fan Li, Han Zhang, Yuekui Yang, Zhenyu Guo, and Di Wang. 2020. Distributed Equivalent Substitution Training for Large-Scale Recommender Systems. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, Virtual Event China, 911–920. doi:10.1145/3397271.3401113
[37] Zhi Shen, Bo Jiang, Xinbing Wang, and Chenghu Zhou. 2022. ARC-learning: A self-tuning cache policy under dynamic popularity. In *2022 IEEE 8th International Conference on Computer and Communications (ICCC)*. IEEE, 610–615.
[38] Xuan Sun, Hu Wan, Qiao Li, Chia-Lin Yang, Tei-Wei Kuo, and Chun Jason Xue. 2022. RM-SSD: In-Storage Computing for Large-Scale Recommendation Inference. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, Seoul, Korea, Republic of, 1056–1070. doi:10.1109/HPCA53966.2022.00081
[39] Shang-Hung Ti, Tseng-Yi Chen, Tsung Tai Yeh, Shuo-Han Chen, and Yu-Pei Liang. 2024. OC-DLRM: Minimizing the I/O Traffic of DLRM Between Main Memory and OCSSD. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, Valencia, Spain, 1–2. doi:10.23919/DATE58400.2024.10546725
[40] Hu Wan, Xuan Sun, Yufei Cui, Chia-Lin Yang, Tei-Wei Kuo, and Chun Jason Xue. 2021. FlashEmbedding: storing embedding tables in SSD for large-scale

recommender systems. In *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems*. ACM, Hong Kong China, 9–16. doi:10.1145/3476886.3477511

[41] Jizhe Wang, Pipei Huang, Huan Zhao, Zhibo Zhang, Binqiang Zhao, and Dik Lun Lee. 2018. Billion-scale Commodity Embedding for E-commerce Recommendation in Alibaba. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM, London United Kingdom, 839–848. doi:10.1145/3219819.3219869

[42] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. 2014. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD. In *Proceedings of the Ninth European Conference on Computer Systems*. ACM, Amsterdam The Netherlands, 1–14. doi:10.1145/2592798.2592804

[43] Yichao Wang, Huifeng Guo, Ruiming Tang, Zhirong Liu, and Xiuqiang He. 2020. A Practical Incremental Method to Train Deep CTR Models. http://arxiv.org/abs/2009.02147 arXiv:2009.02147 [cs].

[44] Yi Wang, Peiquan Jin, and Shouhong Wan. 2020. HotKey-LSM: A Hotness-Aware LSM-Tree for Big Data Storage. In *2020 IEEE International Conference on Big Data (Big Data)*. IEEE, Atlanta, GA, USA, 5849–5851. doi:10.1109/BigData50022.2020.9377736

[45] Yitu Wang, Shiyu Li, Qilin Zheng, Andrew Chang, Hai Li, and Yiran Chen. 2023. EMS-i : An Efficient Memory System Design with Specialized Caching Mechanism for Recommendation Inference. *ACM Transactions on Embedded Computing Systems* 22, 5s (Oct. 2023), 1–22. doi:10.1145/3609384

[46] Yingcan Wei, Matthias Langer, Fan Yu, Minseok Lee, Jie Liu, Ji Shi, and Zehuan Wang. 2022. A GPU-specialized Inference Parameter Server for Large-Scale Deep Recommendation Models. In *Proceedings of the 16th ACM Conference on Recommender Systems*. ACM, Seattle WA USA, 408–419. doi:10.1145/3523227.3546765

[47] 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*. ACM, Virtual USA, 717–729. doi:10.1145/3445814.3446763

[48] Weijie Zhao, Jingyuan Zhang, Deping Xie, Yulei Qian, Ronglai Jia, and Ping Li. 2019. AIBox: CTR Prediction Model Training on a Single Node. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*. ACM, Beijing China, 319–328. doi:10.1145/3357384.3358045