EVSTORE: Storage and Caching Capabilities for Scaling Embedding Tables in Deep Recommendation Systems

Daniar H. Kurniawan
University of Chicago
Chicago, IL, USA
daniar@uchicago.edu

Ruipu Wang
Beijing University of Technology
Beijing, China
rexwoodrp@gmail.com

Kahfi S. Zulkifli
Fandi A. Wiranata
Bandung Institute of Technology
Bandung, Jawa Barat, Indonesia
{sbhnkahfi,fandi.z.w}@gmail.com

John Bent
Seagate Technology
Fremont, CA, USA
john.bent@seagate.com

Ymir Vigfusson
Emory University
Atlanta, GA, USA
ymir@mathcs.emory.edu

Haryadi S. Gunawi
University of Chicago
Chicago, IL, USA
haryadi@cs.uchicago.edu

ABSTRACT

Modern recommendation systems, primarily driven by deep-learning models, depend on fast model inferences to be useful. To tackle the sparsity in the input space, particularly for categorical variables, such inferences are made by storing increasingly large embedding vector (EV) tables in memory. A core challenge is that the inference operation has an all-or-nothing property: each inference requires multiple EV table lookups, but if any memory access is slow, the whole inference request is slow. In our paper, we design, implement and evaluate EVSTORE, a 3-layer EV table lookup system that harnesses both structural regularity in inference operations and domain-specific approximations to provide optimized caching, yielding up to 23% and 27% reduction on the average and p90 latency while quadrupling throughput at 0.2% loss in accuracy. Finally, we show that at a minor cost of accuracy, EVSTORE can reduce the Deep Recommendation System (DRS) memory usage by up to 94%, yielding potentially enormous savings for these costly, pervasive systems.

CCS CONCEPTS

• **Information systems** → **Novelty in information retrieval**; • **Computer systems organization** → **n-tier architectures**; **Secondary storage organization**; **Pipeline computing**; *Real-time system architecture*; • **Computing methodologies** → *Neural networks*.

KEYWORDS

Recommendation Systems; Deep learning; Caching systems; Inference systems; Performance

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada

ACM ISBN 978-1-4503-9916-6/23/03…$15.00
https://doi.org/10.1145/3575693.3575718

ACM Reference Format:
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 '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3575693.3575718

1 INTRODUCTION

Recommendation systems are used prominently across modern online services to help people make decisions. They capture user behavior and preferences to display personalized advertisements [29, 30], rank news [10, 24], and recommend products [69]. The impact of recommendation systems on user engagement is tremendous. Recent studies show that a significant amount of content—30% of all traffic on Amazon's website, 60% of the videos on YouTube, and 75% of the viewed movies on Netflix came from suggestions made by recommendation algorithms [7, 8, 62, 74].

In the age of Deep Learning, Deep Recommendation Systems (DRSs) are widely used to deliver high-quality recommendations [30, 78], but tackling *categorical ("sparse") input features* is their Achilles' heel. Modern DRSs, such as Facebook's post recommendation systems [30], often contain hundreds or thousands of categorical features (*e.g.*, users, posts, or pages), each of which can contain millions or even tens of billions of possible categories. To make the complexity of the deep neural network (DNN) tractable, sparse categorical data is usually converted to ("dense") vectors of numbers before being fed to the model. The most popular conversion is via *embedding vector tables*, or "**EV tables**" for short (§2).

By reducing the DNN complexity, EV tables sacrifice space for faster computation, and thus require significant memory. Consequently, the space management of EV tables becomes challenging: many real-world EV tables contain billions of embedding vectors [31, 69] that require tens of TBs of memory capacity. Such DRAM-heavy architectures account for significant operational costs for DRS users measured in millions of dollars—nearly 80% of all AI-related deployment in Facebook's data centers in 2020 directly supported DRSs [30]. Additionally, industry's insatiable appetite for improved recommendation accuracy is driving the rapid growth

of EV tables in DRS. As users become more reliant on these systems, they expect higher quality recommendations that are tailored to their individual preferences. To meet this demand, recommendation systems must be able to encode richer semantic relationships, which requires larger EV tables. This has led to a tripling of EV table sizes every two years (1.5× annual growth) [16, 38].

Unfortunately, the state-of-the-art DRSs are simply not equipped to handle the exponential growth of EV table sizes. Open-source DRSs platforms like Facebook's DLRM [54] and Google's DCN [70, 71], for example, store the *full* EV tables in DRAM and lack support for responding to lookups from backend storage when memory is exhausted. This brings several downsides. When the entire memory is mostly occupied by EV tables of a specific DRS model, the server is not able to run other DRSs concurrently, potentially reducing resource utilization of the server and the overall throughput of the recommendation service. Furthermore, storing the entire EV tables in memory is costly as the price of DRAM keeps increasing, especially due to shortages in global supply [13]. A natural solution to this problem is by moving the large EV tables to the backend storage (SSDs or HDDs). There are recent publications in this space that focus on optimizing the backend storage for EV table lookups [28, 68, 72]. While existing storage solutions advance the state of the art, their adoption is limited due to the need of customized devices (*e.g.*, custom SSDs or FPGA implementations).

In this paper, *we take a different approach*: How should we revisit this problem from the context of the DRS platform itself? Can we add a novel caching layer within the DRS platform (that works on commodity storage backend)? Can the caching layer be optimized specifically for EV access patterns? To address these questions, we built EVSTORE: a novel EV table caching layer in DRS inference pipelines that exploits available DRAM and the structure of EV lookups to optimize end-to-end DRS inference latency. EVSTORE's main contributions lie in EVSTORE's 3-layer "L1-to-L3" caching design (EVCache, EVMix, and EVProx):

(L1) EVCache: We built a caching layer (EVCache) where EV tables are stored as key-values in the DRS memory and backend storage. We harness an all-or-nothing EV access property: an inference will query a set of keys to *all* of the EV tables, hence a cache miss on just *one* of the keys will make the entire inference slow. State-of-the-art cache replacement algorithms do not fit this lookup pattern. Hence, we introduce the concept of *groupability* and extend existing algorithms with "group scores" to rank keys that are likely accessed together and retain them in the cache, in turn increases the chances of getting a "perfect-hit" where all of them simultaneously can be found in memory.

(L2) EVMix: To accommodate diverse latency and accuracy tradeoffs, we delegate some space from the L1 into an "L2" segment that stores lower precision (16, 8, or 4 bits instead of 32-bit floating point) embedding values. For instance, whereas the first layer stores 32-bit floating point values (`fp32`), the second layer can store lower precisions (*e.g.*, in 16, 8, or 4 bits). We call this combination of L1 and L2 as EVMix, a *mixed-precision* caching. This brings several advantages: allowing more key-value pairs to be cached, increasing hit rates, accelerating inferences, and boosting throughput in trade for a minor loss of accuracy.

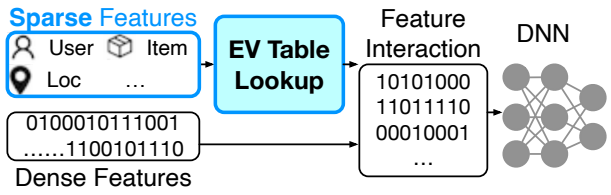

Figure 1: DRS and EV Tables (§2). EV tables are used to accurately translate the sparse categorical data into dense vectors of numbers by revealing hidden relationships between input features. These dense vectors can then be combined with other dense features before being fed into the DNN model to obtain the inference result.

(L3) EVProx: Finally, we leverage another unique characteristic of embedding values: The value for a key that is not in the cache can be replaced by a *surrogate* key whose value is "*approximately similar*" to the original key's value. We add a *key-to-key* caching layer (L3) that maps each key to a surrogate key with a similar embedding value. Furthermore, we choose surrogates that are likely to reside in the L1/L2 cache to help alleviate accesses to the backend storage. To the best of our knowledge, the closeness of embedding keys, computed using well-established statistical methods for similarity analysis [26, 48, 59], has not been previously used for DRS performance optimization.

We have fully integrated EVSTORE within Facebook (Meta)'s DLRM [54], including various implementation-level optimizations and offline supporting tools (≈9 KLOC) that are released publicly [1]. Our evaluation based on real production DRS traces shows that EVSTORE can reduce the average and p90 latency by up to 23% and 27% respectively, while increasing the throughput by 4× at only 0.2% accuracy reduction. Collectively, fully optimized EVSTORE implementation can achieve a 94% reduction of the DRS memory footprint. These memory savings correspond to hundreds of millions of dollars for a large cloud provider [47].

2 BACKGROUND AND MOTIVATION

Consider a system asked to make product recommendations related to the query "`food that kitty likes`". After processing the natural language string with standard NLP methods like tokenization and stemming [39, 67], the system is provided with a set of sparse (categorical) and dense (numerical) input features. These features include high-dimensional representations of the words in the sentence from the NLP engine, as well as supplemental information, such as user attributes and location (**Figure 1**).

Deep recommendation systems (DRS) are recommendation engines that leverage deep neural networks (DNNs). Unfortunately, sparse categorical data, in particular those resulting from processing text data, are a poor match for the DNNs due to the unwieldy space and time complexity they impose during training. Instead, input data is usually condensed before being consumed by the DNNs—sparse text data, for instance, undergoes *word embedding* into lower-dimensional vector space.

Embedding vectors (EV) are the most popular method for densifying sparse input features for the DRS, effectively translating

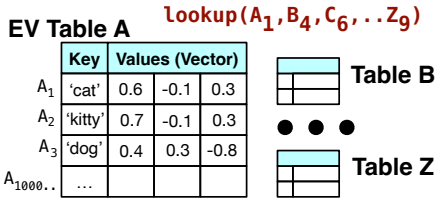

Figure 2: EV table structure and lookup (§2). An example of EV tables *A*–*Z* in a DRS. Each EV table represents the conversion for a single type of categorical feature. A lookup involves finding a key in each of the *N* tables ($N = 26$ if the DRS model has 26 categorical features which correspond to EV table *A*–*Z*). Different DRS model might use different set of EV tables.

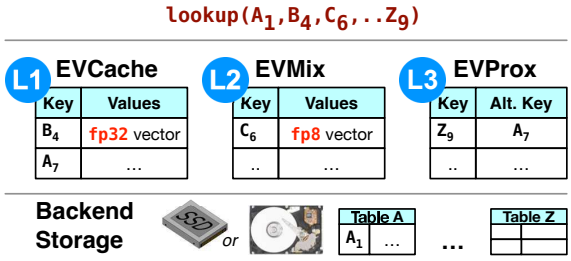

Figure 3: EVSTORE design overview (§3). EVSTORE is composed of EVCache (L1), an EV table caching layer with various cache replacement options (§3.1+§4); L2, a second caching layer which stores lower precision embedding such as `fp8` to enables EVMix, (§3.2+§5); and EVProx (L3), an embedding approximation layer that caches mapping to surrogate keys (§3.3+§6). EVSTORE is fully implemented in a Facebook PyTorch-based DRS inference pipeline (§3.4+§7). The `lookup(A1,B4,C6,..,Z9)` will lead to B_4 hit in L1, C_6 hit in L2, Z_9 "hit" in L3 as it is replaced with the value from a surrogate key A_7, and A_1 miss that will incur a disk access.

sparse categorical data into dense vectors of numbers [18]. Internally, the translation is done by means of an *EV table* in memory that simply returns the appropriate vector value, say $(0.7, -0.1, 0.3)$, corresponding to a given key, say '`kitty`', as illustrated in **Figure 2**. By reducing the dimensionality of the data, EV tables also reveal hidden relationships between inputs. For example, note that "`kitty`" and "`cat`" are practically synonyms in EV table *A* in Figure 2 because of the proximity of the corresponding embedding vectors. The DNN itself need not recognize the synonymy of "`kitty`" and "`cat`": since similar words cluster together in the embedding space, the queries "`food that kitty likes`" and "`food that cat likes`" will produce comparable results.

EV tables are crucial components of a DRS, so let us consider their structure and anatomy in more detail. Internally, each row in an EV table consists of a "key" index and a number of columns of floating point values representing the embedding vector corresponding to the key. Under NLP word embedding, for instance, the key may be a dictionary word like "`cat`". The key could also represent a more complex category, such as the hash of a compound string. The embedding vector columns are the values for latent features or *dimensions*. Each cell is typically a 32-bit floating point number (`fp32`). The cells are initialized as random values and gradually updated via backward propagation during training towards higher fidelity embedding vectors. The number of latent features is a design decision: more dimensions increase the lookup precision at the expense of larger tables.

A DRS lookup is the top-level inference query. Because each EV table represents the conversion for a single type of categorical feature, such as word-embedding within an NLP model, a single inference may involve dozens of different EV tables, each with potentially millions of rows [46, 52, 82]. In Figure 2, for example, 26 different EV tables must be consulted for a single inference. We denote DRS lookups by:

$$\texttt{lookup}(A_1, B_4, C_6, .., Z_9),$$

where the number in the subscript represents a key in the table. For instance, B_4 refers to key number 4 in Table B.

EV tables are large and growing. Today's recommendation models have enormous feature sets to capture complex user behavior and preferences [23, 24, 30, 81, 83, 84]. Each categorical feature could assume 10^7–10^{10} different possible values [31, 58, 78], implying that billions of embedding vectors are needed in practice to represent every unique feature. A billion embedding vectors (rows) with 400 dimensions (columns) [46, 52, 82] of `fp32` type (cell size) would easily occupy 1.5 TB of memory. Furthermore, industry's insatiable appetite for improved recommendation accuracy demands more rows, extra columns, and larger vectors (cells) to encode richer semantic relationships. Thus, the models are growing rapidly—the sizes are tripling every two years (1.5× annual growth), following Moore's Law [16, 38], while the underlying DRAM-hungry DRS implementations already weigh heavily in company budgets [30].

DRS pipelines are up against a scaling wall. Crucially, all trends point to the continued burgeoning of DRS system sizes. Recent projections predict that EV table sizes will imminently be dozens of TB for some companies [16], flirting with the limits of even the greatest memory capacity cloud instances available[1]. To continue scaling DRS, a different approach is required.

3 EVSTORE DESIGN OVERVIEW

We present EVSTORE, a rethinking of DRS pipelines to accommodate large EV tables. With EVSTORE, EV tables are no longer required to completely fit in memory, allowing operators to grow their DRSs or improve inference throughput by packing multiple DRS pipelines among machines without running into rigid memory size constraints of individual machines. To the best of our knowledge, EVSTORE is the first system that adds powerful caching capabilities within a real-world DRS pipeline, including various implementation-level optimizations. There are three key components to the EVSTORE design, depicted in **Figure 3**:

a. EVCache provides the first level of caching ("L1") with various cache replacement options that are specifically tailored to handle EV lookup patterns.

[1] At the time of writing, high-memory instances top out at 24 TiB (AWS), and 12TiB (Azure/Google Cloud).

b. EVMix adds support for multi-tier caching layer ("L1+L2") with mixed precisions (*e.g.*, 32, 16, 8, and 4 bit) across different layers to provide better performance.
c. EVProx accelerates lookups via a novel "L3" layer that caches approximate embedding to opportunistically replace a missing key with a surrogate key that is likely to reside in L1 or L2.

3.1 EVCache

By adding a caching layer for EV lookups to the DRS pipeline, the cache replacement policy begins to dominate the performance of the lookup workload. Cache replacement algorithms have primarily been designed for items with independent request patterns (such as key-value stores), or where accesses concern ranges of consecutive memory (such as virtual memory and storage systems). Unlike traditional caches, however, DRS lookups exhibit the aforementioned "all-or-nothing" property when accessing cached EV tables. That is, for every inference request, the key-value lookup must be done across all constituent EV tables at the same time, *e.g.* `lookup(`A_1`,`B_4`,`C_6`,..,`Z_9`)`—a cache miss for just *one* of the keys (*e.g.*, A_1) will make the entire inference slow. This uncompromising attribute stems from the neural network (NN) architecture: the output value from each EV table is a portion of the input vector into the neural network, without which the NN yields ill-defined results.

We evaluated both popular and state-of-the-art caching algorithms (LRU, LFU, ARC, CAR, Cacheus, ClockPro [19, 44, 50, 60]) against DRS workloads with the all-or-nothing property and found their performance to leave an opportunity for improvement (§4.1). We noticed that existing cache algorithms could be infused with a novel notion of *"groupability"*. That is, EV-friendly algorithms ought to consider the fact that keys are accessed as a *group* in EV lookups. In a departure from ordinary caching systems, the input into our EVCache layer involves multiple keys at once as a group, rather than just a single key. With grouped keys, the objective of our caching system is then to maximize the chance of getting a "***perfect hit***" where all of the keys are found in the cache (§4.2). We then also speak of *perfect hit rate* instead of just *hit rate* for single key lookups.

To demonstrate the flexibility of the groupability notion, we extended three popular algorithms (LFU, CAR, and ARC) into EV-LFU, EV-CAR, and EV-ARC, respectively (§4.3). These three EVCache variants have different implementations and characteristics that offer adaptability and choices in handling a variety of DRS workloads. For example, EV-CAR and EV-ARC both adapt well to EV-based and classical individual lookups in that it bolsters perfect hit rates without sacrificing the individual hit rates, whereas EV-LFU is highly optimized for DRS workloads at the expense of lower individual hit rates. Maximizing the perfect hit rate poses an interesting algorithmic question: what simple online heuristics can factor in groupability without undue computational overhead? We detail our approach in Section 4.2.

3.2 EVMix: Mixed-Precision Caching

Another family of approaches for increasing cache performance, besides improving the replacement policy, is to conduct domain-specific packing, either through lossless or lossy compression of values [14, 34, 64]. To balance EVSTORE's all-important latency goal with recommendation accuracy, we delegate some space from the L1 into an "L2" segment that stores lower precision embedding values. We call this combination of L1 and L2 as EVMix, a *mixed-precision* caching. Moreover, the two cache tiers will have different sizes and data precision but run the same cache replacement policy. Recalling that EV are stored as 32-bit floating point values (`fp32`), there is an opportunity to lower the resolution of the floating point value to 4, 8, or 16 bits—allowing the cache to keep more values in memory in exchange for a minor reduction in accuracy. For instance, whereas the first layer (L1) stores 32-bit floating point values (`fp32`), the second layer (L2) can store lower precisions (*e.g.*, in 16, 8, or 4 bits). Users can adjust the resolution and size to balance the desired accuracy and performance. EVMix uses fast coding optimizations that harness the specifics of embedding vector management, detailed in Section 5.

3.3 EVProx: Approximate Embedding

Another unique characteristic of embeddings that differentiates them from typical key-value data: embedding vectors reside in relatively smooth (high-dimensional) metric spaces with well-defined distances between vectors. Thus, building on ideas from nearest-neighbor clustering, the original value of a key may be *approximated* by the "similar" value of a nearby neighbor. That the neighbors have comparable values stem from an empirical smoothness property called the *embedding value similarity* [33]. While such clustering techniques are popular for analyzing and reducing the complexity of high-dimensional data, we are not aware of any work that exploits them explicitly for performance optimization.

Using these ideas, we propose another layer, EVProx, that allows a key-value cache miss to be replaced by a *surrogate* key whose value is likely to be cached in L1/L2, hence avoiding a lookup to the backend storage. Without this L3, if a key is not available in L1 and L2, slow disk access would be needed. Accordingly, L3 can be viewed as a *key-to-key* caching layer that maps a key to a surrogate key with a similar embedding value. For example, in **Figure 3**, the key Z_9 is a miss on L1 and L2. Before going to the disk, we check L3 and find that A_7 is the surrogate key of Z_9. Since A_7 is already stored in L1, the disk access is prevented. Furthermore, since having a key-to-key caching requires much less space compared to caching the whole embedding value, EVProx needs minimal space and will only occupy a small percentage ($\leq 5\%$) of the total cache size. Section 6 further describes the challenges of implementing the L3. For instance, for every key, how do we establish the appropriate surrogate keys? Also, which key is more likely to reside in L1/L2?

3.4 Implementation and Integration

Our final contribution is in the implementation and integration of EVSTORE in a real DRS platform, specifically the Facebook DLRM framework [54]. We explored various implementations along several dimensions including supporting various storage backends (RocksDB[3], SQLite[9], CORTX [12], and UNIX files [65]). We then embedded a new caching layer inside the DLRM through two approaches: via the tensor library and via the EVCache layer with our optimized data structures. We also migrated our Python implementation to C++ to better support mixed precision, harness multithreading, and optimize data reuse with the

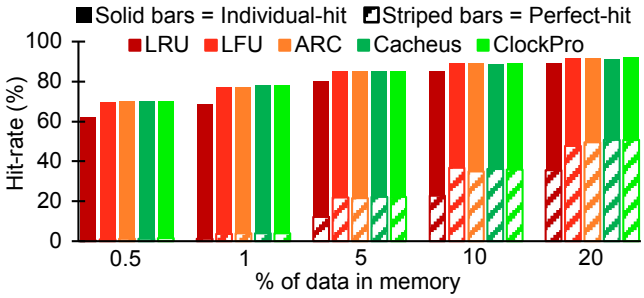

Figure 4: Individual vs. perfect hit rates (§4.1). Existing algorithms have high individual hit rates (solid bars), but relatively low perfect hit rates (striped bars) across various cache sizes.

help of dynamic memory allocation and pointer manipulations. We added ≈7 KLOC to the Facebook DLRM and ≈2 KLOC of offline tools for benchmarking EVCache algorithms and EVProx approximate embeddings.

4 EVCACHE (L1)

In this section, we evaluate the performance of different caching algorithms on EV lookup workloads (§4.1), describe how we apply our groupability principle to improve the perfect hit rates of these algorithms (§4.2), and demonstrate how the principle can be adopted across various caching policies (§4.3).

4.1 The Importance of Perfect Hits

The caching literature is replete with algorithms, from the basic policies (such as LRU, CLOCK, and LFU [25, 44, 55, 65]) to the more dynamic/adaptive variants (such as LIRS [37], CAR [19], ARC [50], ClockLIRS [37], and ClockPro [36]), and finally the machine-learning based ones (such as Cacheus [60] and LeCAR [66]). To understand how they relate to our problem domain, we evaluate the performance of these algorithms on EV lookup workloads. Recall that to serve a single inference request with N sparse features, the DRS must convert those sparse features to N dense features by doing EV lookups to N different EV tables. *Any* cache miss on one of the EV tables requires access to the backend storage (*e.g.*, SSD and HDD) which generally is orders of magnitude slower than memory access, thus slowing down the entire inference.

To quantify caching performance, we use two metrics. First, the **individual hit** rate, the typical metric used when evaluating caching algorithms, concerns the ratio of key-value lookups that are found in memory, regardless of how many embedding tables are used in a single inference. Next, the **perfect hit** rate is the ratio of how often *all* N keys (from a single inference request) are found in the memory, a scenario where no data needs to be fetched from the disk before running pass forward phase in DRS pipeline.

Figure 4 shows the results when we have $N = 26$ using the Criteo dataset [6] (details in the evaluation section). Here we only show 5 algorithms for readability. For the individual hit rate (solid bars), as expected, the algorithms can reach 60–90% hit rate (vertical axis) when the cache size is 0.5–20% of the size of all the tables (horizontal axis). *However, the perfect hit rate is significantly lower, ranging only from 1% to 50%* (the striped bars), mainly because existing algorithms do not take into account the *group*-based access pattern. Moreover, as the cache size increases, the individual hit rate tends to increase in a lower rate than the perfect hit. This demonstrates that while current algorithms may be effective at finding individual items in the cache, they are less effective at finding all items in a set.

4.2 Replacement Policy Extension

While traditional cache lookups rely on one key per lookup, EVCache operates on multiple keys for every single inference (we call them "**grouped keys**"). Fortunately, in a DRS the cardinality of the group is fixed (*e.g.*, 26 keys whose values will be supplied to a constant number of features in the neural network model). EVCache introduces the concept of "groupability" into embedding cache management by adding a scoring metric `groupScore` for every key in the cache. Keys with high scores will remain in the cache while those with lower scores will likely be evicted. Therefore, we need a caching algorithm that prioritizes embeddings with high group scores over the ones with low group scores, hence increasing the perfect hit. Below we describe how EVCache works from the perspectives of four fundamental caching operations: cache lookup, state update, insertion, and eviction.

Cache lookup: An inference will trigger a grouped-keys lookup, *e.g.* `lookup(`A_1`,`B_4`,..,`Z_9`)`. EVCache will calculate the total cache hits among the 26 individual key lookups. Let's suppose, 20 out of the 26 are cache hits. EVCache will memorize the group score of 20 and use it in the next caching operations.

Cache state update: For every key with a cache hit, *e.g.* B_4, its value stored in the cache will be read and prepared to be supplied to the neural network. EVCache will then update the B_4's group score in the cache with the `max` of the current and the new score. For example, if key B_4 is a hit and its current group score is 15, then EVCache will update B_4's score to 20 (the memorized score). The detail on the "`max`-based" group scoring and other scoring methods are covered at the end of this section.

Cache insertion: For every key with a cache miss, EVCache looks up the value from the backend storage and inserts the key-value to the cache with a score value of 20 (the memorized score). If the cache is full, EVCache needs to evict some key-values from the cache, *even if* they have higher scores than the scores of the to-be-inserted keys. This is because decades of caching research have shown that recency (introduced by the newly inserted keys) is an important factor in caching performance [25, 36, 37, 55].

Cache eviction: The key-values in the cache are *sorted based on the group scores*. EVCache by default evicts keys with the lowest group scores. Note that within one group score, there could be any arbitrary number of keys. Since eviction will happen frequently, we must use the appropriate data structure to avoid any bottleneck and minimize the overhead. Thus, we pick an `unordered_set` data structure to store those keys efficiently. Specifically, there is one `unordered_set` per group score. This data structure gives minimum overhead during eviction because it has an O(1) worst-case runtime.

Summary: We keep our "`max`-based" group scoring method relatively simple for two reasons: it is computationally cheap while

giving the best perfect-hit rate improvement compared to other scoring methods we tried, including average, sum, median, static, and dynamic-based ones. Score calculation based on average, sum, and median will not only increase the metadata size but also the computational cost. We also tried an incremental update with a static increase of x (*e.g.*, $x = 1$) but struggled to define an optimal value of x in a dynamic workload. Furthermore, since every newly inserted key has the same value of x, highly groupable keys may readily be evicted soon after they are inserted. Defining a dynamic value of x likely requires a more complex implementation—some of the approaches we tried decreased the perfect hit rate by 50% despite being 30× slower.

Overall, our groupability concept targets the relationships between cached embedding data that were requested at the same time. Harnessing relationships between items have been extensively explored in the cache literature, ranging from long-standing observations about the relative recency of requested data [22, 25], tenuring highly-frequent items [44], exploiting other data attributes [20–22], and even learning request histories through non-linear machine-learning approaches [63]. To the best of our knowledge, the systems literature has not before considered caches where requests arrive together as a set of items. Under such a model, the relationship between items in the same set adds a dimension to the analysis that transcends the traditional dynamical notions of frequency and recency that abound in the cache literature. Our intuition is to strongly inform cache eviction by providing *fate sharing of friends* through a scoring function–to have items that are accessed together reinforce, or abate, the scores of one another. The next section shows how we integrated the scoring extension (as part of the groupability concept) into popular cache replacement policies.

4.3 EVCache Variants

To show generality, we implemented our extension to three popular (base) algorithms: LFU [44], CAR [19], and ARC [50]. Our three EVCache variants (**EV-LFU**, **EV-CAR**, and **EV-ARC**) have different implementations and characteristics. The main differentiator is how the base algorithms could accommodate group scores into their data positioning mechanism which also influences their eviction policy. In the interest of space, we will not describe the base algorithms in detail (interested readers can refer to our code [1]).

1. EV-LFU: This algorithm is the modified version of the Least Frequently Used (LFU) cache replacement policy. We replace the default frequency counter in LFU [44] with a group score. This means that upon a cache miss, EV-LFU will evict the cached item with the lowest group score. If there are multiple items with the same group score, EV-LFU will evict the least recently inserted item. The group score used in EV-LFU has a maximum value (*e.g.*, 26 in our main experiment), which ensures that the scores of items in the cache do not become too large over time. When most of the cached items reach the maximum score, recently cached keys with lower group scores start to face higher eviction pressure. To avoid class imbalance, EV-LFU implements a flushing mechanism with a tunable knob. Specifically, if the number of `maxScoreKey` (key with maximum group score) is higher than the "`maxScoreKeyCapacity`" (*e.g.*, 20%), EV-LFU will reduce the population of the `maxScoreKey` by X% (where X can be adjusted dynamically).

Furthermore, both LFU and EV-LFU are categorized as *stack algorithms* which makes them free of Belady anomaly. Specifically, in a stack algorithm, the items evicted by a larger cache will be a subset of those evicted by a smaller cache if both were to see the same request sequence—a property known as cache inclusion—independently of those cache sizes. Conveniently, the hit rate of stack algorithms increases monotonically with cache size [61], which provides a further degree of robustness to EV-LFU in practical settings.

2. EV-ARC: ARC [50] is an adaptive algorithm designed to recognize access recency and frequency by dividing the cache into two lists: `R-list` (recency-based) and `F-list` (frequency-based). `R-list` holds items accessed once while `F-list` keeps items accessed more than once since admission. To dynamically adjust the size of the probationary segment (`R-list`) and the protected segment (`F-list`), ARC uses information about recently evicted cache items (stored as `R-ghost` and `F-ghost` lists). For EV-ARC, we add group score as a metadata to every cached item. We then modify the `F-list` to use EV-LFU's counting, eviction policy, and flushing mechanisms. The difference is that cached items flushed from the `F-list` will be transferred to the tail of the `R-list`. The ghost cache size will be adjusted so that the number of the cached pages in `R-list` and `R-ghost` is equal to the number of the cached page in the `F-list` and `F-ghost`.

3. EV-CAR: CAR [19] is an algorithm that combines ARC and the popular CLOCK second-chance algorithm. For EV-CAR, we modify the reference bit, R variable, so that it will store the group score instead of just storing 0 or 1. During the eviction phase, the CLOCK hand will only evict the cached item that has $R = 0$, otherwise, it will be challenged by the incoming key. If the incoming key's group score is larger than the current item (pointed by the CLOCK hand), EV-CAR will not evict that item, but give a second chance to the current item by setting its R to 0. EV-CAR also modifies the CLOCK mechanism by introducing a "progressive decrement" method which allows the R value to be decreased regardless of the group score of that item. This method guarantees the CLOCK hand to find an item to evict within a single rotation ($O(n)$ complexity where n is the number of items in the cache). In a cache hit, EV-CAR applies max-based scoring (§4.2) which replaces the current group score if the new score is bigger.

5 EVMIX (L2)

To make our caching layer more versatile in addressing various latency and accuracy tradeoffs, we introduce EVMix, a multi-tier mixed-precision EV caching system. In this section, we first describe the advantages of EVMix (§5.1), its design (§5.2), and the bit coding optimizations (§5.3).

5.1 Advantages of Mixed Precisions

An embedding vector is stored as floating point values. In most systems such as DLRM [54] and DCN [70], the default precision is `fp32` (32 bits). However, EVMix caching layer can store those values in a lower precision format such as in 16, 8, or even 4 bits depending on the target accuracy.

EVMix can bring several advantages. ***(a)*** *Faster inference latency*. By accessing smaller bit representations, we can improve

the average EV lookup latency by 15%, which is significant because EV lookup can cover 40% of the end-to-end inference latency. ***(b) More cached items and higher cache hits***. With lower precisions, we can cache more embeddings (*e.g.*, 8x more cached items when the 32 bit EV is converted to 4 bit), and by implication, the cache hits will be higher, which in turn increases the throughput of the caching layer. ***(c) Configurability via multiple layers***. With multi-tier caching, one can adjust the size and the precision of the first level cache and the second level cache based on the latency-accuracy tradeoffs to make caching more versatile. (more in the evaluation section).

5.2 Multi-Tier, Mixed-Precision Design

As shown earlier in Figure 3, EVMix is the combination of L1 and L2 which collectively forms a mixed-precision caching. Each tier runs the same cache replacement policy. L1 stores high or medium precision data (*e.g.*, 32 or 16 bit) and L2 stores lower precision data relative to L1's. (*e.g.*, 4 bit). Users can adjust the precision of L1 and L2 and their sizes based on the performance-accuracy tradeoffs. The size proportion of L1/L2 is fully adjustable. If L1 and L2 have the same memory size, the L2 can carry at least 2× more items (due to the lower precision storage). Upon a cache miss on L1, we try to get a lower-precision data from L2. If we also get a miss in L2, we will fetch the raw data from the backend storage and put their representations to either L1 or L2 based on the group score. To minimize the accuracy loss associated with using EVMix, the popular items are stored in L1 while the less popular ones are packed in L2. Our L1/L2 placement algorithm also ensures that the items are not redundantly stored.

Furthermore, to maximize performance, we implement EVMix in C++ which utilizes multithreading capabilities to parallelize any atomic operations in both layers. To simplify the logic and to reduce the context switching, we design the thread organization in such a way that the task for L2's threads is triggered and managed by L1's thread. In addition, we only implement event-driven paradigm on specific tasks that require heavy I/O and computation such as reading from files and binary decoding operations. Finally, we utilize a confined memory sharing to capture the results from all threads concurrently with minimum blocking.

5.3 Bit Coding Optimization

As part of the process above, EVMix stores the embedding data in an encoded format (4, 8, 16, or 32 bit) and continuously decodes the cached data on every cache hit. The decoded data will be fed to the neural network model in the subsequent phase of the DRS pipeline. To further improve the performance of EVMix, the decoding process must be optimized, especially for the 16, 8, and 4 bit format since there is no default (standardized) floating-point binary format for them.

As EVSTORE is built specifically for caching embedding vector data, it exploits the fact that the values of these vectors range only from -1 to 1, rather than an arbitrary range of values. Moreover, the typical value distribution is a Gaussian bell-shaped curve where the occurrence/frequency is most highly concentrated near 0. Therefore, to make the most efficient use of each bit, we design the coding procedure to better represent this "dense region" of values. We design the coding procedure for simplicity to ensure that decoding remains computationally cheap and does not become a bottleneck in our caching systems.

The scheme works as follows. **(a) 16 bit:** We store the value as an unsigned short. The mapping is straightforward, the smallest-positive EV value will be mapped to 0, while the biggest-positive EV value to 65534. We utilize the last digit as our sign bit to cheaply differentiate the positive and negative embedding. Specifically, if the last digit is odd, the value is considered negative, and if the last digit is even, the value is considered positive. The decoding phase will convert each value into a corresponding floating-point value proportionally. **(b) 8 bit:** In this case, we can only store values ranging from 0 to 255. Similar to the 16 bit, we map the embedding value linearly. The -1 is mapped to 0; the +1 is mapped to 254; and, everything that falls in between will be mapped proportionally (*e.g.*, 0.23 is mapped to 156). As a result, we use 255 values out of 256 which consists of 127 values covering the negative EV, another 127 covering the positive EV, and 1 value that is mapped into 0. **(c) 4 bit:** Although 4 bit can represent 16 values, but we only use 15 values (7 positives, 7 negatives, and a zero mapped value) to cover the EV range. Most of the value mappings are focused near 0. Specifically, we pick -0.0625 to 0.0625 as the dense region range in a manner similar to Posit's [57] 4-bit mapping. Overall, our encoding mechanism only uses static dictionary mapping and basic operators (`XOR` and `mod`) which result in a negligible (<1%) CPU overhead.

We further explored Posits library (C++) which is specifically designed to encode embedding values and quantize machine learning weights for lower precision. Despite having a well-researched encoding design that better preserves near-zero values, the library induces costly overhead due to its custom binary operations—compared to our encoding design, the Posit library is 3× slower. Given that the decoding operation will be done on every single value retrieval, we decided to use our simple encoding design as described above.

6 EVPROX (L3)

Recall from §3.3 that our caching capabilities are built from the unique characteristics of EV lookup workloads. In EV lookup workloads, a value of a key can be replaced by a *surrogate* key's value that is "*approximately similar*" to the original key's value. The embedding value similarity [33] is calculated through cosine and Euclidean vector distances [26, 48, 59]. These well-established statistical methods are popular for analyzing and reducing the complexity of high-dimensional data. However, we are not aware of works leveraging them explicitly for performance optimization. Thus, we adopt the approximate embedding concept in our last caching layer, EVProx, allowing a key-value cache miss to be replaced by another similar (and popular) surrogate key whose value is likely to reside in L1/L2, thus preventing a lookup to the backend storage. Furthermore, we populate the L3 with the downgraded keys from L1/L2 in order to better retain the warm keys in the cache.

Our design ensures heavy non-blocking tasks, especially I/O, are conducted in parallel at massive performance savings. When inserting a new key to L3, we enqueue the incoming keys and batch insertions into the L3. L3 uses dedicated I/O threads to fetch all missing

values in parallel. Once all key mappings data are in memory, they are inserted to the L3 sequentially. To best prolong hot items in the L3, we add a reference (*R*) bit to every cached item in L3 that is similar to CLOCK policy's implementation of the second-chance eviction mechanism (§4.3).

6.1 L3 Dataflow

Looking at Figure 3, suppose we perform `lookup(`A_1`,`B_4`,`Z_9`)` and Z_9 is the only key not being cached in L1/L2. Before adding L3, we need to read Z_9 and its value from the disk. Now we consult L3, a *key-to-key* caching layer that will tell us whether there is another key (say A_7) that has a "similar" embedding value to Z_9. The A_7 is called a *surrogate* key to Z_9, and may come from different embedding table. Note that there can be many other keys whose values are similar to the missing key Z_9. In that case, L3 will pick the most popular key (as a surrogate) measured based on its group score. In this example, L3 keeps a mapping between C_6–A_7 because of A_7's high group score, hence increasing the likelihood that A_7 will be found in L1/L2.

Remark that L3 is a special key-to-key caching layer that does not cache any value (it only caches the keys). Thus, if a lookup of Z_9 is a hit in the L3 layer, we can retrieve the surrogate key (in this case, A_7). If A_7 is found in L1/L2, an alternative value is found and no disk access is needed. However, if Z_9 lookup is a miss in L3 or A_7 is also missing from L1/L2, then we will fetch Z_9 and its value from the backend storage and store it in either L1 or L2 as explained in §5.2 about multi-tier and mixed-precision design.

6.2 Preprocessing Surrogate Keys

In designing EVCache, we encountered the following challenges: For every key, how do we determine what other keys are "similar" within the embedding space? Further, among the multiple potentially similar keys, how do we decide which one is most likely to exist in L1 and L2 cache? Finally, how and when should we populate the L3 cache? To answer these questions, we build the key-to-key mapping in an offline preprocessing manner in the following way. Note that we assume throughout that the embedding table remains static during the inference phase.

To perform similarity analysis, we adopt the statistical measures of Euclidean and cosine distances [26, 48, 59] that define similarity in terms of vector-distances [53]. This similarity analysis can be done once and the result can be reused. At the end of this stage, every key in the embedding table has a list of N most-similar neighboring keys (in our setup, the N=10). To produce the L3 key-to-key mapping, we simply pick the most popular key among the top-10 keys. To measure the popularity, we consider the historical accesses and record the access frequency of every key. By the end, supposing there are 1 million keys in the embedding table, then there is a mapping of 1 million keys to another key that is most similar and frequently accessed. Next, those mapping will be stored as a file which will make it easy to perform an online update without any shutdown. The workload type and the size of L1/L2/L3 will greatly affect the remapping frequency. If the popularity ranking is quite stable/static throughout the workload, the remapping can be avoided. In general, the remapping should be done when L3 hit rate drops significantly. The analysis of optimum remapping decisions is out of our scope. It can be studied further in future works. Finally, given that all of these tasks are done in the background, they will not introduce any bottleneck and latency overhead.

7 IMPLEMENTATION

EVSTORE is built within the popular Facebook PyTorch-based DLRM framework [54] that supports both recommendation model training and inference. The EVSTORE implementation is ≈9k LOC (≈4k LOC in C++, ≈4k LOC in Python/Bash scripts, ≈1k LOC in Java). The source code of EVSTORE, including various experiment and deployment setups, is publicly available on our GitHub repository [1]. We believe EVSTORE is the first system to support substantial caching capabilities for the EV Table lookups in this DRS framework. The details of each implementation component are explained below.

Storage backend: We extend the DLRM code to include a custom EV lookup from various key-value storage systems (RocksDB, SQLite, CORTX) and Unix files. This extension is written in Python and is approximately 2KLOC, with the majority of the code being part of the embedding-storage library. The data are stored as a stream of binary values which consists of floating point arrays. To read a specific EV value from a file, we compute the offset of the data using its key, then use seek() to directly jump to the beginning of the bytestream. The data can then be fetched from the file using either a memory map (mmap) or direct IO. Additionally, the data is sent to PyTorch as a bytestream, which eliminates the need for serialization and reduces overhead. We added a module in PyTorch to convert the bytestream into a Tensor format.

Caching layer (Python code in DLRM): The next question is where to implement L1. We first built it inside the storage backends mentioned above, but later realized that the performance could be further improved if it was embedded inside the DRS platform. To find the best place to integrate our caching layer, we must first understand how DRS systems, such as Facebook DLRM, handle the sparse-to-dense conversion. In Facebook DLRM, before the pass forward phase in the inference pipeline, by default the sparse-to-dense feature transformation reads embedding data via the tensor library. Thus, we implement L1's data structure, which mainly utilizes set and hashmap data structures, to replace the default tensor lookup. Turns out, our own choice of data structures is much faster as it is a "thinner" layer compared to the complex tensor library.

Optimized layer (in C++) for EVMix: As we support mixed precisions in L1+L2, we learned that a C++ implementation is easier to manage and optimize, especially for bitwise operations. Furthermore, most of the arrays in the implementation are stored in a plain pointer-to-pointer structure, which has better CPU efficiency than built-in vector data structures.

In the mixed-precision experiment, it is necessary to encode and decode an unusual size of floating-point data, such as 4, 8, and 16 bit values. By default, C++ aligns floating-point data at 32-bit boundaries, so we used `ushort` and `uchar` to store 16 and 8-bit precision data, respectively. When it comes to storing 4-bit data, it is not possible to use the `ushort` or `uchar` data types, as these can only store values up to 16 and 8 bits, respectively. In order to store 4-bit data, we take advantage of the fact that two 4-bit values can be

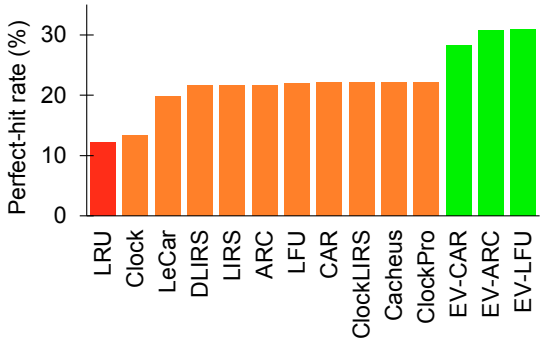

Figure 5: Exp. #1 (§8.2): Perfect hit rates across caching algorithms. EVCache algorithms (EV-CAR, EV-ARC, EV-LFU) have the highest perfect hit rate across all caching algorithms.

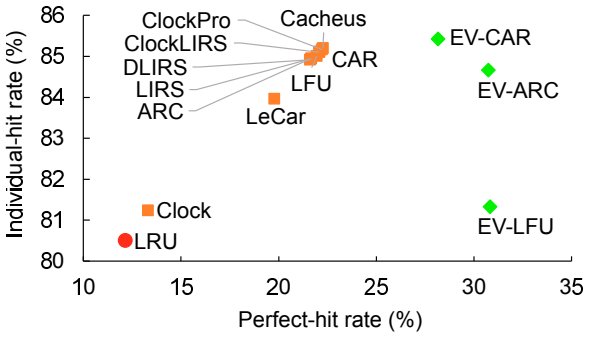

Figure 6: Exp. #1 (§8.2): Individual and perfect hit rates across algorithms. EV-LFU achieves higher perfect hit rate by sacrificing on individual hits.

packed into a single byte of `uchar` data. This allows us to store and manipulate 4-bit data efficiently, without wasting any bit spaces.

Managing concurrent accesses to L1/L2 required the use of multithreading to minimize the overhead of context-switching and locking. To do this, we stored the results in thread-specific memory regions, which allowed us to avoid interference between threads. Additionally, we explored several interfaces to facilitate the data transfer between DRS and the C++ caching layer, including socket [75] and ctypes [42], which we will evaluate later.

Offline tools: Besides changes to the DLRM platform, we also implemented two offline tools, for cache algorithm benchmarking and approximate embedding (EVProx) preparation. The former is written in Java and built on top of the Cache2K simulator [11]. In this platform, we prototyped EVCache algorithms and all our baseline algorithms including LRU, LFU, LIRS, ARC, CAR, and ClockPro. For EVProx preprocessing, we developed an embedding-similarity analysis framework, chiefly written in Python.

8 EVALUATION

To evaluate EVSTORE performance, we subjected it to numerous experiments to determine the end-to-end performance while conducting microbenchmarks over multiple dimensions, such as varying the cache algorithms, cache sizes, number of EV tables, workloads, and the use of EVMix + EVProx. We structure our evaluation as a sequence of experimental questions.

8.1 Experimental Environment and Setup

The DRS inference pipeline: (1) A user visits a webpage that has an advertisement managed by Criteo. (2) When the user interacts with the ads, it will trigger a request sent to the Criteo's server that contains all info about the user, the ads, and the webpage that is currently visited. (3) Once the inference request arrives, the server will take the sparse features, look up the EV tables, convert them to dense features and feed them to the DNN model. (4) By default, each lookup is for 26 keys to 26 tables. (5) With EVSTORE, if a key-value is not in the cache, the DRS pipeline will fetch the data from the raw files in the backend storage. (6) Finally, the inference result will influence the personalized advertisement of the user when they open another webpage managed by Criteo.

Datasets/workloads: We primarily use the **Criteo CTR** (Click-Through Rate) datasets, the largest open-sourced CTR dataset (up to 1 TiB in size) that could simulate EV lookups at scale. There are two CTR datasets released by Criteo, the 1TB data (Criteo-Terabyte) [4] and the Kaggle version (Criteo-Kaggle) [6]. It contains feature values and clicks feedback for millions of display ads. There are 13 dense integer features and 26 sparse categorical features (hence **26 EV tables**). All EV tables have the same embedding dimensions of 36. There are a total of 156 billion total (dense) feature values and over 800 million unique attribute values. In addition, we also use **Avazu**'s CTR dataset [5].

Default values: We omit redundant lines and numbers on some of the graphs for improved readability. These are our default values (unless otherwise noted): cache size of 5% of the total working set (the total size of all tables), the Criteo-Kaggle dataset [6] as the workload, and fp32 as the precision of the embedding values. Latency is measured in average latency in milliseconds.

Machine specification: We use Chameleon cloud's `gpu_rtx6000` and `gpu_v100` nodes [2, 41] which have Intel Xeon Gold CPU @2.60 GHz and 240 GiB Samsung SSD SM863a Series. We limit the DRAM using Linux `cgroup` tools to be small enough such that the DRS essential functions could run, but not big enough to store all the EV tables. When evaluating cache size smaller than the available DRAM, we flush the Operating System (OS) page cache every 0.25 ms to avoid any EV tables being cached by the OS. The method has been thoroughly tested to ensure there is no OS cache leak.

8.2 EVCache

We begin with experiments on the first layer of the cache.

Experiment #1: How much does the EVCache algorithm affect perfect hit rates? **Figure 5** shows that EVCache (EV-∗) algorithm extensions improve upon state-of-the-art algorithms such as LRU [25], CLOCK [55], LeCar [66], LIRS [37], ARC [50], LFU [44], CAR [19], ClockLIRS [37], Cacheus [60], and ClockPro [36]. The perfect hit rates are increased by up to 18%, lending support to the need for groupability for EV-based caches. **Figure 6** breaks the result down further to compare the perfect and individual hit rates (as defined in Section 4.1). Here, EV-CAR and EV-ARC both improve the perfect hit rates without compromising on individual hit rates,

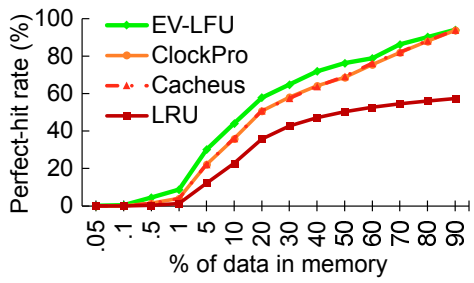

Figure 7: Exp. #2 (§8.2): Perfect hit rates across various cache sizes. Our EV-LFU has the highest perfect hit rate compared to other representative algorithms across various sizes.

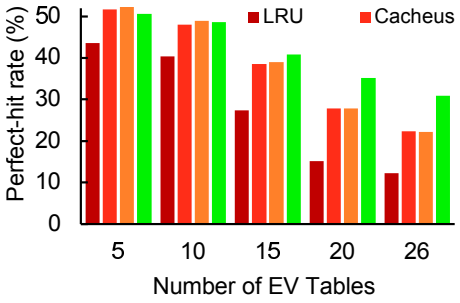

Figure 8: Exp. #3 (§8.2): Perfect hit rates on different number of EV tables. EV-LFU shows steeper benefit as the number of EV tables grows (*e.g.*, 5 to 26).

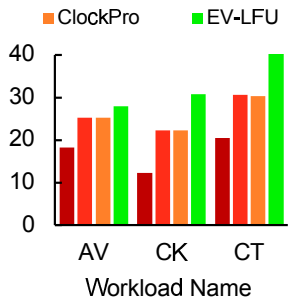

Figure 9: Exp. #4 (§8.2): Perfect hit rates across various datasets. EV-LFU has the best perfect hit rate

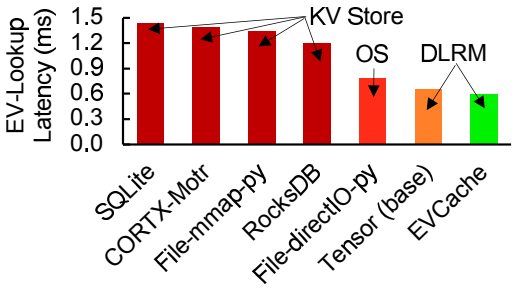

Figure 10: Exp. #5 (§8.2: The most efficient place to implement the caching layer. An optimum place to deploy EVCache is inside the DLRM framework (*e.g.*, PyTorch) using our own data structures as opposed to using PyTorch tensor library or inside the OS or an external dataabase/KV store.

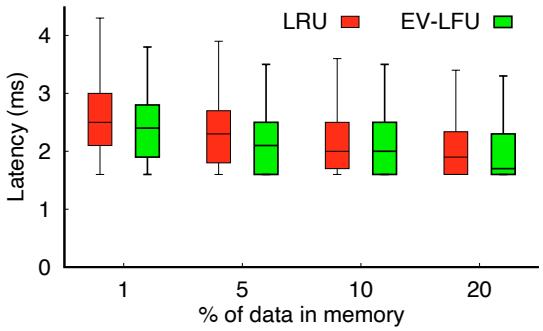

Figure 11: Exp. #7 (§8.2): End-to-end DRS inference latency on various cache sizes. Each bar uses the 1^{st}, 25^{th}, 50^{th}, 75^{th}, and 99^{th} percentiles. Our EV-LFU delivers lower latency compared to the LRU implementation in real DRS platform.

suggesting that they can be used as a general caching algorithm too. In contrast, EV-LFU increases the perfect hit rate while sacrificing the individual hit rate for each of the tables (which is acceptable since the perfect hit rate is more significant for DRS).

Experiment #2: How does EVCache perform across various cache sizes? In **Figure 7**, we vary the cache size from 0.05% to 90% of the total working set (horizontal axis). To reduce clutter, we show four representative algorithms (LRU as a basic algorithm, ClockPro as an adaptive one, Cacheus as an ML-based algorithm, and EV-LFU as EV-Cache variant). Here, the EVCache (specifically EV-LFU) outperforms others across all cache sizes. Compared to LRU, EV-LFU significantly increases the perfect hit rate by up to 35% while surpassing both Cacheus and ClockPro by up to 10%.

Experiment #3: How does the number of EV tables affect performance? Figure 8 shows that the perfect hit rate improves with more EV tables (horizontal axis) when using EV-LFU. As expected, traditional algorithms, being agnostic to relationships between EV tables, struggle to achieve a high perfect hit rate when the number of EV tables grows.

Experiment #4: How does EVCache perform across various datasets? Figure 9 compares the four representative algorithms across three different datasets. We find that our algorithm extensions improve upon other algorithms across all the datasets: Avazu (AV) [5], Criteo-Kaggle (CK) [6], and Criteo-Terabyte (CT) [4].

Experiment #5: Which layer is the best to implement EVCache? When implementing EVCache on Facebook DLRM (in this case inside PyTorch), we tried various storage backends, including key-value (KV) stores (such as SQLite, CORTX-Motr, and RocksDB) and UNIX files via `mmap` and `read/write` APIs. By default, the EV tables in DLRM are stored as "tensor" data structure. However, we implement our own data structure of choice (set and hashmap), as part of EVCache package, to be compared against the default DLRM's tensor. In this experiment, we put all EV tables in the memory, simply to measure the pure cache lookup latency as if we have enough memory to cache all of the EV tables. For key-value stores, we cache the tables in their own caching layers. For UNIX files, we depend on the OS cache. **Figure 10** shows that relying on external caching layers in KV stores or OS cache do not give the best latency compared to adding our own caching layer inside the DLRM. Furthermore, by implementing our own thin caching layer, we get better performance than using the default PyTorch tensor.

Experiment #6: What EVCache algorithm should be implemented? After deciding the best place for the caching layer, we need to decide which algorithm to implement (EV-CAR, EV-ARC, or EV-LFU). For this, we need to port our implementation from the cache simulator to the Facebook DLRM framework. Among them, EV-CAR gives the smallest perfect hit rate, and between EV-ARC

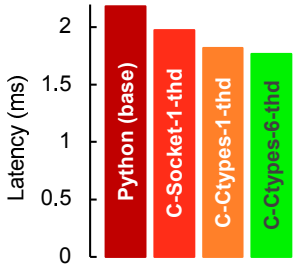

Figure 12: Exp. #8 (§8.3): Implementation variants of EVMix. Python-based implementation improved by a 6-threads C version with Ctypes.

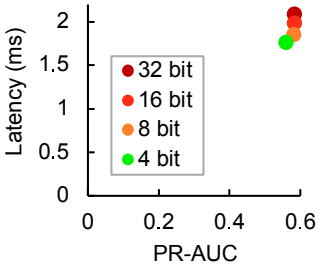

Figure 13: Exp. #9 (§8.3): Trade-off between latency and accuracy. Reducing the precision from 32 to 4 bits decreases the accuracy only slightly while greatly improving the latency.

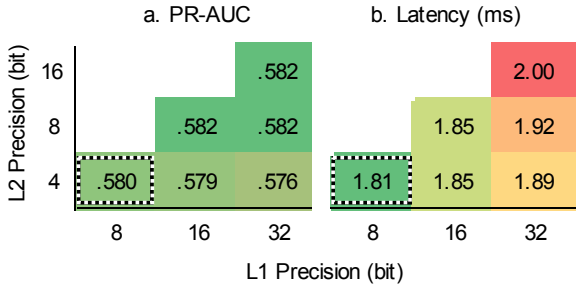

Figure 14: Exp. #10 (§8.3): Trade-off between latency and accuracy across various L1/L2 mixed-precision caches. We vary the L1 precision (horizontal axis) and L2 precision (vertical axis) and report the resulting accuracy (left figure) and end-to-end latency (right figure).

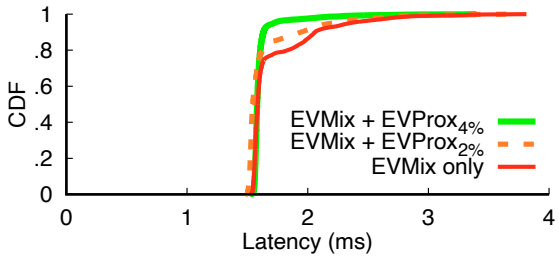

Figure 15: Exp. #11 (§8.3): Tail latency improvement with EVProx. Compared to standalone EVMix, adding L3 (EVProx) layer reduces the 95^{th} and 99^{th} latency by 27% and 22%.

and EV-LFU, they provide comparable performance in small cache sizes but EV-LFU is slightly better at higher (≥ 50%) cache sizes. In our cache simulator, Cacheus, ClockPro, EV-ARC, and EV-LFU are written in 800, 430, 270, and 130 LOC respectively. Thus, EV-LFU is more straightforward to implement by having simpler/less code compared to other algorithms. For this reason, we decided to port EV-LFU to the DLRM.

Experiment #7: How much does EVCache affect the end-to-end inference latency? At this point, in Facebook DLRM, we implemented LRU (as a baseline) and EV-LFU (as a representative of EVCache algorithms). We choose LRU as our baseline because it has the fastest lookup and the most implemented policy in the production systems. While there are more complex policies such as CAR, LIRS, CLOCKPro, etc., but they are up to 2x slower than LRU and have higher metadata space overhead. Here is the end-to-end inference latency break down: initialization (20%), EV lookup (40%), and the DNN forward propagation (40%). **Figure 11** shows a whisker plot comparing the baseline LRU vs. EV-LFU. The Python-based EV-LFU implementation delivers lower latency.

8.3 EVMix and EVProx

Next, we evaluate EVMix and EVProx layers of EVSTORE.

Experiment #8: What implementation architecture best supports EVMix? Figure 12 shows various implementation efforts we performed in re-architecting our caching layer in PyTorch and the resulting end-to-end latency. Originally, we implemented our caching data structure in Python. However, Python only supports `fp32` precision, thus we adopted a C implementation to enable storing data in lower resolution (*e.g.*, 16, 8, and 4 bits). "C-socket" refers to the C implementation that uses sockets for DLRM data transfer, "C-Ctypes" as the C implementation that uses Ctypes binding to connect our C caching to DLRM, and "*-*N*-thd" implies the number of threads being implemented to reduce cache contention §5.2. Based on our experiment, the "C-Ctypes-6-thd" delivers the best performance compared to other implementation choices.

Experiment #9: What are the latency-accuracy tradeoffs in floating point resolution (32 to 4 bits)? After we finalized our C-Ctypes-6-thd implementation, we can now evaluate the accuracy/latency tradeoffs when using lower precisions. **Figure 13** shows that reducing the precision from 32 bit to 4 bit speeds up the end-to-end latency (vertical axis) by 15% and only decreases the "PR-AUC" (horizontal axis) only by 2%. We use "PR-AUC" (Area Under the Precision-Recall Curve) to evaluate the performance of our classifier to counteract label imbalance such as in our Criteo dataset, as is standard practice [27, 35]. Intuitively, PR-AUC measures the extent to which a classifier correctly identifies all positive labels without mistaking too many others as positive.

Experiment #10: How does latency trade off against accuracy in mixed-precision L1+L2 caches? In this experiment, we divide the total cache size to L1 and L2 equally (*i.e.*, 50-50). **Figure 14** shows the accuracy (the cell content of Figure 14a) and average end-to-end latency (in Figure 14b) of EVMix as we change the embedding precision in the L1 tier (horizontal axis) and the L2 tier (vertical axis). For example, if we move from 32-bit L1 and 16-bit L2 to a 32-bit L1 and 4-bit L2 (the top-right and bottom-right corners), we improve the average latency from 2 ms to 1.89 ms and reduce the accuracy slightly from 0.582 (best case) to 0.576. Combining 8-bit L1 and 4-bit L2 gives us the best EVMix result as marked by the dotted rectangles where we reduced the latency by 10% with only 0.2% loss of accuracy.

Experiment #11: How much is the tail latency improvement with L3 (EVProx)? In **Figure 15**, we show the latency CDF of EVProx variants compared to EVMix. The "EVMix + $\text{EVProx}_{4\%}$" gives the best latency CDF in which we dedicate 4% of the cache size for L3 (EVProx) key-to-key mapping and split the rest for L1

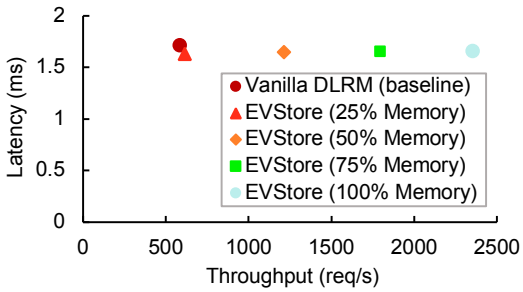

Figure 16: Exp. #12 (§8.4): EVSTORE enables multi-DRS deployment in a single node. EVSTORE's scale-out deployment quadrupled the throughput while retaining the low latency.

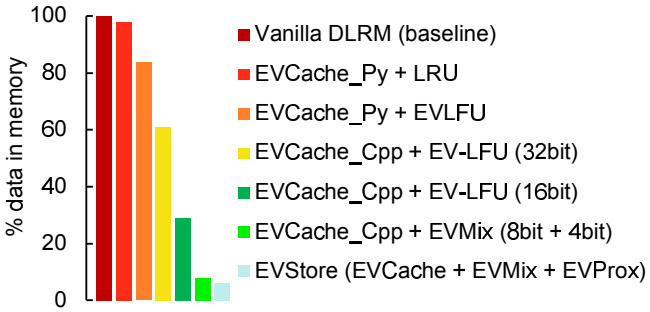

Figure 17: Exp. #13 (§8.4): Basic to fully optimized EVSTORE. The y-axis shows the minimum memory footprint to satisfy SLA target of 2 ms avg. end-to-end inference latency. Fully optimized EVSTORE implementation reduces 94% of the memory footprint compared to Facebook's vanilla DLRM.

and L2. Compared to the pure EVMix, adding the "EVProx$_{4\%}$" successfully reduces the 95^{th} and 99^{th} tail latency by 27% and 22% respectively. This experiment is conducted on 20% cache size.

8.4 Putting It All Together

Experiment #12: Does packing multiple DRSs on a machine improve throughput? As EVSTORE removes the memory requirement, in **Figure 16**, we show that we can concurrently run 4 DRSs on one machine (limited by the number of 4 GPUs in Chameleon's `gpu_v100` node) by giving 25% of the memory space to each DRS. As a result, we quadrupled the throughput (inferences/second) of the DRS. The figure also shows our final EVSTORE implementation improves the latency compared to Facebook's vanilla DLRM.

Experiment #13: Can we reduce the memory footprint of the DRS service while meeting typical SLAs? **Figure 17** shows that to meet an SLA of 2 ms average inference latency, the vanilla DLRM will require 100% of the data to be present in memory. In contrast, EVSTORE's most optimum implementation with all features enabled (rightmost bar) needs only 6% of data to be in memory, which is a 94% reduction of memory requirement in trade for the 0.2% accuracy drop. Finally, the middle bars show how the range of EVSTORE optimizations and features demand 30% to 80% of the data to be in memory. These results demonstrate the effectiveness of EVSTORE in reducing the memory footprint of the DRS service, while still meeting typical SLAs.

9 RELATED WORK

In addition to the studies surveyed throughout the paper, there are some recent publications on optimizing DRAM cache and GPU-resident cache utilization during DRS training [51, 56, 73, 76, 79, 80]. The focus, however, is on training rather than inference, and ignores systems-level nuances of cache policy and optimizations.

Another nascent body of work has extensively studied improving key-value store performance by exploiting the GPU [32, 77], NMP [17, 40, 43], SSD characteristics [15, 49], and lookup query properties [45]. They are orthogonal to EVSTORE in that could help increase the throughput of key-value store operations, which can be beneficial for DRS that rely on these stores. For instance, NMP can help convert the raw EV data into a Tensor format which will reduce the CPU load. However, these techniques do not address the specific challenges associated with the growth of EV table sizes, which is the focus of EVSTORE. Additionally, many papers require either bespoke hardware modifications or emerging memory technologies, which make them elusive for commodity hardware deployments. EVSTORE, on the other hand, is designed to be compatible with commodity hardware, making it a more practical and accessible solution for improving the performance of DRS.

10 CONCLUSION

We have introduced EVSTORE: a novel 3-tier EV caching layer to address the continuous growth of EV tables in deep recommendation systems. EVSTORE is a practical system that brings several advantages. DRS designers no longer need to worry about the memory size limitation of their EV tables since users with low-memory servers can still run DRSs with large EV tables. Recommendation services can also run concurrent DRSs to increase throughput and thus potentially bolster revenue. By dislodging the monolithic DRAM-hungry DRS architectures of today with a scalable systems-oriented approach, carrying relatively modest downsides, EVSTORE has the potential to curb the enormous and ballooning operational costs and expensive resources needed to run a competitive DRS across the industry.

Scientifically, we believe EVSTORE opens several doors for future work, including in the realm of EV caching (are there better policies?) and cache management (what is the best L1-L2-L3 size arrangement?). In addition, there are many components inside EVSTORE that can be further improved, such as the bit-encoding method, the L3 remapping strategies, and the popularity ranking update mechanism. It also spurs questions around the role of emerging memory technologies and GPU-accelerated caching on future recommendation systems.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their tremendous feedback and comments. This material was supported by funding from NSF (grant Nos. CNS-1526304, CNS-1405959, CCF-2028427, and CCF-2119184) as well as generous donations from Seagate Technology. The experiments in this paper were performed in Chameleon [2, 41]. Any opinions, findings, and conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the NSF or other institutions.

REFERENCES

[1] [n. d.]. https://github.com/ucare-uchicago/ev-store-dlrm.
[2] [n. d.]. Chameleon Cloud Testbed. https://www.chameleoncloud.org/.
[3] [n. d.]. RocksDB. http://rocksdb.org/.
[4] 2013. Download Terabyte Click Logs. https://labs.criteo.com/2013/12/download-terabyte-click-logs/.
[5] 2014. Click-Through Rate Prediction: Predict whether a mobile ad will be clicked. https://www.kaggle.com/c/avazu-ctr-prediction.
[6] 2014. Display Advertising Challenge. https://www.kaggle.com/c/criteo-display-ad-challenge.
[7] 2018. Notes from the ai frontier insights from hundreds of use cases. https://www.mckinsey.com/featured-insights/artificial-intelligence/.
[8] 2019. Use cases of recommendation systems in business current applications and methods. https://emerj.com/ai-sector-overviews/use-cases-recommendation-systems/.
[9] 2020. SQLite. https://www.sqlite.org/index.html.
[10] 2021. How machine learning powers Facebook's News Feed ranking algorithm. https://engineering.fb.com/2021/01/26/ml-applications/news-feed-ranking/.
[11] 2022. Benchmarks for Java In Memory Caches. https://github.com/cache2k/cache2k-benchmark.
[12] 2022. CORTX-Motr. https://github.com/Seagate/cortx-motr.
[13] 2022. Memory Prices. https://memory.net/memory-prices/.
[14] Alaa R Alameldeen and David A Wood. 2004. Adaptive cache compression for high-performance processors. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*.
[15] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. 2009. FAWN: A fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*.
[16] Ehsan K. Ardestani, Changkyu Kim, Seung Jae Lee, Luoshang Pan, Valmiki Rampersad, Jens Axboe, Banit Agrawal, Fuxun Yu, Ansha Yu, Trung Le, Hector Yuen, Shishir Juluri, Akshat Nanda, Manoj Wodekar, Dheevatsa Mudigere, Krishnakumar Nair, Maxim Naumov, Chris Peterson, Mikhail Smelyanskiy, and Vijay Rao. 2021. Supporting Massive DLRM Inference Through Software Defined Memory. https://arxiv.org/abs/2110.11489.
[17] Bahar Asgari, Ramyad Hadidi, Jiashen Cao, Da Eun Shim, Sung Kyu Lim, and Hyesoon Kim. 2021. Fafnir: Accelerating sparse gathering by using efficient near-memory intelligent reduction. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*.
[18] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural Machine Translation by Jointly Learning to Align and Translate. In *Proceedings of The 3rd International Conference on Learning Representations (ICLR)*.
[19] Sorav Bansal and Dharmendra S. Modha. 2004. CAR: Clock with Adaptive Replacement. In *Proceedings of The FAST '04 Conference on File and Storage Technologies*.
[20] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. 2018. LHD: Improving cache hit rate by maximizing hit density. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.
[21] Nathan Beckmann and Daniel Sanchez. 2016. Modeling cache performance beyond LRU. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.
[22] Nathan Beckmann and Daniel Sanchez. 2017. Maximizing Cache Performance Under Uncertainty. In *Proceedings of the 23rd international symposium on High Performance Computer Architecture (HPCA-23)*.
[23] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, Rohan Anil, Zakaria Haque, Lichan Hong, Vihan Jain, Xiaobing Liu, and Hemal Shah. 2016. Wide & Deep Learning for Recommender Systems. In *Proceedings of The 1st Workshop on Deep Learning for Recommender Systems (DLRS@RecSys)*.
[24] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep Neural Networks for YouTube Recommendations. In *Proceedings of The 10th ACM Conference on Recommender Systems (RecSys)*.
[25] Asit Dan and Don Towsley. 1990. 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*.
[26] Per-Erik Danielsson. 1980. Euclidean distance mapping. *Computer Graphics and image processing* (1980).
[27] Jesse Davis and Mark H. Goadrich. 2006. The relationship between Precision-Recall and ROC curves. In *Proceedings of the 23rd international conference on Machine learning*.
[28] Assaf Eisenman, Maxim Naumov, Darryl Gardner, Misha Smelyanskiy, Sergey Pupyrev, Kim M. Hazelwood, Asaf Cidon, and Sachin Katti. 2019. Bandana: Using Non-Volatile Memory for Storing Deep Learning Models. In *Proceedings of The 2nd Conference on Machine Learning and Systems (MLSys)*.
[29] Pankaj Gupta, Ashish Goel, Jimmy Lin, Aneesh Sharma, Dong Wang, and Reza Zadeh. 2013. WTF: the who to follow service at Twitter. In *Proceedings of the 22nd international conference on World Wide Web (WWW)*.
[30] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, David Brooks, Bradford Cottel, Kim M. Hazelwood, Mark Hempstead, Bill Jia, Hsien-Hsin S. Lee, Andrey Malevich, Dheevatsa Mudigere, Mikhail Smelyanskiy, Liang Xiong, and Xuan Zhang. 2020. The Architectural Implications of Facebook's DNN-based Personalized Recommendation. In *Proceedings of The 26th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*.
[31] Kim M. Hazelwood, Sarah Bird, David M. Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, James Law, Kevin Lee, Jason Lu, Pieter Noordhuis, Misha Smelyanskiy, Liang Xiong, and Xiaodong Wang. 2018. Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective. In *Proceedings of The 24th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*.
[32] Tayler H Hetherington, Mike O'Connor, and Tor M Aamodt. 2015. Memcachedgpu: Scaling-up scale-out key-value stores. In *Proceedings of the 6th ACM Symposium on Cloud Computing*.
[33] Gisli R. Hjaltason and Hanan Samet. 2003. Properties of embedding methods for similarity searching in metric spaces. *IEEE Transactions on Pattern Analysis and machine intelligence* (2003).
[34] Seokin Hong, Bulent Abali, Alper Buyuktosunoglu, Michael B Healy, and Prashant J Nair. 2019. Touché: Towards ideal and efficient cache compression by mitigating tag area overheads. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*.
[35] László Jeni, Jeffrey Cohn, and Fernando De la Torre. 2013. Facing Imbalanced Data - Recommendations for the Use of Performance Metrics. *Proceedings - 2013 Humaine Association Conference on Affective Computing and Intelligent Interaction, ACII 2013*.
[36] Song Jiang, Feng Chen, and Xiaodong Zhang. 2005. CLOCK-Pro: An Effective Improvement of the CLOCK Replacement. In *Proceedings of The 2005 USENIX Annual Technical Conference*.
[37] S. Jiang and X. Zhang. 2002. LIRS: An efficient low inter reference recency set replacement policy to improve buffer cache performance. In *Proceedings of The International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS)*.
[38] Norman P Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, et al. 2021. Ten lessons from three generations shaped Google's TPUv4i: Industrial product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*.
[39] Anne Kao and Steve R. Poteet. 2007. *Natural language processing and text mining*.
[40] Liu Ke, Udit Gupta, Benjamin Youngjae Cho, David Brooks, Vikas Chandra, Utku Diril, Amin Firoozshahian, Kim M. Hazelwood, Bill Jia, Hsien-Hsin S. Lee, Meng Li, Bert Maher, Dheevatsa Mudigere, Maxim Naumov, Martin Schatz, Mikhail Smelyanskiy, Xiaodong Wang, Brandon Reagen, Carole-Jean Wu, Mark Hempstead, and Xuan Zhang. 2020. Recnmp: Accelerating personalized recommendation with near-memory processing. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*.
[41] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbach, Alex Rocha, and Joe Stubbs. 2020. Lessons Learned from the Chameleon Testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*.
[42] Guy K Kloss. 2009. Automatic C library wrapping Ctypes from the trenches. (2009).
[43] Youngeun Kwon, Yunjae Lee, and Minsoo Rhu. 2019. Tensordimm: A practical near-memory processing architecture for embeddings and tensor operations in deep learning. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*.
[44] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong-Sang Kim. 1999. 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*.
[45] Yejin Lee, Seong Hoon Seo, Hyunji Choi, Hyoung Uk Sul, Soosung Kim, Jae W. Lee, and Tae Jun Ham. 2021. MERCI: 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)*.
[46] Adam Lerer, Ledell Wu, Jiajun Shen, Timothée Lacroix, Luca Wehrstedt, Abhijit Bose, and Alex Peysakhovich. 2019. Pytorch-BigGraph: A Large Scale Graph Embedding System. In *Proceedings of The 2nd Conference on Machine Learning and Systems (MLSys)*.

[47] Huaicheng Li, Daniel S. Berger, Stanko Novakovic, Lisa Hsu, Dan Ernst, Pantea Zardoshti, Monish Shah, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. 2022. First-generation Memory Disaggregation for Cloud Platforms. https://arxiv.org/pdf/2203.00241.

[48] Leo Liberti, Carlile Lavor, Nelson Maculan, and Antonio Mucherino. 2014. Euclidean distance geometry and applications. *SIAM review* (2014).

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

[50] N. Megiddo and D. S. Modha. 2003. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of The FAST '03 Conference on File and Storage Technologies*.

[51] Xupeng Miao, Hailin Zhang, Yining Shi, Xiaonan Nie, Zhi Yang, Yangyu Tao, and Bin Cui. 2021. Het: Scaling out huge embedding model training via cache-enabled distributed framework. *arXiv:2112.07221* (2021).

[52] Jason Mohoney, Roger Waleffe, Henry Xu, Theodoros Rekatsinas, and Shivaram Venkataraman. 2021. Marius: Learning Massive Graph Embeddings on a Single Machine. In *Proceedings of The 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[53] James C. Mullikin. 1992. The vector distance transform in two and three dimensions. *CVGIP: Graphical Models and Image Processing* (1992).

[54] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. 2019. Deep learning recommendation model for personalization and recommendation systems. arXiv:1906.00091.

[55] Victor F Nicola, Asit Dan, and Daniel M Dias. 1992. Analysis of the generalized clock buffer replacement scheme for database transaction processing. In *Proceedings of the 1992 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*.

[56] Even Oldridge, Julio Perez, Ben Frederickson, Nicolas Koumchatzky, Minseok Lee, Zehuan Wang, Lei Wu, Fan Yu, Rick Zamora, O Yılmaz, Alec Gunny, and Vinh Nguyen. 2020. Merlin: a gpu accelerated recommendation framework. *Proceeding s of IRS* (2020).

[57] E. Theodore L. Omtzigt, Peter Gottschling, Mark Seligman, and William Zorn. 2020. Universal Numbers Library: design and implementation of a high-performance reproducible number systems library. *arXiv:2012.11011* (2020).

[58] Jongsoo Park, Maxim Naumov, Protonu Basu, Summer Deng, Aravind Kalaiah, Daya Shanker Khudia, James Law, Parth Malani, Andrey Malevich, Nadathur Satish, Juan Miguel Pino, Martin Schatz, Alexander Sidorov, Viswanath Sivakumar, Andrew Tulloch, Xiaodong Wang, Yiming Wu, Hector Yuen, Utku Diril, Dmytro Dzhulgakov, Kim M. Hazelwood, Bill Jia, Yangqing Jia, Lin Qiao, Vijay Rao, Nadav Rotem, Sungjoo Yoo, and Mikhail Smelyanskiy. 2018. Deep Learning Inference in Facebook Data Centers: Characterization, Performance Optimizations and Hardware Implications. https://arxiv.org/abs/1811.09886.

[59] Gang Qian, Shamik Sural, Yuelong Gu, and Sakti Pramanik. 2004. Similarity between Euclidean and cosine angle distance for nearest neighbor queries. In *Proceedings of the 2004 ACM symposium on Applied computing*.

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

[61] Trausti Saemundsson, Hjortur Bjornsson, Gregory Chockler, and Ymir Vigfusson. 2014. Dynamic performance profiling of cloud caches. In *Proceedings of the ACM Symposium on Cloud Computing*.

[62] Amit Sharma, Jake M Hofman, and Duncan J Watts. 2015. Estimating the causal impact of recommendation systems from observational data. In *Proceedings of the Sixteenth ACM Conference on Economics and Computation*.

[63] Zhenyu Song, Daniel S Berger, Kai Li, Anees Shaikh, Wyatt Lloyd, Soudeh Ghorbani, Changhoon Kim, Aditya Akella, Arvind Krishnamurthy, Emmett Witchel, et al. 2020. Learning relaxed belady for content distribution network caching. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[64] Dimitra Tsigkari and Thrasyvoulos Spyropoulos. 2022. An approximation algorithm for joint caching and recommendations in cache networks. *IEEE Transactions on Network and Service Management* (2022).

[65] Uresh Vahalia. 1996. Unix Internals: The New Frontiers.

[66] G. Vietri, L. V. Rodriguez, W. A. Martinez, S. Lyons, J. Liu, R. Rangaswami, M. Zhao, and G. Narasimhan. 2018. Driving Cache Replacement with ML-based LeCaR. In *Proceedings of The 10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*.

[67] S Vijayarani and R Janani. 2016. Text mining: open source tokenization tools-an analysis. *Advanced Computational Intelligence: An International Journal (ACII)* (2016).

[68] 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 (APSys)*.

[69] 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 (KDD)*.

[70] Ruoxi Wang, Bin Fu, Gang Fu, and Mingliang Wang. 2017. Deep & cross network for ad click predictions. In *Proceedings of ADKDD*.

[71] Ruoxi Wang, Rakesh Shivanna, Derek Cheng, Sagar Jain, Dong Lin, Lichan Hong, and Ed Chi. 2021. DCN v2: Improved deep & cross network and practical lessons for web-scale learning to rank systems. In *Proceedings of the Web Conference 2021*.

[72] 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 (ASPLOS)*.

[73] Minhui Xie, Youyou Lu, Jiazhen Lin, Qing Wang, Jian Gao, Kai Ren, and Jiwu Shu. 2022. Fleche: an efficient GPU embedding cache for personalized recommendations. In *Proceedings of the Seventeenth European Conference on Computer Systems*.

[74] Xing Xie, Jianxun Lian, Zheng Liu, Xiting Wang, Fangzhao Wu, Hongwei Wang, and Zhongxia Chen. 2018. Personalized recommendation systems: Five hot research topics you must know. https://www.microsoft.com/en-us/research/lab/microsoft-research-asia/articles/personalized-recommendation-systems/. *Microsoft Research Lab-Asia* (2018).

[75] Ming Xue and Changjun Zhu. 2009. The socket programming and software design for communication based on client/server. In *Pacific-Asia Conference on Circuits, Communications and Systems*.

[76] Jie Amy Yang, Jianyu Huang, Jongsoo Park, Ping Tak Peter Tang, and Andrew Tulloch. 2020. Mixed-Precision Embedding Using a Cache. *arXiv:2010.11305* (2020).

[77] Kai Zhang, Kaibo Wang, Yuan Yuan, Lei Guo, Rubao Lee, and Xiaodong Zhang. 2015. Mega-kv: A case for gpus to maximize the throughput of in-memory key-value stores. *Proceedings of the VLDB Endowment* (2015).

[78] Weijie Zhao, Deping Xie, Ronglai Jia, Yulei Qian, Ruiquan Ding, Mingming Sun, and Ping Li. 2020. Distributed Hierarchical GPU Parameter Server for Massive Scale Deep Learning Ads Systems. In *Proceedings of The 3rd Conference on Machine Learning and Systems (MLSys)*.

[79] Weijie Zhao, Deping Xie, Ronglai Jia, Yulei Qian, Ruiquan Ding, Mingming Sun, and Ping Li. 2020. Distributed hierarchical gpu parameter server for massive scale deep learning ads systems. *Proceedings of Machine Learning and Systems* (2020).

[80] 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*.

[81] Zhe Zhao, Lichan Hong, Li Wei, Jilin Chen, Aniruddh Nath, Shawn Andrews, Aditee Kumthekar, Maheswaran Sathiamoorthy, Xinyang Yi, and Ed H. Chi. 2019. Recommending what video to watch next: a multitask ranking system. In *Proceedings of the 13th ACM Conference on Recommender Systems (RecSys)*.

[82] Da Zheng, Xiang Song, Chao Ma, Zeyuan Tan, Zihao Ye, Jin Dong, Hao Xiong, Zheng Zhang, and George Karypis. 2020. DGL-KE: Training Knowledge Graph Embeddings at Scale. In *Proceedings of The 43rd International ACM SIGIR conference on research and development in Information Retrieval (SIGIR)*.

[83] Guorui Zhou, Na Mou, Ying Fan, Qi Pi, Weijie Bian, Chang Zhou, Xiaoqiang Zhu, and Kun Gai. 2019. Deep Interest Evolution Network for Click-Through Rate Prediction. In *Proceedings of The 31st Innovative Applications of Artificial Intelligence Conference*.

[84] Guorui Zhou, Xiaoqiang Zhu, Chengru Song, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, Han Li, and Kun Gai. 2018. Deep Interest Network for Click-Through Rate Prediction. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD)*.

Received 2022-07-07; accepted 2022-09-22