

VineCache: Adaptive and Group-Aware Embedding Caching for Deep Recommendation Inference

Menglei Chen, Yu Hua*, Ming Zhang, Zhijun Yang

Huazhong University of Science and Technology

*Corresponding Author: Yu Hua (csyhua@hust.edu.cn)

Abstract—Modern deep recommendation inference systems heavily rely on key-value databases to manage massive embedding vectors (EVs). To accelerate EV lookups during inference, these databases typically leverage caching mechanisms to keep frequently accessed EVs in fast-access memory tiers. However, traditional caching schemes in existing databases operate at the object granularity, which leads to a fundamental granularity mismatch with the group-level access patterns of recommendation inference workloads. This mismatch decouples object hit rates from actual query success rates, hence leading to low cache efficiency. To bridge this granularity gap, we propose VineCache, an adaptive and group-aware embedding caching scheme that efficiently aligns cache management granularity with computation dependencies. By introducing a dynamic experience point-based scoring mechanism, VineCache efficiently evaluates EVs based on their contributions to group hit rather than individual frequency. VineCache leverages a hit count-based scoring mechanism and a group-aware priority weighting design to dynamically capture EV lookup patterns. VineCache further employs multi-level eviction and aging mechanisms to adapt to varying access patterns. Moreover, VineCache employs a reinforcement learning scheme to automatically optimize parameters. Experiments with real-world deep recommendation systems show that VineCache outperforms state-of-the-art caching schemes, reducing end-to-end inference latency by 14% on average.

Index Terms—Deep recommendation systems, embedding caching, caching schemes

I. INTRODUCTION

Recommendation systems are widely used in various applications to provide customized recommendation services, such as recommended products [1], [2], personalized advertisements [3], [4], and interesting news [5], [6]. The main driving force of the recommendation system is its ability to capture user behaviors and preferences and further extract essentials from massive information sources. With advanced hardwares (e.g., GPUs), the recommendation systems widely leverage *deep learning recommendation model* (DLRM) [7], [8] to significantly improve the accuracy, called *deep recommendation system* [9], [10], [11], [12], [13], [14]. These deep recommendation systems have significant impacts on user experiences and behaviors. For example, recent studies [15], [16], [4] report that deep recommendation systems account for 35% of Amazon traffic, 60% of YouTube views, and 75% of Netflix movies viewed. To meet the demands of high quality of service (QoS) requirements, deep recommendation systems are always in pursuit of high performance (e.g., low inference latency) [17], [18], [19]. Given the same QoS requirements, a more efficient recommendation system can evaluate more

items and provide more accurate recommendations, which is hence more likely to satisfy user interests [18].

Modern deep recommendation systems typically consist of two components: an embedding layer for generating low-dimensional representations of input features and a GPU-accelerated multi-layer perceptron (MLP) for scoring items. Input features are divided into sparse features (e.g., user ID) and dense features (e.g., user age). Sparse features are typically represented as high-dimensional one-hot vectors, resulting in massive MLP parameters. To reduce complexity, sparse features are encoded into embedding vectors (EVs) [20]. These EVs are trained to become precise and stored in EV tables after training. During inference, the recommendation system retrieves the required EVs from the EV tables using sparse features as keys, a process called *EV lookup*.

With over a decade of growth in users and item numbers of recommendation systems, storing EVs has become intractable due to their high space requirements. For example, an EV table with a billion 512-dimensional FP32 EVs requires 2 terabytes of space [21], [22], which cannot be fully stored in the limited-size memory of GPU. Furthermore, to increase the recommendation accuracy and support richer feature representations, modern recommendation systems store more EVs with higher dimensions [23], [24]. Recent studies show that EV tables will scale to hundreds of terabytes [25], [26], exceeding even CPU memory capacity. As a result, EV tables are stored in large-capacity but slower storage devices (e.g., SSDs) [27], [18]. However, accessing the low-speed storage devices causes high latency [28]. Reports from Alibaba [29], [30] and Meta [31], [32] show that embedding layers account for over 60% of recommendation inference latency in production.

To accelerate EV lookup and reduce latency, prior systems [27], [18], [33], [34] adopt embedding caches to store frequently accessed EVs in faster tiers (e.g., DRAM). During inference, the system first checks the embedding cache. If the EV is found, the recommendation system obtains the EV without accessing the slow backend storage. However, unlike traditional caching workloads, EV lookup workloads represent a unique class of multi-key point queries with an implicit *all-or-nothing* property. Specifically, each inference request atomically requires a *group* of EVs. The latency is determined by the slowest access in this group, enforcing a strict group-level granularity for performance optimization. As a result, even only a *single* EV miss will deteriorate the end-to-end inference latency. On the other hand, inference

achieves the lowest latency when *all* EVs hit the cache, called a *group hit*. Since the caching tier is orders of magnitude faster than the backend storage devices, a group hit is much faster than encountering a miss (e.g., $8.66\times$ as shown in §II-C). Therefore, achieving high group hit rates (the portion of group-hit EV lookup in all EV lookups) is critical for providing high-performance inference.

Unfortunately, existing caching designs fail to achieve high group hit rates due to the granularity mismatch between caching schemes and EV lookup. Traditional caching schemes (e.g., LRU [35] and LFU [36]) operate at the object granularity, maximizing the probability of individual EV hits. However, in deep recommendation inference, the effective unit of performance is the request group. A high object hit rate does not guarantee a high group hit rate if the few missing EVs are scattered across many different requests. While recent designs attempt to improve group hit rates by leveraging group-level metrics (e.g., max hit count [27]), they face challenges with the dynamic and evolving nature of real-world workloads. In real-world scenarios, access patterns are highly dynamic, shifting due to changing user behaviors and the continuous introduction of new content [37]. Over time, the hotness of specific EVs may decline, but such EVs become hard to be evicted due to static or rigid prioritization mechanisms. This causes an inability to effectively adapt to changing access patterns, resulting in suboptimal group hit rates and degraded system performance.

Therefore, we argue that an effective embedding caching scheme needs to (1) be fundamentally aware of the all-or-nothing property inherent to group-based EV lookups, and (2) possess the ability to dynamically adapt to varying access patterns. To achieve these design goals, we propose VineCache, an adaptive and group-aware embedding caching scheme that efficiently captures EV lookup patterns across diverse access patterns. In this paper, the *EV lookup pattern* indicates the implicit EV correlations under the group-based EV accesses, while the *access pattern* indicates the hotness distribution among EVs. The core innovation of VineCache lies in a dynamic *experience point* (EXP) scoring system that quantifies and ranks EV importance. Each EV is categorized into a specific *level*, determined by its accumulated EXP. VineCache prioritizes caching EVs of higher levels. The EXP grows when an EV is involved in an inference. Upon reaching a level-specific exponential threshold (e.g., 2^i for level i), the EV ascends to a higher level, called “*Level UP*”.

To efficiently capture group access patterns and ensure adaptability, VineCache proposes several synergistic key designs: (1) a ***hit count-based EXP scoring*** that directly associates the importance of an EV with its participation in group-based lookup, (2) a ***group-aware priority weighting*** mechanism that adaptively weights the base EXPs to encourage promising new EVs while preventing stale high-level EVs from persisting excessively, (3) a ***multi-level eviction scheme incorporated with an aging mechanism***, which avoids premature eviction of low-level EVs while enabling quick demotion of less relevant high-level ones, and (4)

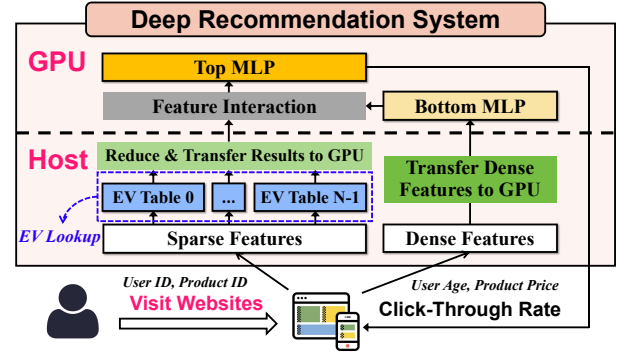


Fig. 1. The overview of a deep recommendation system.

a ***reinforcement learning-based parameter optimizer*** that automatically adjusts key parameters in response to workload changes, ensuring robust performance at runtime.

In order to demonstrate the end-to-end performance in real-world applications, we integrate VineCache into a popular open-sourced framework from Meta (i.e., DLRM [38], [7]). We evaluate system performance using representative large-scale datasets [39], [40], [41]. Compared with state-of-the-art caching schemes, including LRU [35], LFU [36], ARC [42], S3-FIFO [43], Cacheus [44], and EV-LFU [27], VineCache provides $1.75\times$ higher group hit rates and $1.14\times$ lower end-to-end inference latency on average.

In summary, we make the following contributions:

- We formally identify and analyze the granularity mismatch between traditional object-level caching schemes and the group-based nature of embedding lookups as the primary bottleneck for inference latency. We demonstrate that optimizing for object hits is insufficient for group-centric workloads.
- We propose VineCache, an adaptive and group-aware embedding caching scheme that improves group hit rates by leveraging a dynamic EXP-based scoring system.
- We integrate VineCache into a real-world recommendation system and conduct experiments using representative datasets. Compared with state-of-the-art caching schemes, VineCache significantly improves the cache efficiency and inference latency.

The source code of VineCache is publicly available at <https://github.com/LighT-chenml/VineCache>.

II. BACKGROUND AND MOTIVATION

A. Deep Recommendation Systems

Figure 1 illustrates the workflow of a deep recommendation system for product recommendation. When a user visits a website, it collects and transmits user and product features (sparse and dense) to the system. Sparse features, being high-dimensional, are densified into embedding vectors (EVs) [20], initialized as random values and refined through training. The system stores trained EVs in EV tables for different sparse features, and retrieves them during inference through *EV lookups*. Dense features are encoded by a bottom MLP into vectors with the same dimension as the EVs of sparse features. These vectors are combined (e.g., by dot

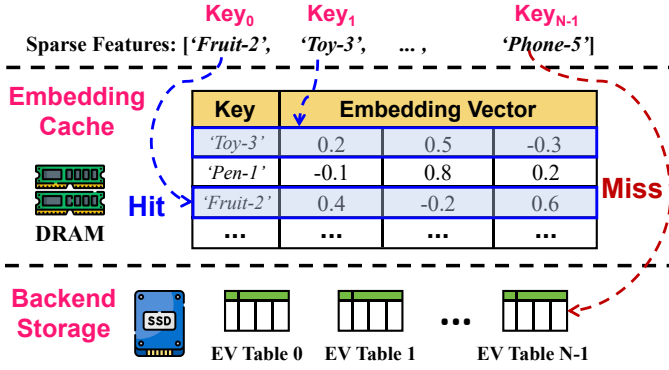


Fig. 2. An example of embedding vector lookup.

products and concatenation) and passed to a top MLP, which outputs product scores, such as click-through rates (CTR). Products with higher scores are then recommended to users.

Although different deep recommendation systems have various ad-hoc designs [9], [45], [10], [46], [12], [47], they follow the *embedding-MLP* paradigm [48], [49], where the embedding layer captures low-dimensional representations of input features, and the top MLP computes the scores of the items. While recent designs [50], [51], [52] explore leveraging large language models for recommendations, the embedding-MLP DLRM architecture remains dominant in most scenarios due to its cost-efficiency.

B. Embedding Vector Lookup

Figure 2 illustrates an example of EV lookup, where the input is a *group* of sparse features. For each sparse feature, the recommendation system performs an EV table lookup using the feature as a key (e.g., a string like “Toy-3” or a hash value). In production systems, embedding tables are typically managed by key-value databases (e.g., RocksDB [53]). EV lookup is fundamentally a memory-bound multi-key point query [29], [30]. This is because modern recommendation systems maintain billions of EVs to represent users and items, consuming extensive storage space. For instance, an EV table with a billion EVs of 512 dimensions in FP32 format requires nearly 2 terabytes [21], [22]. Moreover, to increase the recommendation accuracy and support richer feature representations, modern recommendation systems in industry tend to store more EVs with higher dimensions [23], [24], resulting in larger EV tables that are expected to exceed hundreds of terabytes [25], [26]. Consequently, EV tables are stored in large-capacity but low-speed storage devices (e.g., SSD), and EV lookups must retrieve data from these backends.

EV lookup is critical to recommendation inference latency. Alibaba [29], [30] and Meta [31], [32] report that the embedding layers account for over 60% of inference latency in production. Studies show that the primary bottleneck is the time-consuming EV lookup [27], [18], [33], [34]. To accelerate EV lookups, prior designs [27], [18], [33] leverage embedding caches to store frequently accessed EVs in faster access tiers (e.g., DRAM). Upon an EV table lookup, the recommendation system first queries the embedding cache. On a cache *hit*, the EV is retrieved directly from the cache, without

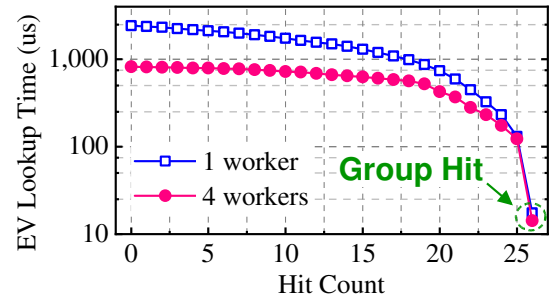


Fig. 3. The EV lookup time with various hit counts.

accessing the slow backend storage. On a cache *miss*, the EV is fetched from backend storage and inserted into the cache.

C. Challenges of Embedding Caches

Unlike traditional caches, the embedding cache exhibits an implicit *all-or-nothing* property. During inference, EVs retrieved from all EV table lookups are required for the feature interaction. As a result, a *single* embedding cache miss can even deteriorate end-to-end inference latency. Conversely, complete cache hits across *all* EV table lookups, termed a *group hit*, result in the lowest inference latency. Figure 3 presents the average EV lookup time with various hit counts (the number of hits in an EV lookup). The results show that a group hit is 7.49 $\times$ and 8.66 $\times$ faster than encountering a miss when respectively performing the EV lookup with 1 and 4 workers (experimental setup is detailed in §IV), highlighting the significance of group hit across different embedding lookup configurations. Group hits improve both average and tail latency, enabling recommendation systems to meet stringent quality of service (QoS) requirements. Unfortunately, it is nontrivial for existing caching schemes to achieve group hits due to the following challenges:

Challenge 1: Granularity mismatches between caching schemes and EV lookup. The recommendation systems perform EV lookups at the EV group level, while traditional caching schemes focus on maximizing object-level hit rates without considering the unique challenges of group-level access patterns. We evaluate multiple representative caching schemes using the Criteo-Kaggle CTR dataset [39]. As shown in Figure 4, existing caching schemes enable high EV hit rates (the portion of hit EVs in all involved EVs), ranging from 50% to 90%. However, these schemes fail to deliver comparable group hit rates (the portion of group-hit lookup in all EV lookups), which only range from 1% to 42%. The gap between the EV and the group hit rates is large, especially when the cache size (the portion of cached EVs in all EVs) is small ($<1\%$). Such a hit rate gap stems from the granularity mismatches between caching schemes and EV lookup. Mathematically, let P_{obj} be the object hit rate and P_{group} be the group hit rate. Under independent assumptions, $P_{group} \approx P_{obj}^N$ (where N is group size). However, real-world access is correlated. The granularity mismatch arises because traditional caches maximize P_{obj} without considering the correlation structure required to maximize P_{group} . They treat EVs as independent units rather

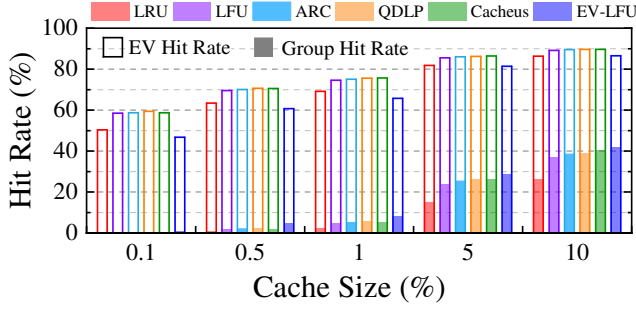


Fig. 4. The gaps between EV and group hit rates.

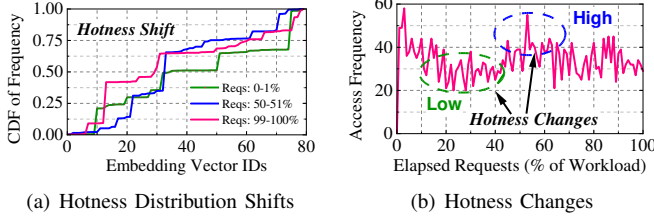


Fig. 5. The hotness distributions and changes of EVs. (We randomly selected a portion of EVs for a clear display)

than members of a critical group.

Challenge 2: Limited adaptability to dynamic access patterns.

The access patterns of recommendation workloads frequently change due to varying user behaviors throughout the day. Analyzing the day-0 part of the Criteo-Kaggle dataset [39], we observe distinct access patterns across requests from different workload segments (e.g., 0-1%, 50-51%, 99-100%), as shown in Figure 5(a). For instance, the first 20 EVs account for 42% of total accesses in the 99-100% segment but only 14% of total EV accesses in the 50-51% segment. Furthermore, the hotness of individual EVs changes significantly over time, as depicted in Figure 5(b). Dynamic access patterns make it challenging for caching schemes to adapt quickly. For example, EV-LFU [27], an LFU variant tailored for embedding caches, ranks EV importance by max-hit-based scoring and employs a flushing mechanism to remove irrelevant EVs. This mechanism struggles with drastic access pattern shifts. In dynamic scenarios, older EVs with high max-hits may remain in the cache unnecessarily, while newer but potentially valuable EVs may be evicted prematurely. This hinders the embedding cache to achieve high group hit rates across varying access patterns.

III. VINECACHE DESIGN

To enable low-latency embedding lookups, we propose VineCache, an adaptive and group-aware caching scheme that captures EV lookup patterns and adapts to diverse access patterns. To bridge the granularity mismatch, VineCache shifts the optimization target from individual objects to request groups. The key idea is to evaluate EVs with a dynamic group hit-aware scoring system. In this section, we first present the high-level overview of VineCache and then dive into the details of each component.

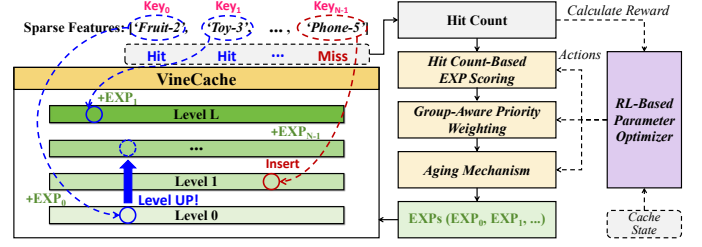


Fig. 6. The overview of VineCache.

A. Overview

Figure 6 shows the overview of VineCache. From a database perspective, VineCache operates as an advanced tiered buffer pool. In VineCache, each cached EV is assigned a *level*. EVs of higher levels are prioritized to be cached. To determine EV levels, we introduce the concept of *experience point* (EXP). EXPs are initialized to 0 and increase when the corresponding EVs participate in an inference. If an EV is evicted from VineCache, its EXP resets to 0. Each level has an associated EXP threshold. When the EXP of a cached EV reaches the threshold of its current level, it undergoes a “Level UP”, advancing to the next level.

After an EV lookup, the involved EVs can gain EXPs calculated by EXP scoring. For the involved EVs that are not cached, VineCache fetches them from backend storage and caches them. The newly cached EVs can also gain EXPs from the EV lookup. When VineCache is full (i.e., the total size of cached EVs exceeds the capacity of VineCache), it evicts EVs with the lowest level. For EVs with the same level, VineCache evicts EVs using existing caching algorithms (e.g., LRU [35]). In our implementation, EVs with the same level are managed by a *sub-cache* that employs a specific caching algorithm (e.g., LRU cache). Each sub-cache can accommodate arbitrary numbers of EVs. Initially, all cached EVs are at level 0. When the level of a newly inserted EV does not exist, VineCache adds the corresponding level. It is easy for VineCache to add a level by simply adding a sub-cache for the level.

The architectural decision to employ discrete levels provides two fundamental benefits. First, it amortizes management overhead by replacing global priority sorting with localized sub-cache operations (e.g., LRU or MFU), triggering structural updates only when exponential thresholds are crossed. Second, the discrete tiers act as a low-pass filter against transient access noise, ensuring that EV promotions reflect sustained relevance rather than momentary query spikes.

Compared with existing caching schemes, VineCache exhibits the following salient features:

- **Dynamic group hit-aware scoring.** VineCache is aware of the all-or-nothing property of embedding cache. VineCache calculates the gained EXPs via hit count-based EXP scoring (§III-B). To ensure high group rates across various access patterns, VineCache leverages the group-aware priority weighting (§III-C) to adjust the gained EXPs of the involved EVs, preventing EVs from accumulating extremely high EXPs.
- **Adaptive designs.** VineCache adopts multi-level eviction and EXP aging mechanisms to dynamically adapt to different

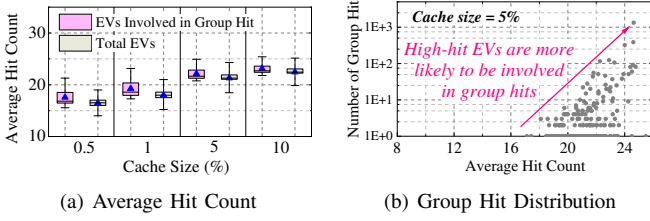


Fig. 7. The average hit counts of EVs and the group hit distribution (under the LRU caching scheme).

access patterns. When evicting EVs, VineCache considers multiple levels and selects an EV to evict among these levels, which avoids premature evictions for low-level EVs and promotes quick demotions for high-level EVs (§III-D). Moreover, VineCache dynamically increases the gained EXPs to capture the recency of EVs (§III-E). Furthermore, VineCache adopts a reinforcement learning-based parameter optimizer to fine-tune its parameters (§III-F).

• **Flexible incorporation.** VineCache can easily cooperate with existing caching algorithms. To demonstrate the flexibility and generality of VineCache, we implemented VineCache variants over several representative caching algorithms (§III-G).

B. Hit Count-Based EXP Scoring

To achieve a high group hit rate, EXP scoring in VineCache needs to capture EV lookup patterns, which is however non-trivial. While an elaborate EXP scoring mechanism (e.g., using integer linear programming) can achieve high group hit rates, it causes severe time consumption, which is not cost-efficient for the time-sensitive embedding cache. An intuitive EXP scoring can achieve low overheads by increasing the EXPs of EVs when these EVs are involved in a group hit. However, such an intuitive scoring mechanism cannot identify important EVs when group hits are not many, especially with small cache sizes, thus leading to low group hit rates. Overall, an efficient EXP scoring mechanism needs to achieve the sweet point between high group hit rates and low runtime overheads.

We observe that high-hit EVs are crucial for facilitating group hits. Figure 7(a) shows the average hit counts (i.e., the number of hits in an EV lookup) of EVs that are involved in group hits under the LRU caching scheme. The results demonstrate that the EVs involved in group hits typically have high average hit counts. Figure 7(b) presents the numbers of group hits and average hit counts of EVs (each point indicates an EV). The results show that the EVs with high hit counts are more likely to be involved in group hits. Based on these observations, in VineCache, we adopt a finer-grained EXP scoring mechanism, which uses the hit count as a heuristic to calculate the EXP. Such a hit count-based scoring mechanism offers two advantages: (1) High efficiency. A high hit count indicates a high probability for an EV to be involved in group hits, even if the EV has not been involved in a group hit. (2) Low computation cost. We can easily obtain the hit count by summing up the results of cache lookups. By prioritizing EVs with high hit counts, VineCache implicitly aligns the object-level eviction decision with the group-level computational dependencies, effectively mitigating the granularity mismatch

without expensive group-state tracking.

To minimize computational overhead, VineCache calculates the gained EXP using two simple mathematical functions: a piecewise linear function and an exponential function. Both functions take the hit count as input, ensuring that EVs with higher hit counts gaining greater EXPs, thereby reflecting their relative importance. The piecewise linear function increases the gained EXP linearly with hit count, using steeper slopes for higher hit-count intervals to amplify the reward for frequently accessed EVs. In contrast, the exponential function grows the gained EXP at an exponential rate as hit count rises, more aggressively distinguishing high-hit EVs. While the two functions differ in growth pattern, they share the same design principle: to significantly widen the EXP gap as hit counts increase, thereby prioritizing EVs that are critical for group hits. By facilitating EVs involved in the same inference, both EXP function variants enhance VineCache’s ability to improve the group hit rate, while maintaining low computational cost.

To differentiate the priorities of levels, we adopt an exponential EXP threshold for each level (i.e., 2^i for level i). This EXP threshold ensures that higher-level embedding vectors (EVs) require exponentially more experience points to advance, which prevents hasty promotions caused by temporary access spikes and preserves stable importance ranking. Moreover, this design aligns naturally with the hit count-based EXP accumulation, where higher hit counts lead to greater EXP gains, hence enabling the cache to respond to both short-term popularity and long-term value. Furthermore, the exponential threshold form is simple to calculate and easy to scale, adding negligible computational overhead while supporting adaptive caching under dynamic access patterns.

C. Group-Aware Priority Weighting

While hit count-based EXP scoring enables VineCache to capture the EV lookup patterns, the old EVs possibly accumulate too-high EXPs after many EV lookups, which are hard to be evicted. Therefore, in order to provide high group hit rates across diverse access patterns, it is important to further make the EXP scoring adaptive. Intuitively, we can simply decrease the EXPs of EVs when the EVs are becoming cold. However, it is hard to efficiently identify cold EVs and demote them, while scanning all cached EVs for identifying cold EVs incurs high overheads. Instead of decreasing the too-high EXPs for cold EVs, our key idea is to prevent EVs from accumulating too-high EXPs.

To prevent high-level EVs from becoming excessively dominant and hindering cache adaptivity, we propose the *group-aware priority weighting* (GAPW for short) to enhance the adaptivity of the EXP scoring. Specifically, GAPW encourages low-level EVs to gain high EXPs while limiting the gained EXPs of high-level EVs. GAPWs act as weights for calculating weighted EXPs. After computing the base EXPs via the EXP function, we multiply them by the GAPWs to obtain the final gained EXPs. The GAPW calculation follows a coherent design philosophy: *differentiated amplification to encourage promotion, followed by controlled compression for stability*.

① Input of GAPW			② Difference of levels		③ Non-linear Normalization
Key	EXP	Level	SumDist	LogSum	GAPW
'Fruit-2'	13.6	4	56	4.03	0.77
'Toy-3'	3.2	2	224	5.41	0.83
'Phone-5'	163.5	8	-280	-5.63	0.16

Fig. 8. An example of calculating GAPW (using a group size of 3).

This two-stage approach ensures that EVs within a group are appropriately promoted while preventing excessive volatility in the caching system. Figure 8 illustrates the GAPW calculation. **Stage 1: Cubic amplification for emphasizing relative level disparity.** Aligned with the level-based priority granularity in VineCache, we use the levels of EVs as the input of the GAPW calculation. After obtaining the levels of all EVs in the group, we calculate $LevelDist$ to quantitatively represent the level difference between EV_x and EV_y :

$$LevelDist(x, y) = (Level_x - Level_y)^3 \quad (1)$$

The cubic function form amplifies the level difference, making the weighting more sensitive to large disparities. This amplification can quickly promote low-level EVs when they are accessed together with high-level EVs, since they will gain more EXP. To further quantify the overall level difference between EV_x and other EVs in the group, we calculate the $SumDist$ for EV_x :

$$SumDist(x) = \sum_{i=0}^{N-1} LevelDist(i, x) \quad (2)$$

The N is the number of EVs in the group, which is typically the number of EV tables.

Stage 2: Logarithmic compression and sigmoid-based normalization for ensuring stability. After summing the cubic differences for each EV, we obtain $SumDist$, whose range can be extremely wide due to the preceding amplification. Directly using $SumDist$ as a weight would lead to two problems: (1) Numerical instability. A single request with large level disparities could dominate the scoring. (2) Over-rewarding. Low-level EVs (especially newly inserted ones) might receive disproportionately high EXP, causing premature fixation in the cache. To mitigate these issues, we apply a two-step compression. In the first step, we take the absolute logarithm of $SumDist$ to narrow its dynamic range while preserving the order of magnitudes:

$$AbsLog(x) = \begin{cases} \ln(x) & x > 0 \\ 0 & x = 0 \\ -\ln(-x) & x < 0 \end{cases} \quad (3)$$

$$LogSum(x) = AbsLog(SumDist(x)) \quad (4)$$

This step implements a diminishing-returns principle: larger differences are compressed more heavily, preventing extreme values from overwhelming the scoring. In the second step, we use a *Sigmoid* function to smoothly map the compressed value into the interval (0, 1):

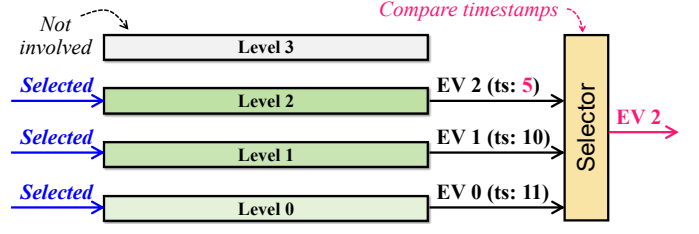


Fig. 9. The illustration of multi-level eviction (using 4 levels as an example, where the l_{min} is 0).

$$Sigmoid(x) = \frac{1}{1 + e^{-x}} \quad (5)$$

$$GAPW(x) = Sigmoid(LogSum(x)) \quad (6)$$

The *Sigmoid* function serves three purposes: (1) keeping the final GAPW weight within a reasonable range, (2) providing gradual response to moderate differences while saturating smoothly for extreme inputs, and (3) enabling the resulting weight to be directly used as a multiplicative factor for EXP gains. Moreover, to provide better control of the impact of GAPW on EXPs, we allow VineCache to linearly scale $LogSum$ before calculating the GAPW:

$$GAPW(x) = Sigmoid(A_{GAPW} * LogSum(x)) \quad (7)$$

The A_{GAPW} is a user-configured parameter, called GAPW coefficient. A larger GAPW coefficient leads to more pronounced differences in the EXPs gained across levels. We recommend a modest GAPW coefficient (e.g., 0.2 – 0.5) according to our sensitivity analysis (§IV-F).

Since we only use common math functions to calculate GAPWs, the overheads of calculating GAPWs are negligible. It is worth noting that although the computational complexity of Equation 2 is $O(N)$, it can be expressed as:

$$SumDist(x) = \sum_{i=0}^{N-1} L_i^3 - 3 * L_x * \sum_{i=0}^{N-1} L_i^2 + 3 * L_x^2 * \sum_{i=0}^{N-1} L_i - N * L_x^3 \quad (8)$$

For simplicity, we use L_i and L_x to respectively represent $Level_i$ and $Level_x$. The computational complexity of Equation 8 can be optimized to $O(1)$ by calculating three sums in advance. Therefore, the GAPW efficiently makes group hit-aware EXP scoring more adaptive while only introducing trivial overheads in calculating math functions.

D. Multi-Level Eviction Scheme

Since VineCache regards EVs of higher levels as more important, VineCache prioritizes evicting EVs with the lowest level. However, this may lead to premature evictions. Specifically, newly inserted EVs often correspond to low levels, which are prioritized for evictions. As the access pattern changes, these newly inserted EVs can become important. Frequent evictions on such EVs decrease the EV and group hit rates.

To address this problem, VineCache considers multiple levels in evictions and selects an EV to evict from these levels. As shown in Figure 9, VineCache uses a random-based method to decide the set of levels to be considered. Specifically,

suppose the lowest level is l_{min} , the level l has a $P_e^{(l-l_{min})}$ possibility to be considered, where P_e is the multi-level eviction probability and can be configured by users (e.g., 5%). The lower the level is, the higher the possibility of it being considered. Empty levels are skipped and not considered. After deciding the set of levels, VineCache collects the victim EVs (the EVs to be evicted) from the sub-caches of these levels. Each sub-cache uses a certain caching algorithm (e.g., LRU) to decide its victim EV. VineCache further selects the EV to evict among these victim EVs according to specific metrics. In our implementation, we use timestamps (indicating the time of last access) to evaluate victim EVs and evict the EV with the oldest timestamp.

In addition to alleviating the premature eviction problem, multi-level eviction delivers high cost-efficiency. First, the multi-level eviction involves a few computations (for generating random numbers) and memory accesses (for checking if the level is empty), hence only introducing negligible runtime overhead. Second, the multi-level eviction promotes quick demotions [43], [54] on high-level EVs, which facilitates evicting high-level but useless EVs and improving group hit rates. Moreover, this probabilistic selection inherently serves as a regularization mechanism against cache thrashing. In scenarios involving abrupt workload distribution shifts, strict priority-based evictions often lead to massive, disruptive replacements of entire cache tiers. By introducing a randomized eviction across levels, VineCache achieves a smoothing effect, gracefully evicting stale hot items while allowing newly inserted working sets time to accumulate sufficient EXPs.

It is worth noting that although the multi-level eviction may also evict EVs in the high levels, it does not decrease VineCache efficiency. This is because the caching algorithm adopted in the intra-level eviction and the selection among victim EVs prevent important EVs from being evicted. Overall, the multi-level eviction facilitates group hits under dynamic access patterns.

E. Aging Mechanism

To facilitate evictions on the old but useless EVs, VineCache embraces aging to shift to different access patterns without introducing high runtime overhead. Aging is a cost-efficient way for the cache to capture object recency by simply increasing the scores of objects. Its efficiency has been widely proved by prior works [55], [56]. In VineCache, we increase the gained EXPs of EVs.

In VineCache, after obtaining the gained EXPs based on the EXP function and GAPW, we multiply these obtained EXPs by the A_{exp} to obtain the final gained EXPs. A_{exp} is a global value initialized to 1. After performing a specific number of evictions (i.e., N_{aging} of the cache capacity), VineCache increases A_{exp} . The N_{aging} is called the aging threshold and can be configured by users (e.g., 10%). Since the increase of A_{exp} is controlled by N_{aging} , we simply increase the A_{exp} by a small value (e.g., 0.01) when the number of evictions exceeds the aging threshold. Such an aging design increases the gained EXPs of recently accessed EVs. In tandem with the

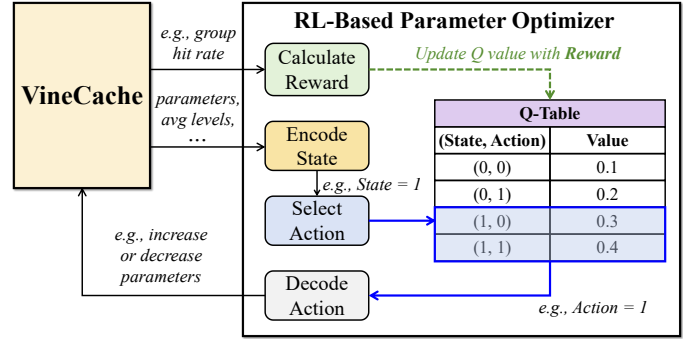


Fig. 10. The illustration of parameter optimizer (using 2 states and 2 actions as an example).

GAPW, newly inserted EVs have more opportunities to gain more EXPs and reach higher levels.

F. Parameter Optimizer

Modern caching systems [57], [58] feature many configurable parameters that can be fine-tuned to enhance cache efficiency. Similarly, VineCache is characterized by several configurable parameters (i.e., EXP functions, A_{GAPW} , P_e , and N_{aging}) that influence cache efficiency. These parameters collectively form complex and high-dimensional configuration space. Manually tuning these parameters is a challenging, time-consuming, and often suboptimal process. Furthermore, the optimal configuration is rarely static, which is highly dependent on the access patterns. Consequently, automated parameter optimization is essential for robust and efficient cache performance.

Prior works [59], [44], [60], [61] leverage multi-armed bandit (MAB) algorithm [59] to dynamically change the caching algorithms. Intuitively, VineCache could also leverage the MAB algorithm to optimize system parameters by treating each parameter set as an expert. However, this approach requires each expert to maintain its own independent caching data structure, leading to substantial time and space overheads. These inefficiencies make the MAB-based optimization strategy impractical for real-world implementation.

Reinforcement learning-based optimizer. To achieve a low-overhead and efficient parameter optimizer, VineCache leverages reinforcement learning (RL) [62], [63] to fine-tune parameters. RL is a powerful framework in machine learning where an agent learns to make sequential decisions by interacting with an environment, receiving feedback in the form of rewards. It has been widely used to solve optimization and control problems in complex systems such as database systems [64], [65] due to its ability to learn adaptive policies without requiring explicit supervision.

In VineCache, we implement an RL-based optimizer using the standard Q-learning algorithm [62], which introduces fewer overheads compared with deep learning-based RL algorithms. The Q-learning algorithm is a model-free, off-policy RL technique that estimates the expected cumulative rewards (called Q-values) for each action in a given state and uses these values to guide decision-making. The Q-values are stored in a Q-table and are iteratively updated during the learning process. To apply Q-learning within VineCache, we define the

following key components:

- **Agent:** The parameter optimizer acts as the learning agent, adapting its decisions to improve performance over time.
- **Environment:** The embedding cache serves as the environment, providing feedback to the agent.
- **States:** The system states are represented by encoding current parameter values and the average level of cached EVs.
- **Actions:** The possible changes to the parameters are encoded to represent the actions the agent can take.
- **Rewards:** To guide learning, we calculate the rewards based on the delta of group hit rates, which reflect improvements in cache efficiency.

By using reinforcement learning, VineCache dynamically tunes its parameters in response to changes in system behavior, ensuring robust and efficient optimization.

The workflow of the optimizer. Figure 10 illustrates the workflow. (1) An optimizer first obtains the current state by encoding the parameters and the average level of cached EVs. Specifically, each parameter and the average level are discretized into several values (e.g., 10). A set of these values forms a state. (2) The optimizer selects the next action using the ϵ -greedy algorithm [62]. It randomly selects an action with a probability of ϵ (linearly decreases from 1 to 0.01). Otherwise, it searches the Q-table and chooses the action with the highest Q-value based on the current state. (3) The optimizer calculates the reward of the last action and updates its corresponding Q-value using the standard updating formula [62] with the learning rate of 0.1. (4) The optimizer decodes the selected action and applies the corresponding operations to parameters. For each parameter, the optimizer increases/decreases its value or keeps it unchanged. We periodically optimize the parameters with a moderate frequency, achieving an efficient balance between timely parameter adjustment and low overheads.

Performance stability. VineCache monitors hit rates and inference performance during runtime. It records the parameters when these metrics become stable and show improvement. If a sudden performance degradation is detected, VineCache rolls back the parameters to the previously recorded stable state, thereby ensuring system performance stability.

G. VineCache Variants

We demonstrate VineCache’s generality and flexibility by incorporating two popular caching algorithms.

Recency-based. VineCache-LRU incorporate the least recent used (LRU) algorithm [35] into VineCache, which is a widely used recency-based caching algorithm. For each level, VineCache-LRU uses an LRU cache to manage EVs of this level, i.e., the sub-cache of this level. Upon a cache eviction, the LRU cache of each selected level chooses the least recently used EV as the victim EV of this level. We implement the LRU algorithm in $O(1)$ time complexity via hashmaps plus linked lists. Upon a *Level UP* of an EV, VineCache-LRU removes this EV from the LRU cache of its current level and inserts it into the LRU cache of the next level. Since LRU also promotes the involved item for each hit, VineCache-LRU only adds trivial

overheads caused by Level UPs. VineCache-LRU shows that implementing VineCache variants is simple.

Frequency-based. VineCache-MFU is implemented with the most frequently used (MFU) algorithm, which is similar to the least frequently used (LFU) algorithm [36]. We choose MFU algorithm because the EVs with more accesses are prioritized to be evicted among the EVs of the same level. The implementation of VineCache-MFU is similar to that of the VineCache-LRU, except that VineCache-MFU uses MFU caches to manage EVs of the same level. While LFU [66] can be implemented in $O(1)$ time complexity via linked lists and FIFO queues, MFU incurs high runtime overhead under such an implementation. This is because upon a Level UP of an EV, the frequency of the EV is not 0, and the MFU cache needs to locate the target FIFO queue, whose corresponding frequency is the same as that of the EV. Such a locating process needs to scan all cached EVs in the worst case, which introduces high overheads. Therefore, we choose to use tree-based sets to implement MFU caches, which provide more stable and better performance.

IV. PERFORMANCE EVALUATION

A. Experimental Setup

To evaluate VineCache’s performance on real deep recommendation systems, we integrate it into Meta’s open-sourced DLRM framework [38], [7]. Additionally, we use a caching benchmark [67] built on the Cache2K simulator [68] to efficiently measure hit rates across caching schemes with minimal time overhead.

Platform. Experiments are conducted on a Linux server with two Intel 26-core Xeon Gold 6230R CPUs, an NVIDIA V100 GPU, 192 GB DDR4 DRAM, a 960 GB Samsung SM883 SSD, and a 3.6 TB HDD, running Ubuntu 22.04 and Python 3.6.13. We use RocksDB [53] as the storage backend for managing EV tables. To ensure missed EVs are directly retrieved from backend storage, SSD access bypasses the OS page cache.

Comparisons. We compare VineCache with object-oriented caching schemes, including traditional schemes (*LRU* [35], *LFU* [36], [66]), adaptive schemes (*ARC* [42], *S3-FIFO* [43]), and the learning-based scheme (*Cacheus* [44]). We evaluate VineCache against *EV-LFU* [27], a state-of-the-art embedding caching scheme for recommendation systems. We evaluate VineCache variants *VineCache-LRU* and *VineCache-MFU*.

Datasets. We evaluate VineCache using three large-scale real-world datasets: the largest open-sourced CTR datasets released by Criteo (Criteo-Kaggle [39], Criteo-TB [40]) and the Avazu CTR dataset [41]. These datasets provide features and click feedback for millions of advertisements. Criteo datasets include 13 dense integer features and 26 sparse categorical features, with over 156 billion dense features and 800 million unique sparse categorical features. The sparse features are stored in 26 EV tables with the embedding dimensions of 36. Avazu CTR dataset includes over 49 million sparse features in 22 EV tables. To evaluate VineCache under varying access patterns, we divide each dataset into 200

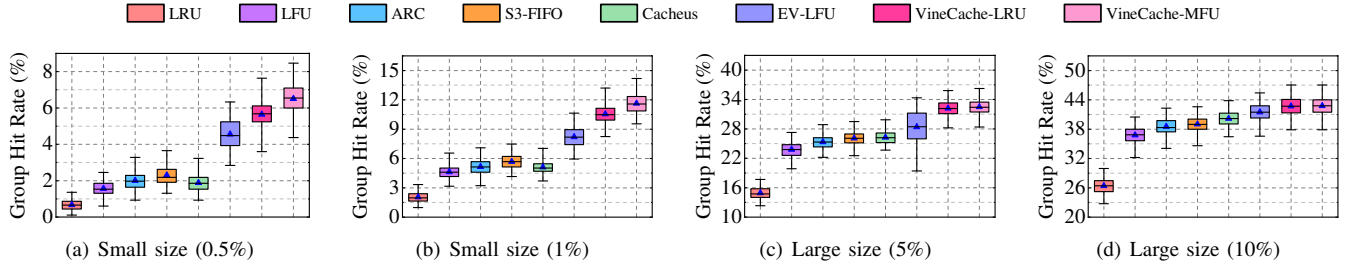


Fig. 11. The group hit rates of different caching schemes with various cache sizes.

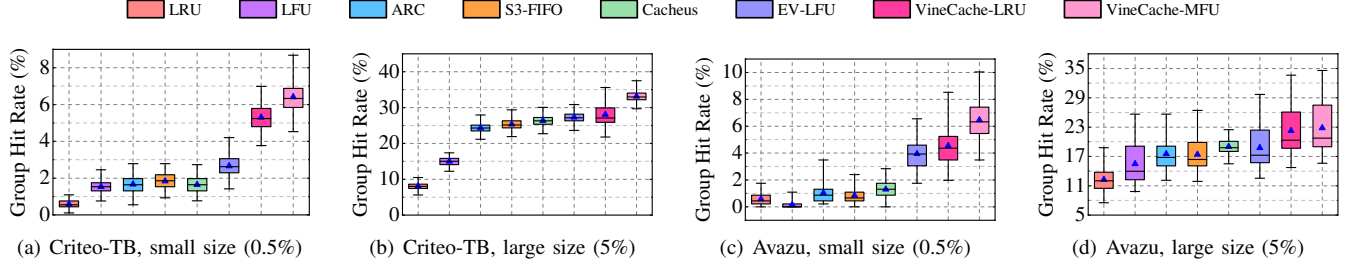


Fig. 12. The group hit rates of different caching schemes under various datasets.

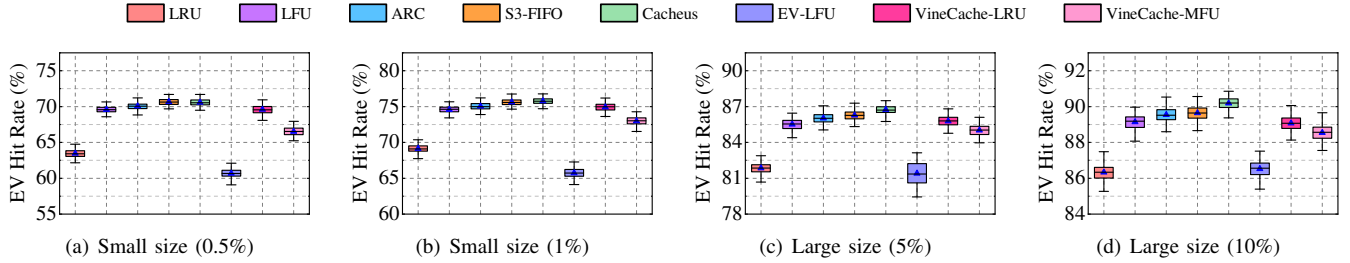


Fig. 13. The EV hit rates of different caching schemes with various cache sizes.

workloads, each containing 0.5% continuous requests of the dataset. For performance metrics like EV/group hit rates, EV lookup time, and inference latency, we use box plots to present aggregated results. The box plot displays the 25th and 75th percentiles, with the whiskers extending to the 1st and 99th percentiles. The line within the box indicates the median value, while the blue triangle shows the average value.

Default configurations. Unless stated otherwise, EVs are stored in FP32 format, and EV tables are stored on SSDs. For VineCache, the default settings include using a piecewise EXP function, storing EXPs in FP32 format, setting the GAPW coefficient to 0.3, the multi-level eviction probability to 5%, and the aging threshold to 10% of cache capacity. All results are reported after cache warm-up.

B. Cache Efficiency

Group hit rate. We evaluate the group hit rates of VineCache and other state-of-the-art caching schemes across different cache sizes, simulating scenarios where the capacity ratio between memory and storage varies. As shown in Figure 11, VineCache consistently demonstrates the highest efficiency among all caching schemes on the Criteo-Kaggle dataset. Specifically, compared with object-oriented caching designs, VineCache-LRU achieves higher group hit rates, improving by an average of 3.62 $\times$ for small cache sizes and 1.75 $\times$ for large cache sizes. These significant improvements stem

from VineCache’s hit count-based EXP scoring mechanism, which captures EV lookup patterns, while the object-oriented schemes are agnostic to the group hits. Among the baselines, EV-LFU achieves the highest group hit rates due to its max-hit-based priority mechanism. However, this max-based approach struggles to adapt to varying access patterns, limiting its ability to deliver higher group hit rates. In contrast, the dynamic EXP-based scoring system enables VineCache to effectively adapt to diverse access patterns, hence providing higher group hit rates than EV-LFU by 1.23 $\times$ on average. For another VineCache variant, VineCache-MFU achieves higher group hit rates than other caching schemes by 4.15 $\times$ and 1.77 $\times$ for small and large cache sizes, respectively. These results highlight VineCache’s flexible and efficient integration of diverse caching algorithms.

To demonstrate the generality of VineCache, we evaluate it across various datasets, as shown in Figure 12. Specifically, for the Criteo-TB and Avazu datasets, VineCache-LRU achieves higher group hit rates than all baselines by 3.98 $\times$ and 4.48 $\times$ for small cache sizes, and 1.81 $\times$ and 1.50 $\times$ for large cache sizes, respectively. These results show that VineCache consistently enhances group hit rates across datasets.

EV hit rate. Figure 13 shows the EV hit rates of different caching schemes with various cache sizes on the Criteo-Kaggle dataset. The object-oriented caching schemes deliver high EV hit rates, since they are optimized for the single-

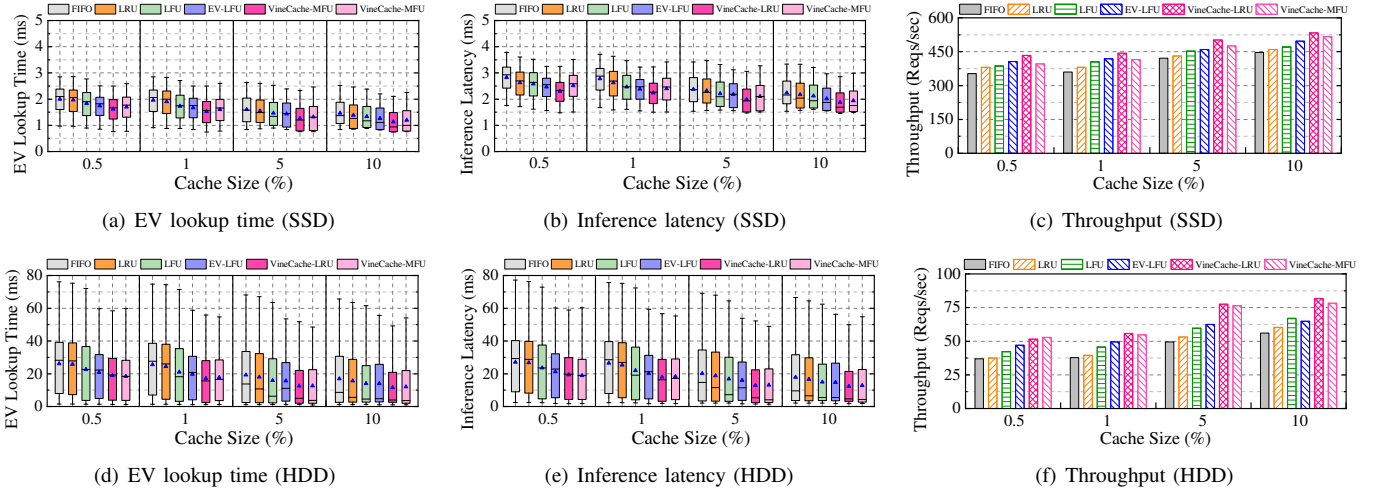


Fig. 14. The end-to-end performance with different caching schemes under various configurations.

object hit rate. However, EV-LFU exhibits the lowest EV hit rates among all caching schemes for most cache sizes. This is because EV-LFU maintains the priorities of non-essential EVs unchanged across workloads, limiting its efficiency. In contrast, VineCache variants deliver EV hit rates comparable to those of object-oriented caching schemes. Specifically, VineCache-LRU improves EV hit rates by up to $1.15\times$ over EV-LFU. We attribute these improvements to its dynamic EXP scoring system, which efficiently evict non-essential EVs and insert useful ones in a timely manner.

C. End-to-End Performance

To evaluate how VineCache enhances real deep recommendation systems, we measure the end-to-end performance of DLRM with different caching schemes on the Criteo-Kaggle dataset, using LRU, LFU, and EV-LFU as baselines. While there are more elaborated baseline schemes, they are slower than LRU, since they only provide similar EV and group hit rates compared with LRU, but cause much higher runtime overhead for cache management.

EV lookup time, inference latency, and throughput. Figure 14(a) shows the EV lookup time for different caching schemes and cache sizes. VineCache-LRU reduces lookup time by 17.6% and 20.6% for small and large caches, respectively. Figure 14(b) indicates that VineCache-LRU achieves the lowest inference latency, outperforming baselines by 13.8% and 14.1%. These gains result from its high EV and group hit rates. Figure 14(c) further shows that VineCache-LRU consistently improves throughput by 13.9% across all cache sizes. The significant improvements stem from that VineCache-LRU achieves high hit rates with minimal runtime overhead.

In contrast, VineCache-MFU, despite higher group hit rates, has higher latencies and lower throughputs than VineCache-LRU. This gap arises because VineCache-MFU employs tree-based data structures for EV management, causing non-trivial runtime overhead. In comparison, VineCache-LRU incurs negligible overhead compared with its original LRU, fully leveraging the benefits of high group hit rates. This highlights the importance to balance high group hit rates and low runtime

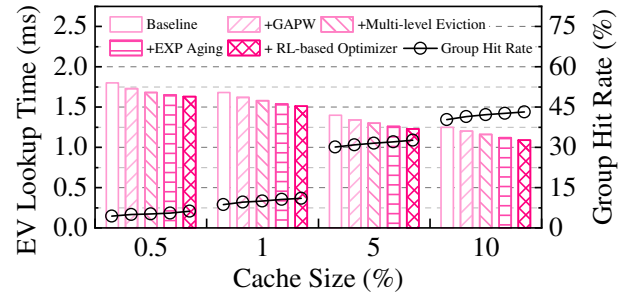


Fig. 15. The ablation study of VineCache designs.

overhead in embedding cache designs.

Heterogeneous storage devices. To demonstrate the effectiveness of VineCache on different storage devices, we evaluate DLRM performance with EV tables stored on HDD. Compared with SSD-based storage, VineCache-LRU delivers higher performance improvements on HDD. As shown in Figures 14(d), 14(e), and 14(f), VineCache-LRU consistently outperforms all baselines, achieving up to 57% higher improvements. This is because HDD’s access latency, being orders of magnitude higher than SSD, emphasizes the advantage of high group hit rates in HDD-based storage.

D. Ablation Study

To evaluate the impact of the proposed mechanisms on cache efficiency and system performance, we perform an ablation study on VineCache using the Criteo-Kaggle dataset, leveraging VineCache-LRU as an example. As shown in Figure 15, we begin with the *Baseline* that only employs hit count-based EXP functions, and incrementally add each proposed mechanism. VineCache-MFU shows similar impacts.

The results show that adopting GAPW improves group hit rates by up to $1.13\times$ and reduces EV lookup time by 4.4%, demonstrating its ability to capture EV lookup patterns and identify EV importance across access patterns. The multi-level eviction mechanism increases group hit rates by $1.08\times$ and reduces EV lookup time by 3.4% by preventing premature evictions and quickly demoting high-level but non-essential EVs. The aging mechanism enhances efficiency by up to

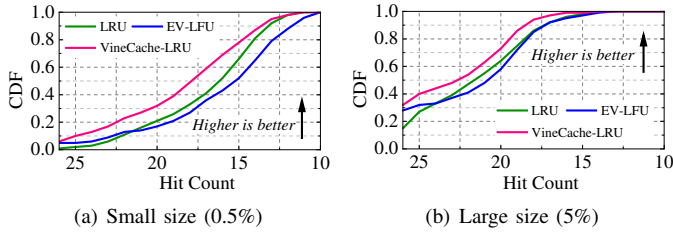


Fig. 16. The cumulative distribution functions of hit counts with different caching schemes.

1.07 $\times$, highlighting its contribution in adapting to diverse access patterns. Additionally, the RL-based parameter optimizer improves efficiency by 1.04 $\times$ on average. These results demonstrate the critical role of these mechanisms in improving cache efficiency and EV lookup performance.

E. Distributions of Hit Counts

To understand how VineCache improves cache efficiency and inference performance, we analyze the cumulative distribution functions (CDFs) of hit counts across caching schemes on the Criteo-Kaggle dataset. As shown in Figure 16, EV-LFU produces more group hits than LRU but has lower peak hit counts. This is because EV-LFU’s max-based scoring over-prioritizes old EVs and fails to evict them while neglecting other critical EVs. In contrast, VineCache’s hit count-based EXP mechanism identifies important EVs across workloads and dynamically adjusts cache contents as access pattern changes. Compared with LRU and EV-LFU, VineCache delivers 2.19 $\times$ and 1.29 $\times$ more high-hit EV lookups (hit count > 20) for small and large cache sizes, respectively. A higher portion of high-hit EVs increases the likelihood of group hits, contributing to better group hit rates. The shift in distribution towards higher hit counts for VineCache indicates that the system successfully identifies and retains the critical EVs required for group hits, effectively resolving the granularity mismatch observed in existing caching schemes.

F. Sensitivity Analysis

We explore how VineCache’s parameters affect cache efficiency using Criteo-Kaggle dataset. Unless specified, we only analyze VineCache-LRU, since VineCache-MFU exhibits similar trends. The parameter optimizer is disabled.

The EXP function. Figure 17 shows how different EXP functions impact group hit rates. For small cache sizes, the exponential function performs better, while for large cache sizes, it performs worse. Although no single EXP function consistently delivers the highest hit rate, their overall impact on cache efficiency is limited. It is worth noting that VineCache outperforms all caching schemes across all cache sizes, regardless of the chosen EXP function.

The GAPW coefficient. Figure 18 demonstrates the effect of the GAPW coefficient on cache efficiency. An unsuitable GAPW coefficient reduces group hit rates. A large coefficient makes EXP values too extreme (near 0 or 1) after applying the *Sigmoid* function, decreasing the efficiency. Conversely, a small coefficient reduces distinctions among EXPs of EVs at different levels, nullifying GAPW.

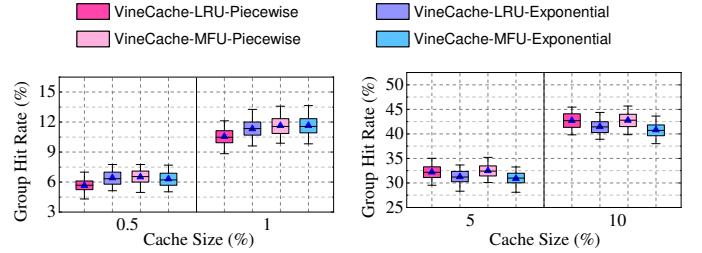


Fig. 17. The efficiency impacts of the EXP function.

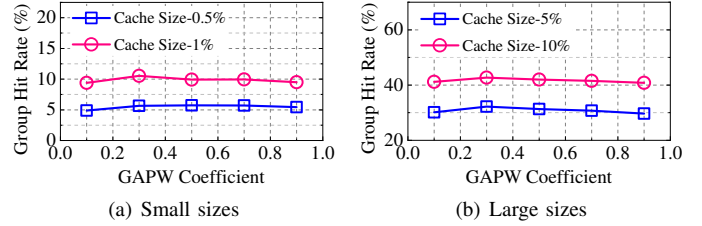


Fig. 18. The efficiency impacts of the GAPW coefficient.

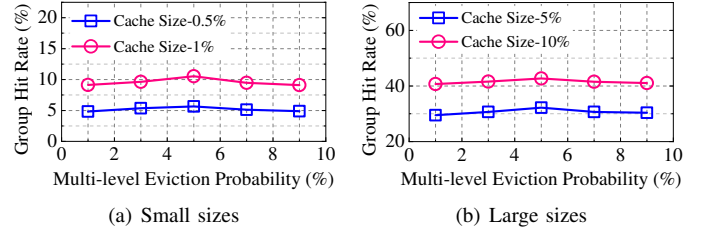


Fig. 19. The efficiency impacts of the eviction probability.

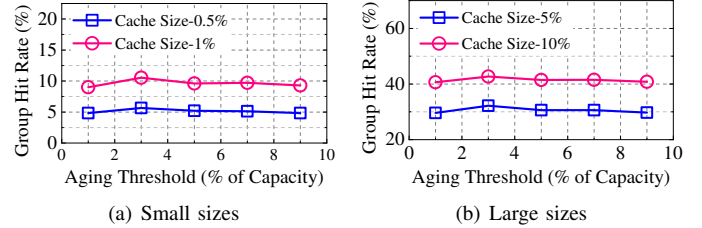


Fig. 20. The efficiency impacts of the aging threshold.

The multi-level eviction probability. Figure 19 presents the impacts of the multi-level eviction probability (i.e., P_e) on cache efficiency. When using a too-large probability, VineCache-LRU would consider too many levels, leading to evictions on important EVs of high levels. On the other hand, a too-small probability would disable multi-level eviction.

The aging threshold. Figure 20 illustrates the impact of aging thresholds on cache efficiency. As the threshold increases, VineCache prioritizes caching high-frequency EVs. However, when some of these EVs turn cold, VineCache overestimates their importance and avoids evicting them, reducing cache efficiency. For a small threshold, VineCache emphasizes EVs with higher recencies. Overall, VineCache remains robust to changes in aging thresholds, exhibiting only slight fluctuations (2.6%) in group hit rates.

The number of EV tables. We also evaluate the effect of the number of EV tables to demonstrate VineCache’s generality. As shown in Figure 21, VineCache improves group hit rates ef-

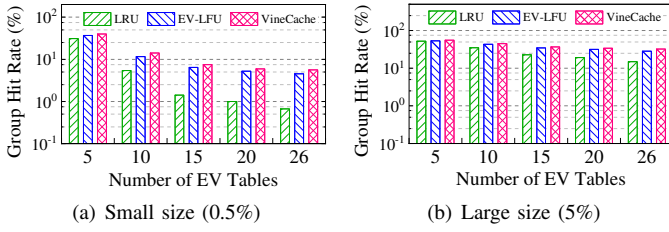


Fig. 21. The efficiency impacts of the number of EV tables.

fectively across varying numbers of EV tables. Although EXP functions calculate gained EXP based on hit counts (related to table counts), their trends stay consistent (i.e., gained EXP increasing with hit count). Therefore, the hit count-based EXP scoring efficiently identifies important EVs across different EV table numbers.

G. Overheads of VineCache

Metadata overhead. VineCache introduces trivial metadata overhead for storing the EXPs and timestamps of EVs. With EXP values stored in FP32 format, the overhead is less than 5.6% for EVs with 36 dimensions stored in FP32 format.

Computation overhead. VineCache incurs limited computation overhead since it relies on simple mathematical functions to calculate gained EXP. The computation time is under 0.03 ms, which is negligible compared to the average EV lookup time (e.g., over 1.5 ms for VineCache-LRU). Additionally, the overhead from the parameter optimizer can be constrained to less than 0.5% by adjusting the optimization frequency.

Management overhead. Dynamic EXP-based scoring does not introduce extra overhead to cache management because traditional cache management already involves checking and updating metadata. Hence, VineCache only introduces trivial overhead for reading and writing metadata without increasing overall management costs.

V. DISCUSSIONS

Finding optimal parameters. Finding parameters for reasonable performance is straightforward, but achieving optimal performance is challenging. Nonetheless, VineCache is robust to parameter variations due to the RL-based optimizer, and default parameters deliver cost-efficient performance across diverse workloads. Sensitivity analysis confirms that VineCache maintains high efficiency despite parameter changes.

The generality of VineCache. VineCache highlights the critical role of caching scheme and efficiency in end-to-end system performance. Its hardware-agnostic design supports diverse hardware hierarchies (e.g., using GPU memory as a cache for CPU DRAM). Moreover, existing caching optimizations (e.g., mixed-precision caching [27] and cache-workflow co-design [18]) are compatible with VineCache and can collaborate with it to boost performance. Furthermore, beyond recommendation systems, we believe VineCache can inspire caching schemes for applications with all-or-nothing workflows, such as deep learning on graphs (e.g., Deep Graph Library [69]) and similarity search (e.g., Faiss [70]). In these scenarios, group-based embedding table lookups would benefit from VineCache’s group-hit optimizations.

VI. RELATED WORK

Caching schemes in databases. Traditional caching schemes aim to maximize single-object hit rates based on object features like frequency [71], [72], [66], [36], recency [73], [74], [35], or object sizes [75], [76]. Learning-based schemes [77], [60], [78], [44], [79], [80] combine multiple features to enhance hit rates. However, these object-oriented approaches are agnostic to the all-or-nothing property of embedding caches, limiting their efficiency in recommendation systems. While prior designs [81], [82], [83], [56] exploit the all-or-nothing property to maximize page or transactional hit rates, they are tailored for scenarios like parallel tasks [81], [82] or transactions [56] and are inefficient for recommendation inference. In contrast, VineCache employs a dynamic EXP-based scoring system to identify important EVs across diverse access patterns, enhancing group hit rates and enabling high-performance recommendation inference.

Caches tailored for recommendation systems. Existing schemes optimize cache utilization during recommendation system training [84], [18], [85], but they focus on training rather than inference and overlook the benefits of enhanced cache efficiency. In contrast, VineCache boosts recommendation system performance with an adaptive, group hit-aware caching scheme. Some approaches [27], [86] improve cache efficiency by storing lower-precision embeddings (e.g., FP16 instead of FP32). VineCache can similarly leverage mixed-precision to store more EVs and achieve higher hit rates. Furthermore, heterogeneous embedding caching schemes cache EVs in GPU and host memory to optimize embedding reduction [87], sparse operations near memory [31], [88], and high-utilization co-designed workflows [18]. VineCache focuses on improving the caching algorithm for higher group hit rates and hence is orthogonal to these designs. These schemes can incorporate VineCache to improve efficiency.

VII. CONCLUSION

We have designed, implemented, and evaluated VineCache, an adaptive and group-aware embedding caching scheme designed specifically to resolve the granularity mismatch in deep recommendation inference. VineCache proposes an EXP-based scoring mechanism to efficiently capture the EV lookup patterns. Moreover, VineCache leverages the group-aware priority weighting to dynamically identify important EVs. We further adopt multi-level eviction and aging mechanisms to make the VineCache adaptive to different access patterns. Compared with existing caching schemes, VineCache consistently provides better end-to-end inference performance, highlighting the importance of improving group-level embedding cache efficiency in enhancing the deep recommendation inference performance.

ACKNOWLEDGMENTS

This work was supported in part by National Natural Science Foundation of China (NSFC) under Grant No. 62125202, U22B2022, 62621005 and Hubei Provincial Natural Science Foundation of China (No. 2026AFA002).

AI-GENERATED CONTENT ACKNOWLEDGEMENT

Authors did not use artificial intelligence (AI) to generate content.

REFERENCES

- [1] J. Wang, P. Huang, H. Zhao, Z. Zhang, B. Zhao, and D. L. Lee, "Billion-scale commodity embedding for e-commerce recommendation in alibaba," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'18)*, 2018.
- [2] G. Linden, B. Smith, and J. York, "Amazon.com recommendations: Item-to-item collaborative filtering," *IEEE Internet Computing*, vol. 7, no. 1, pp. 76–80, 2003.
- [3] P. Gupta, A. Goel, J. Lin, A. Sharma, D. Wang, and R. Zadeh, "WTF: the who to follow service at twitter," in *Proceedings of the 22nd International World Wide Web Conference (WWW'13)*, 2013.
- [4] U. Gupta, C. Wu, X. Wang, M. Naumov, B. Reagen, D. Brooks, B. Cottel, K. M. Hazelwood, M. Hempstead, B. Jia, H. S. Lee, A. Malevich, D. Mudigere, M. Smelyanskiy, L. Xiong, and X. Zhang, "The architectural implications of facebook's dnn-based personalized recommendation," in *Proceedings of the 26th IEEE International Symposium on High Performance Computer Architecture (HPCA'20)*, 2020.
- [5] P. Covington, J. Adams, and E. Sargin, "Deep neural networks for youtube recommendations," in *Proceedings of the 10th ACM Conference on Recommender Systems (RecSys'16)*, 2016.
- [6] Meta, "How machine learning powers Facebook's News Feed ranking algorithm. <https://engineering.fb.com/2021/01/26/ml-applications/news-feed-ranking/>," 2021.
- [7] —, "Deep Learning Recommendation Model for Personalization and Recommendation Systems. <https://github.com/facebookresearch/dlrm>," 2023.
- [8] C. Sima, Y. Fu, M. Sit, L. Guo, X. Gong, F. Lin, J. Wu, Y. Li, H. Rong, P. Aublin, and L. Mai, "Ekko: A large-scale deep learning recommender system with low-latency model update," in *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)*, 2022, pp. 821–839.
- [9] H. Guo, R. Tang, Y. Ye, Z. Li, and X. He, "Deepfm: A factorization-machine based neural network for CTR prediction," in *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence (IJCAI'17)*, 2017, pp. 1725–1731.
- [10] Y. Qu, H. Cai, K. Ren, W. Zhang, Y. Yu, Y. Wen, and J. Wang, "Product-based neural networks for user response prediction," in *Proceedings of the 16th IEEE International Conference on Data Mining (ICDM'16)*, 2016, pp. 1149–1154.
- [11] W. Song, C. Shi, Z. Xiao, Z. Duan, Y. Xu, M. Zhang, and J. Tang, "Autoint: Automatic feature interaction learning via self-attentive neural networks," in *Proceedings of the 28th ACM International Conference on Information and Knowledge Management (CIKM'19)*, 2019, pp. 1161–1170.
- [12] G. Zhou, X. Zhu, C. Song, Y. Fan, H. Zhu, X. Ma, Y. Yan, J. Jin, H. Li, and K. Gai, "Deep interest network for click-through rate prediction," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'18)*, 2018, pp. 1059–1068.
- [13] D. D. Kalamkar, E. Georganas, S. Srinivasan, J. Chen, M. Shiryaev, and A. Heinecke, "Optimizing deep learning recommender systems training on CPU cluster architectures," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'20)*, 2020, p. 43.
- [14] R. Jain, T. Chou, O. Kayiran, J. Kalamatianos, G. H. Loh, M. T. Kandemir, and C. R. Das, "Load and mlp-aware thread orchestration for recommendation systems inference on cpus," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'25)*, 2025, pp. 589–603.
- [15] C. A. Gomez-Urbe and N. Hunt, "The netflix recommender system: Algorithms, business value, and innovation," *ACM Transactions on Management Information Systems*, vol. 6, no. 4, pp. 13:1–13:19, 2016.
- [16] B. Smith and G. Linden, "Two decades of recommender systems at amazon.com," *IEEE Internet Computing*, vol. 21, no. 3, pp. 12–18, 2017.
- [17] Z. Wang, Y. Wang, B. Feng, D. Mudigere, B. Muthiah, and Y. Ding, "El-rec: Efficient large-scale recommendation model training via tensor-train embedding table," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'22)*, 2022, pp. 70:1–70:14.
- [18] M. Xie, Y. Lu, J. Lin, Q. Wang, J. Gao, K. Ren, and J. Shu, "Fleche: an efficient GPU embedding cache for personalized recommendations," in *Proceedings of the 17th European Conference on Computer Systems (EuroSys'22)*, 2022, pp. 402–416.
- [19] X. Sun, W. Wang, S. Qiu, R. Yang, S. Huang, J. Xu, and Z. Wang, "STRONGHOLD: fast and affordable billion-scale deep learning model training," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'22)*, 2022, pp. 71:1–71:17.
- [20] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," in *Proceedings of the 3rd International Conference on Learning Representations (ICLR'15)*, 2015.
- [21] A. Lerer, L. Wu, J. Shen, T. Lacroix, L. Wehrstedt, A. Bose, and A. Peysakhovich, "Pytorch-biggraph: A large scale graph embedding system," in *Proceedings of Machine Learning and Systems (MLSys'19)*, 2019.
- [22] J. Mohoney, R. Waleffe, H. Xu, T. Rekatsinas, and S. Venkataraman, "Marius: Learning massive graph embeddings on a single machine," in *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI'21)*, 2021, pp. 533–549.
- [23] Z. Qiu, J. Yang, J. Zhang, C. Li, X. Ma, Q. Chen, M. Yang, and Y. Xu, "Frozenhot cache: Rethinking cache management for modern hardware," in *Proceedings of the 18th European Conference on Computer Systems (EuroSys'23)*, 2023.
- [24] N. P. Jouppi, D. H. Yoon, M. Ashcraft, M. Gottscho, T. B. Jablin, G. Kurian, J. Laudon, S. Li, P. C. Ma, X. Ma, T. Norrie, N. Patil, S. Prasad, C. Young, Z. Zhou, and D. A. Patterson, "Ten lessons from three generations shaped google's tpuv4i: Industrial product," in *Proceedings of the 48th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA'21)*, 2021.
- [25] C. Zeng, L. Luo, Q. Ning, Y. Han, Y. Jiang, D. Tang, Z. Wang, K. Chen, and C. Guo, "FAERY: an fpga-accelerated embedding-based retrieval system," in *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)*, 2022, pp. 841–856.
- [26] F. Lai, W. Zhang, R. Liu, W. Tsai, X. Wei, Y. Hu, S. Devkota, J. Huang, J. Park, X. Liu, Z. Chen, E. Wen, P. Rivera, J. You, C. J. Chen, and M. Chowdhury, "Adaembd: Adaptive embedding for large-scale recommendation models," in *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI'23)*, 2023, pp. 817–831.
- [27] D. H. Kurniawan, R. Wang, K. S. Zulkifli, F. A. Wiranata, J. Bent, Y. Vigfusson, and H. S. Gunawi, "Evsore: 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 (ASPLOS'23)*, 2023, pp. 281–294.
- [28] Y. Hua, X. Liu, and I. Stoica, "The golden rule of big memory: Persistence is not harmful," *Communications of the ACM*, vol. 69, no. 5, pp. 51–55, 2026.
- [29] W. Jiang, Z. He, S. Zhang, T. B. Preußer, K. Zeng, L. Feng, J. Zhang, T. Liu, Y. Li, J. Zhou, C. Zhang, and G. Alonso, "Microrec: Efficient recommendation inference by hardware and data structure solutions," in *Proceedings of Machine Learning and Systems (MLSys'21)*, 2021.
- [30] W. Jiang, Z. He, S. Zhang, K. Zeng, L. Feng, J. Zhang, T. Liu, Y. Li, J. Zhou, C. Zhang, and G. Alonso, "Fleetrec: Large-scale recommendation inference on hybrid GPU-FPGA clusters," in *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD'21)*, 2021, pp. 3097–3105.
- [31] L. Ke, U. Gupta, B. Y. Cho, D. Brooks, V. Chandra, U. Diril, A. Firoozshahian, K. M. Hazelwood, B. Jia, H. S. Lee, M. Li, B. Maher, D. Mudigere, M. Naumov, M. Schatz, M. Smelyanskiy, X. Wang, B. Reagen, C. Wu, M. Hempstead, and X. Zhang, "Recnmp: Accelerating personalized recommendation with near-memory processing," in *Proceedings of the 47th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA'20)*, 2020, pp. 790–803.
- [32] G. Zhou, N. Mou, Y. Fan, Q. Pi, W. Bian, C. Zhou, X. Zhu, and K. Gai, "Deep interest evolution network for click-through rate prediction," in *Proceedings of the 33rd AAAI Conference on Artificial Intelligence (AAAI'19)*, 2019, pp. 5941–5948.
- [33] H. Ye, S. Vedula, Y. Chen, Y. Yang, A. M. Bronstein, R. G. Dreslinski, T. N. Mudge, and N. Talati, "GRACE: A scalable graph-based approach to accelerating recommendation model inference," in *Proceedings of*

the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23), 2023, pp. 282–301.

- [34] X. Song, Y. Zhang, R. Chen, and H. Chen, “UGACHE: A unified GPU cache for embedding-based deep learning,” in *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP'23)*. ACM, 2023, pp. 627–641.
- [35] A. Dan and D. F. Towsley, “An approximate analysis of the LRU and FIFO buffer replacement schemes,” in *Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems (SIGMETRICS'90)*, 1990, pp. 143–152.
- [36] D. Lee, J. Choi, J. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. Kim, “On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies,” in *Proceedings of the 1999 ACM SIGMETRICS international conference on Measurement and modeling of computer systems (SIGMETRICS'99)*, 1999, pp. 134–143.
- [37] Z. Wang, Y. Wang, J. Deng, D. Zheng, A. Li, and Y. Ding, “RAP: resource-aware automated GPU sharing for multi-gpu recommendation model training and input preprocessing,” in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'24)*, 2024, pp. 964–979.
- [38] M. Naumov, D. Mudigere, H. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C. Wu, A. G. Azzolini, D. Dzhulgakov, A. Malle-
vich, I. Cherniavskii, Y. Lu, R. Krishnamoorthi, A. Yu, V. Kondratenko, S. Pereira, X. Chen, W. Chen, V. Rao, B. Jia, L. Xiong, and M. Smelyan-
skiy, “Deep learning recommendation model for personalization and
recommendation systems,” *CoRR*, vol. abs/1906.00091, 2019.
- [39] Kaggle, “Display Advertising Challenge. <https://www.kaggle.com/c/criteo-display-ad-challenge>,” 2014.
- [40] C. A. Lab, “Download Criteo Terabyte Click Logs Dataset. <https://labs.criteo.com/2013/12/download-terabyte-click-logs/>,” 2013.
- [41] Kaggle, “Click-Through Rate Prediction. <https://www.kaggle.com/c/ava-zu-ctr-prediction>,” 2014.
- [42] N. Megiddo and D. S. Modha, “ARC: A self-tuning, low overhead replacement cache,” in *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST'03)*, 2003.
- [43] J. Yang, Y. Zhang, Z. Qiu, Y. Yue, and R. Vinayak, “FIFO queues are all you need for cache eviction,” in *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP'23)*, 2023, pp. 130–149.
- [44] L. V. Rodriguez, F. B. Yusuf, S. Lyons, E. Paz, R. Rangaswami, J. Liu, M. Zhao, and G. Narasimhan, “Learning cache replacement with CACHEUS,” in *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST'21)*, 2021, pp. 341–354.
- [45] J. Lian, X. Zhou, F. Zhang, Z. Chen, X. Xie, and G. Sun, “xdeepfm: Combining explicit and implicit feature interactions for recommender systems,” in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'18)*, 2018, pp. 1754–1763.
- [46] Z. Xu, Y. Qu, W. Zhang, L. Liang, and H. Chen, “Inbox: Recommendation with knowledge graph using interest box embedding,” *Proceedings of the VLDB Endowment (PVLDB)*, vol. 17, no. 13, pp. 4641–4654, 2024.
- [47] Z. Chen, C. Zhang, C. Xu, Z. Zhang, J. Wang, W. Qian, and A. Zhou, “Scheduling data processing pipelines for incremental training on mlp-based recommendation models,” in *Proceedings of the 2025 International Conference on Management of Data (SIGMOD'25)*, 2025, pp. 350–363.
- [48] X. He, L. Liao, H. Zhang, L. Nie, X. Hu, and T. Chua, “Neural collaborative filtering,” in *Proceedings of the 26th International Conference on World Wide Web (WWW'17)*, 2017, pp. 173–182.
- [49] M. Naumov, D. Mudigere, H. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C. Wu, A. G. Azzolini, D. Dzhulgakov, A. Malle-
vich, I. Cherniavskii, Y. Lu, R. Krishnamoorthi, A. Yu, V. Kondratenko, S. Pereira, X. Chen, W. Chen, V. Rao, B. Jia, L. Xiong, and M. Smelyan-
skiy, “Deep learning recommendation model for personalization and
recommendation systems,” *CoRR*, vol. abs/1906.00091, 2019.
- [50] K. Bao, J. Zhang, Y. Zhang, W. Wang, F. Feng, and X. He, “Tallrec: An effective and efficient tuning framework to align large language model with recommendation,” in *Proceedings of the 17th ACM Conference on Recommender Systems (RecSys'23)*, 2023, pp. 1007–1014.
- [51] J. Zhang, R. Xie, Y. Hou, W. X. Zhao, L. Lin, and J. Wen, “Recommendation as instruction following: A large language model empowered recommendation approach,” *CoRR*, vol. abs/2305.07001, 2023.
- [52] L. Friedman, S. Ahuja, D. Allen, Z. Tan, H. Sidahmed, C. Long, J. Xie, G. Schubiner, A. Patel, H. Lara, B. Chu, Z. Chen, and M. Tiwari, “Leveraging large language models in conversational recommender systems,” *CoRR*, vol. abs/2305.07961, 2023.
- [53] Meta, “RocksDB: A Persistent Key-Value Store for Flash and RAM Storage <https://github.com/facebook/rocksdb>,” 2025.
- [54] J. Yang, Z. Qiu, Y. Zhang, Y. Yue, and K. V. Rashmi, “FIFO can be better than LRU: the power of lazy promotion and quick demotion,” in *Proceedings of the 19th Workshop on Hot Topics in Operating Systems (HotOS'23)*, 2023, pp. 70–79.
- [55] J. Aguilar and E. L. Leiss, “A web proxy cache coherency and replacement approach,” in *Proceedings of the First Asia-Pacific Conference on Web Intelligence: Research and Development (WI'01)*, 2001.
- [56] A. Cheng, D. Chu, T. Li, J. Chan, N. Crooks, J. M. Hellerstein, I. Stoica, and X. Yu, “Take out the trache: Maximizing (tra)nsactional ca(cache) hit rate,” in *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI'23)*, 2023.
- [57] Redis, “The preferred, fastest, and most feature-rich cache, data structure server, and document and vector query engine. <https://github.com/redis/redis.git>,” 2025.
- [58] Memcached, “A high-performance, distributed memory object caching system. <https://memcached.org/>,” 2025.
- [59] I. Ari, A. Amer, R. B. Gramacy, E. L. Miller, S. A. Brandt, and D. D. E. Long, “ACME: adaptive caching using multiple experts,” in *Distributed Data & Structures 4, Records of the 4th International Meeting (WDAS'02)*, vol. 14, 2002, pp. 143–158.
- [60] G. Vietri, L. V. Rodriguez, W. A. Martinez, S. Lyons, J. Liu, R. Rangaswami, M. Zhao, and G. Narasimhan, “Driving cache replacement with ml-based lecar,” in *Proceedings of the 10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'18)*, 2018.
- [61] J. Shen, P. Zuo, X. Luo, Y. Su, J. Gu, H. Feng, Y. Zhou, and M. R. Lyu, “Ditto: An elastic and adaptive memory-disaggregated caching system,” in *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP'23)*, 2023, pp. 675–691.
- [62] C. J. C. H. Watkins and P. Dayan, “Technical note q-learning,” *Machine Learning*, vol. 8, pp. 279–292, 1992.
- [63] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, “Playing atari with deep reinforcement learning,” *CoRR*, vol. abs/1312.5602, 2013.
- [64] J. Zhang, K. Zhou, G. Li, Y. Liu, M. Xie, B. Cheng, and J. Xing, “CDBTune: An efficient deep reinforcement learning-based automatic cloud database tuning system,” *VLDB Journal*, vol. 30, no. 6, pp. 959–987, 2021.
- [65] G. Li, X. Zhou, S. Li, and B. Gao, “Qtune: A query-aware database tuning system with deep reinforcement learning,” *Proceedings of the VLDB Endowment (PVLDB)*, vol. 12, no. 12, pp. 2118–2130, 2019.
- [66] D. Mátáni, K. Shah, and A. Mitra, “An O(1) algorithm for implementing the LFU cache eviction scheme,” *CoRR*, vol. abs/2110.11602, 2021.
- [67] D. H. Kurniawan, “Cache Algorithm Benchmark. <https://github.com/ucare-uchicago/cache-benchmark>,” 2023.
- [68] Cache2K, “Benchmarks for Java In Memory Caches. <https://github.com/cache2k/cache2k-benchmark>,” 2023.
- [69] D. G. Library, “Python package built to ease deep learning on graph, on top of existing DL frameworks. <https://github.com/dmcl/dgl>,” 2025.
- [70] Meta, “A library for efficient similarity search and clustering of dense vectors. <https://github.com/facebookresearch/faiss>,” 2025.
- [71] A. Blankstein, S. Sen, and M. J. Freedman, “Hyperbolic caching: Flexible caching for web applications,” in *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC'17)*, 2017.
- [72] G. Einziger, R. Friedman, and B. Manes, “Tinylfu: A highly efficient cache admission policy,” *ACM Transactions on Storage (ToS)*, vol. 13, no. 4, pp. 35:1–35:31, 2017.
- [73] T. Johnson and D. E. Shasha, “2q: A low overhead high performance buffer management replacement algorithm,” in *Proceedings of 20th International Conference on Very Large Data Bases (VLDB'94)*, 1994, pp. 439–450.
- [74] E. J. O’Neil, P. E. O’Neil, and G. Weikum, “The LRU-K page replacement algorithm for database disk buffering,” in *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data (SIGMOD'93)*, 1993.
- [75] M. Abrams, C. R. Standridge, G. Abdulla, E. A. Fox, and S. M. Williams, “Removal policies in network caches for world-wide web documents,” in *Proceedings of the ACM SIGCOMM 1996 Conference on*

Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM'96), 1996.

- [76] P. Cao and S. Irani, "Cost-aware WWW proxy caching algorithms," in *Proceedings of the 1st USENIX Symposium on Internet Technologies and Systems (USITS'97)*, 1997.
- [77] Z. Song, D. S. Berger, K. Li, and W. Lloyd, "Learning relaxed belady for content distribution network caching," in *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*, 2020.
- [78] N. Beckmann, H. Chen, and A. Cidon, "LHD: improving cache hit rate by maximizing hit density," in *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*, 2018.
- [79] J. Yang, Z. Mao, Y. Yue, and K. V. Rashmi, "Gl-cache: Group-level learning for efficient and high-performance caching," in *Proceedings of the 21st USENIX Conference on File and Storage Technologies (FAST'23)*, 2023, pp. 115–134.
- [80] H. Al-Sayeh, M. A. Jibril, and K. Sattler, "Agile-ant: Self-managing distributed cache management for cost optimization of big data applications," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 17, no. 11, pp. 3151–3164, 2024.
- [81] G. Ananthanarayanan, A. Ghodsi, A. Warfield, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica, "Pacman: Coordinated memory caching for parallel jobs," in *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI'12)*, 2012, pp. 267–280.
- [82] J. Wang, B. Berg, D. S. Berger, and S. Sen, "Maximizing page-level cache hit ratios in largeweb services," *ACM SIGMETRICS Performance Evaluation Review*, vol. 46, no. 2, pp. 91–92, 2018.
- [83] C. C. Aggarwal, J. L. Wolf, and P. S. Yu, "Caching on the world wide web," *IEEE Transactions on Knowledge and Data Engineering*, vol. 11, no. 1, pp. 95–107, 1999.
- [84] X. Miao, H. Zhang, Y. Shi, X. Nie, Z. Yang, Y. Tao, and B. Cui, "HET: scaling out huge embedding model training via cache-enabled distributed framework," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 15, no. 2, pp. 312–320, 2021.
- [85] W. Zhao, D. Xie, R. Jia, Y. Qian, R. Ding, M. Sun, and P. Li, "Distributed hierarchical GPU parameter server for massive scale deep learning ads systems," in *Proceedings of Machine Learning and Systems (MLSys'20)*, 2020.
- [86] E. T. L. Omtzigt, P. Gottschling, M. Seligman, and W. Zorn, "Universal numbers library: design and implementation of a high-performance reproducible number systems library," *CoRR*, vol. abs/2012.11011, 2020.
- [87] Y. Lee, S. H. Seo, H. Choi, H. U. Sul, S. Kim, J. W. Lee, and T. J. Ham, "MERC: efficient embedding reduction on commodity hardware via sub-query memoization," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)*, 2021.
- [88] H. Kal, S. Lee, G. Ko, and W. W. Ro, "SPACE: locality-aware processing in heterogeneous memory for personalized recommendations," in *Proceedings of the 48th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA'21)*, 2021, pp. 679–691.