THE UNIVERSITY OF CHICAGO

EVSTORE: SCALING EMBEDDING TABLES FOR DEEP RECOMMENDATION SYSTEMS

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY
DANIAR H. KURNIAWAN

CHICAGO, ILLINOIS
OCTOBER 2022

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Recommendation systems are used prominently across modern online services to help people make decisions. The impact of recommendation systems on user engagement is tremendous. Modern Deep Recommendation Systems (DRS), such as Facebook's post recommendation systems, 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. 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 and Google's DCN, for example, store the full EV tables in DRAM and lack support for responding to lookups from backend storage when memory is exhausted.

Accordingly, we propose 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. In this project, we propose to (1). design an embedding-aware caching algorithm that better exploits the regularity of the DRS inference workload, (2). evaluate the new caching algorithms against state-of-the-art policies, (3). study the effectiveness of our algorithm in various production traces, (4). built a caching layer (EVCache) to evaluate our new caching algorithm within the real DRS pipeline, (5). develop a mixed-precision (EVMix) caching method to increase hit rate in trade off a minor accuracy loss.

CHAPTER 1

THESIS STATEMENT

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. We propose 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 2-layer caching design where we harness an all-or-nothing EV access property and enable mixed-precision caching to increase hit rates, accelerate inferences, and boost throughput in trade for a minor loss of accuracy.

CHAPTER 2
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 [21, 22], rank news [7, 17], and recommend products [48]. 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 [5, 6, 41, 52].

Deep recommendation systems (DRSs) are widely used to deliver high-quality recommendations [22, 53], but tackling *categorical ("sparse") input features* is their Achilles' heel. Modern DRSs, such as Facebook's post recommendation systems [22], 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.

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 [23, 48] 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 [22]. Moreover, industry's insatiable appetite for improved recommendation accuracy demands larger EV tables to encode richer semantic relationships. Thus, the recommendation models are growing rapidly—the sizes are tripling every two years (1.5× annual growth) [10, 28].

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 [35] and Google's DCN [49, 50], 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 [8]. 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 [20, 47, 51]. 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 2-layer "L1" and "L2" caching design (EVCache and EVMix).

CHAPTER 3

BACKGROUND

Here we explain the details about embedding table, challenges faced by modern DRS, and the existing caching policies.

3.1 Deep Recommendation Systems

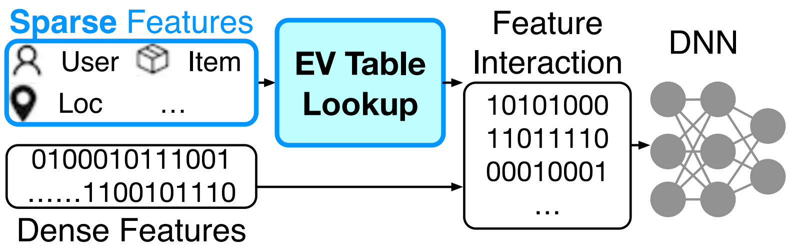

Figure 3.1: **DRS and EV Tables (Figure 3.1).** *To make DNNs less complex, sparse data is converted to embedding vectors via EV tables.*

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 [29, 46], 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** 3.1).

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.

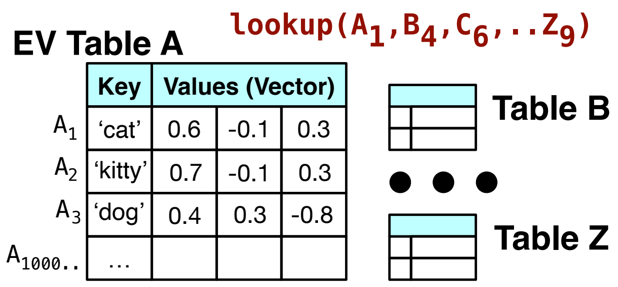

Figure 3.2: **EV table structure and lookup.** *An example of EV tables A–Z in a DRS. A lookup involves finding a key in each of the 26 tables.*

Embedding vectors (EV) are the most popular method for densifying sparse input features for the DRS, effectively translating sparse categorical data into dense vectors of numbers [11]. 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 3.2**. 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 3.2 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 [32, 34, 55]. In Figure 3.2, for example, 26 different EV tables must be consulted for a single inference. We denote DRS lookups by:

$$\texttt{lookup(}\texttt{A}_1\texttt{,}\texttt{B}_4\texttt{,}\texttt{C}_6\texttt{,..,}\texttt{Z}_9\texttt{),}$$

where the number in the subscript represents a key in the table. For instance, $\texttt{B}_4$ 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 [16, 17, 22, 54, 56, 57]. Each categorical feature could assume 10^7–10^{10} different possible values [23, 38, 53], implying that billions of embedding vectors are needed in practice to represent every unique feature. A billion embedding vectors (rows) with 400 dimensions (columns) [32, 34, 55] 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$\times$ annual growth), following Moore's Law [10, 28], while the underlying DRAM-hungry DRS implementations already weigh heavily in company budgets [22].

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 [10], flirting with the limits of even the greatest memory capacity cloud instances available[1]. To continue scaling DRS, a different approach is required.

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

3.2 Existing Caching Policies

These are the state-of-the-art caching algorithm that have been developed in the recent years:

CLOCK / Second Chance Replacement: CLOCK (known as Second Chance replacement policy) is a classical policy dating back to 1968 that was proposed as a low-complexity approximation to LRU. An LRU needs to move the accessed item to the most recently used position on every cache hit, at which point, to ensure consistency and correctness, it serializes cache hits behind a single global lock. CLOCK eliminates this lock contention, and, hence, can support high concurrency and high throughput environments. Unfortunately, CLOCK is still plagued by disadvantages of LRU such as disregard for ”frequency”, susceptibility to scans, and low performance.

Adaptive Replacement Cache (ARC): ARC [23] is an adaptive caching algorithm that is designed to recognize both recency and frequency of access. ARC divides the cache into two LRU lists, T1 and T2. T1 holds items accessed once while T2 keeps items accessed more than once since admission. Since ARC uses an LRU list for T2, it is unable to capture the full frequency distribution of the workload and perform well for LFU-friendly workloads. For a scan workload, new items go through T1 protecting frequent items previously inserted into T2.

Clock with adaptive replacement (CAR): CAR is a cache replacement algorithm, which combines Adaptive Replacement Cache (ARC) and CLOCK, and it has performance comparable to ARC, and substantially outperforms both LRU and CLOCK. The algorithm CAR is self-tuning and requires no user-specified magic parameters.

Low Interference Recency Set (LIRS): LIRS [13] is a state-of-the-art caching algorithm based on reuse distance. LIRS handles scan workloads well by routing one-time accesses via its short filtering list Q. However, LIRS's ability to adapt is compromised because of its use of a fixed-length Q. In particular, if reuse distances exceed the 1% length, LIRS is unable to recognize reuse quickly enough for items with low overall reuse. And, similar to ARC, LIRS does not have access to the full frequency distribution of accessed items which limits its effectiveness for LFU-friendly workloads.

Dynamic LIRS (DLIRS): DLIRS [20] is a recently proposed caching policy that incorporates adaptation in LIRS. DLIRS dynamically adjusts the cache partitions assigned to high and low reuse-distance items. Although this strategy achieves performance comparable to ARC for some cache size configurations with LRU-friendly workloads while maintaining LIRS's behavior for scans, we found its performance inconsistent across the workloads we tested against. Finally, it inherits the LFU-unfriendliness of LIRS.

Learning Cache Replacement (LeCaR): LeCaR [34] is a machine learning-based caching algorithm that uses reinforcement learning and regret minimization to control its dynamic use of two cache replacement policies, LRU and LFU. LeCaR was shown to outperform ARC for small cache sizes for real-world workloads [34]. However, LeCaR has drawbacks relating to adaptiveness, overhead, and churnfriendliness. In Section 3, we discuss these limitations further.

CACHEUS: CACHEUS uses online reinforcement learning with regret minimization to build a caching algorithm that attempts to optimize for dynamically manifesting workload primitive types. Since CACHEUS' design draws heavily from LeCaR, we review it briefly first, conduct an investigative study of LeCaR, and finally discuss the CACHEUS algorithm.

CHAPTER 4

EVSTORE DESIGN

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 two key components to the EVSTORE design, depicted in **Figure 4.1**:

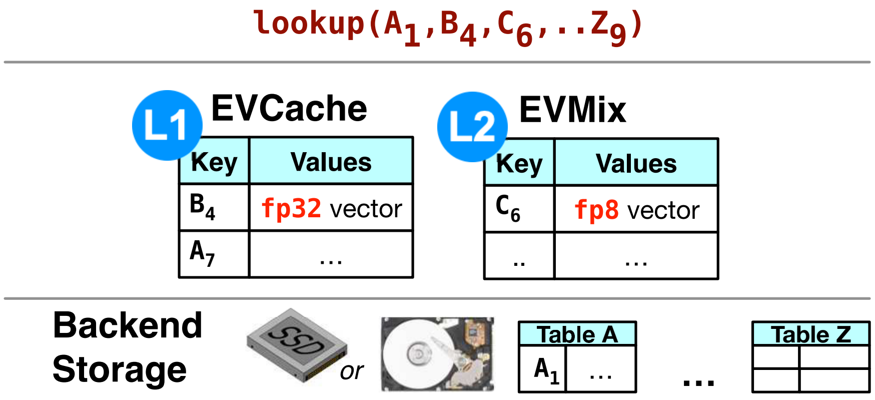

Figure 4.1: **EVSTORE design overview.** EVSTORE *is composed of EVCache (L1), an EV table caching layer with various cache replacement options (section 4.1+chapter 5); L2, a second caching layer that stores lower precision embedding such as in* `fp8` *(section 4.2+chapter 6). The* `lookup(`A_1`,`B_4`,`C_6`,...,`Z_9`)` *will lead to* B_4 *hit in L1,* C_6 *hit in L2, and* A_1 + Z_9 *misses will incur disk access to the backend storage.*

(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.

4.1 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.* $\texttt{lookup(A}_1\texttt{,B}_4\texttt{,C}_6\texttt{,..,Z}_9\texttt{)}$—a cache miss for just *one* of the keys (*e.g.*, $\texttt{A}_1$) 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 [12, 31, 33, 39]) against DRS workloads with the all-or-nothing property and found their performance to leave an opportunity for improvement (section 5.1). We noticed that existing cache algorithms could be infused with a novel notion of *"groupability"*. That is, EVfriendly 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 (section 5.2). 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 (section 5.3). 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 5.2.

4.2 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 [9, 24, 43]. 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 6.

CHAPTER 5

EVCACHE (L1)

We next evaluate the performance of various caching algorithms on EV lookup workloads (section 5.1), detail how we apply our groupability principle to help bolster perfect hit rates (section 5.2), and demonstrate how the principle may be adopted across various caching policies (section 5.3).

5.1 The Importance of Perfect Hits

The caching literature is replete with algorithms, from the basic policies (such as LRU, CLOCK, and LFU [18, 31, 36, 44]) to the more dynamic/adaptive variants (such as LIRS [26], CAR [12], ARC [33], ClockLIRS [26], and ClockPro [27]), and finally the machine-learning based ones (such as Cacheus [39] and LeCAR [45]). 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 5.1 shows the results when we have $N = 26$ using the Criteo dataset [4] (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

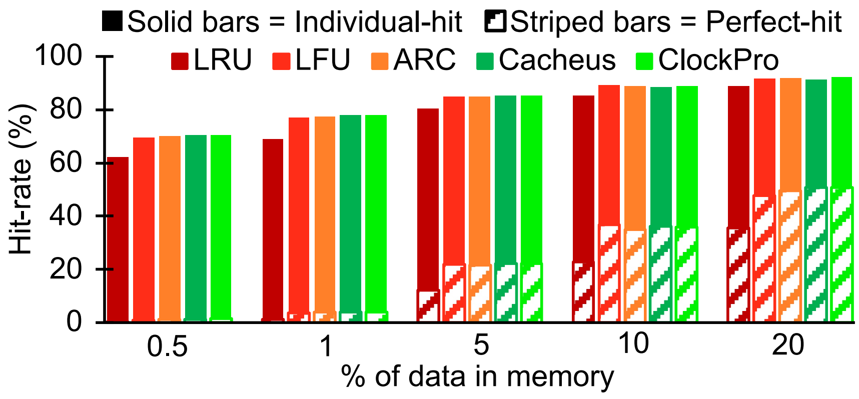

Figure 5.1: **Individual vs. perfect hit rates (section 5.1).** *Existing algorithms exhibit high individual hit rates (solid bars) but relatively low perfect hit rates (striped bars) across various cache sizes (horizontal axis).*

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.

5.2 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 20 as the group score and utilize it in the next caching operations.

Cache state update: For every key with a cache hit, *e.g.* `B4`, its value stored in the cache will be read and prepared to be supplied to the neural network. Next, EVCache updates the `B4`'s group score in the cache with the `max` of the current and the new score, further detail on the "`max`-based" group scoring is covered at the end of this section. For example, if key `B4` is a hit and its current group score is 15, then EVCache will update `B4`'s score to 20 (the memorized score).

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. The reason is that decades of caching studies show recency (introduced by the newly inserted keys) is an important component of caching[18, 26, 27, 36].

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 [15, 18], tenuring highly-frequent items [31], exploiting other data attributes [13, 14, 15], and even learning request histories through non-linear machine-learning approaches [42]. 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 into popular cache replacement policies.

```
                                EVCache Pseudocode
---------------------------------------------------------------------------------------
Global variable:
    struct EVData = an embedding vector data struct
    List<EVData> ArrCachedEV = Array to cache EVData.

EVData[] funcRequest(List<string> keys[]):
    EVData[] arrEVData = []
    int aggHit = funcGetAggHit(keys[])

    /** Update data in the ArrCachedEV **/
    for each k in keys:
        if (k in ArrCachedEV):
            /** A HIT **/
            /** Update position of the key in the ArrCachedEV **/
            currEVData = funcUpdate(k, aggHit)
        else:
            /** A MISS **/
            currEVData = funcFetchData(k)
            funcInsert(currEVData, aggHit)
        endif
        /** Insert a new page to arrEVData **/
        arrEVData.append(currEVData)
    endfor
    return arrEVData

int funcGetAggHit(List<string> keys[]):
    int aggHit = 0
    for each k in keys:
        if (k in ArrCachedEV):
            aggHit ++
        endif
    endfor
    return aggHit

void funcUpdate(String key, int aggHit):
    /** Use aggHit to update position of the data. The higher the aggHit, the more
     secure the position. We will use funcGetEVData() to get the EV-data based on the
     given key. **/
    return EVData

void funcInsert(EVData newEVData, int aggHit):
    if (ArrCachedEV is full):
        funcEvict()
    endif
    /** Insert the newEVData to ArrCachedEV. Use the aggHit to position the data. The
     higher the aggHit, the more secure the position. **/

void funcEvict():
    /** Evict the Least Groupable data at ArrCachedEV **/

EVData funcGetEVData(List ArrCachedEV, string key,
        int _aggHit):
    /** Find the requested embedding data at ArrCachedEV[] based on the given key. If
     the _aggHit > data's aggHit, we will replace it with the _aggHit. **/
    return EVData

EVData funcFetchData(string key):
    /** Get the embedding data from secondary storage based on the given key **/
    return EVData
----------------------------------------------------------------------------------------
```

5.3 EVCache Variants

To show generality, we implemented our extension to three popular (base) algorithms: LFU [31], CAR [12], and ARC [33]. 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.

1. EV-LFU: We replace the default frequency counter in LFU [31] with a group score. In other words, upon a miss, EV-LFU will evict the cached item with the lowest group score. If there are multiple candidates, EV-LFU evicts the least recently inserted item. EV-LFU's group score has a maximum value (*e.g.*, 26 in our main benchmark). When most of the cached items have 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 [40], which provides a further degree of robustness to EV-LFU in practical settings.

```
                                 EV-LFU Algorithm
-------------------------------------------------------------------------------------
Global variable:
    struct EVData {string key, float[] value, int aggHit } = mbedding vector struct.
    int cacheSize = capacity of the cache.
    int maxAggHit = The upper limit of aggHit value.
    List<EVData> ArrCachedEV = Array to cache EVData.
    int nPerfectPage  = The number of perfectPages.
    float flushingRate  = The ratio of perfectPage to delete during the flushing.
    float perfectPageCapacity = The threshold to initiate the flushing phase.

Initialization:
    flushingRate = 0.1
    perfectPageCapacity = 0.9

EVData funcUpdate(string key, int aggHit):
    EVData currEVData = funcGetEVData(key)
    if (currEVData.aggHit < aggHit) then
        currEVData = funcSetAggHit(currEVData, aggHit)
    endif
    return currEVData

void funcInsert(EVData newEVData, int aggHit):
    if (nPerfectPage >= perfectPageCapacity * cacheSize):
        funcFlushPerfectPages()
    else if (ArrCachedEV is full):
        funcEvict()
    endif
    newEVData = funcSetAggHit(newEVData, aggHit)
    ArrCachedEV.insert(newEVData)

void funcFlushPerfectPages():
    for(int i = 0; i < flushingRate * nPerfectPage; i++):
        /** Evict the least recent perfectPage.**/
    endfor
    nPerfectPage -= flushingRate * nPerfectPage

EVData funcSetAggHit(EVData currEVData, int aggHit):
    if (aggHit == maxAggHit):
        nPerfectPage ++
    endif
    currEVData.aggHit = aggHit
    return currEVData

void funcEvict():
    /** Evict the least recently used data at ArrCachedEV, similar to LFU's method **/
---------------------------------------------------------------------------------------
```

2. EV-ARC: ARC [33] 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 caches 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`.

```
                              EV-ARC Algorithm
--------------------------------------------------------------------------------
Global variable:
    struct EVData { string key, float[] value, int aggHit} = Embedding vector struct.
    int cacheSize = capacity of the cache.
    int maxAggHit = The upper limit of aggHit value.
    List<EVData> listRecentEV = recency List
    List<EVData> listFrequentEV = frequency List
    int nPerfectPage  = The number of perfectPages.
    float flushingRate  = The ratio of perfectPage to delete during the flushing.
    float perfectPageCapacity = The threshold to initiate the flushing phase.

Initialization:
    flushingRate = 0.1
    perfectPageCapacity = 0.95

EVData funcUpdate(string key, int aggHit):
    /** Check whether the "key" exists at the recency or frequency list. **/
    if ( listRecentEV.contains(key) ):
        return funcGetEVData(listRecentEV, key, aggHit)
    else if ( listFrequentEV.contains(key) ):
        return funcGetEVData(listFrequentEV, key, aggHit)
    else:
        /** This is a MISS, must insert a new page. **/
        return funcInsert(key, aggHit)
    endif

void funcInsert(EVData newEVData, int aggHit):
    if (nPerfectPage at frequencyList >=
        perfectPageCapacity * cacheSize):
     funcFlushPerfectPages()
    /** This function will insert the "newEVData" and its aggHit to the recency/
     frequency list based on various conditions (such as recencyTarget value). However,
     we didn't change any of those behaviors. Also, the minimum frequency counter might
     be updated, similar to the EV-LFU case. **/
    return newEVData

void funcFlushPerfectPages():
    for(int i = 0; i < flushingRate * nPerfectPage; i++):
        /** Evict the least recent perfectPage.**/
    endfor
    nPerfectPage -= flushingRate * nPerfectPage

void funcEvict():
    /** Depends on the recencyTarget, the eviction could happen at listRecentEV or
     listFrequentEV. To evict the data at listRecentEV, it will remove the least
     recently used data. If the eviction is performed at listFrequentEV, it will find
     the group of data that has smallest "counter" and evict the least recently used
     within that group. **/
--------------------------------------------------------------------------------
```

3. EV-CAR: CAR [12] 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 modify 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 (section 5.2) which replaces the current group score if the new score is bigger.

```
                                    EV-CAR Algorithm
---------------------------------------------------------------------------------------
Global variable:
    int recencyTarget = the maximum page that should be stored at clockRecentEV.
    int decPartitionSize = cacheSize/maxAggHit; which is the number of pages need to
            be checked before increasing the decrementRate.
    struct EVData {
        string key, float[] value, boolean referenced, int aggHit
    } = The struct of embedding vector.
    CLOCK<EVData> clockRecentEV = recency CLOCK
    CLOCK<EVData> clockFrequentEV = frequency CLOCK

EVData funcUpdate(string key, int aggHit):
    /** Check whether the "key" exists at the recency or frequency clock. **/
    if ( clockRecentEV.contains(key) ):
        return funcGetEVData(clockRecentEV, key, aggHit)
    else if ( clockFrequentEV.contains(key) ):
        return funcGetEVData(clockFrequentEV, key, aggHit)
    else:
        /** This is a MISS, must insert a new page **/
        return funcInsert(key, aggHit)
    endif

EVData funcInsert(string key, int aggHit):
    EVData newEVData = funcFetchData(key)
    if ( cache is full):
        funcReplace(aggHit)
    endif
    /** This function will insert the "newEVData" to the recency clock data structure
     and might adjust the recencyTarget (which will define the size of recency and
     frequency clock) based on various conditions. However, we didn't change any of
     those behaviors. The noteworthy difference is that EV-LFU adds aggHit value to
     the page's metadata when inserting a new page to the cache. **/
    return newEVData

void funcReplace(int aggHit)
    /** This is the Eviction phase where the clock "hand" is trying to find the
     least-groupable page to evict. **/
    int counter = 0;
    int decrementRate = 1;
    while (true):
        if ( clockRecentEV.size > recencyTarget):
            /** Get the data pointed by the "hand" **/
            EVData data = clockFrequentEV.getFirstData()
            /** If data.referenced bit is 1, the data will be moved to clockFrequentEV.
             Otherwise, it will be removed from the clockRecentEV and then break the
             loop. This is the same as the original mechanism at CAR policy **/
        else:
            /** Get the data pointed by the "hand" **/
            EVData data = clockFrequentEV.getFirstData()
            if (data.referenced):
                if (data.aggHit <= aggHit):
                    data.reference = false
                else if (data.aggHit-decrementRate <= 0):
                    /** Progressive Decrement Phase **/
                    data.aggHit -= decrementRate;
                    counter++;
                    if (counter >= decPartitionSize):
                        decrementRate++;
                        counter = 0;
                    endif
                /** advance the hand to the next page **/
                endif
            else:
                funcEvict(clockFrequentEV);
                break;
            endif
        endif
    endwhile

void funcEvict(CLOCK<EVData> clockData):
    /** Evict the data pointed by "hand" at clockData. **/

EVData funcGetEVData(CLOCK<EVData> clockData, string key,
        int aggHit):
    EVData currEVData = clockEV.get(key)
    if (currEVData.aggHit < aggHit):
        currEVData.aggHit = aggHit
    endif
    return currEVData
-------------------------------------------------------------------------------------
```

CHAPTER 6

EVMIX (L1 + 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. We first describe the advantages (section 6.1), the design (section 6.2), and finally, the bit coding optimizations (section 6.3).

6.1 Advantages of Mixed Precisions

An embedding vector is stored as floating point values. In most systems such as DLRM [35] and DCN [49], 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).

6.2 Multi-Tier, Mixed-Precision Design

As shown earlier in Figure 4.1, 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 $2\times$ 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 while using EVMix, the popular items are stored in L1 while the less popular ones are packed in L2. The L1/L2 placement algorithm also ensures that the items are not redundantly stored.

Furthermore, to maximize performance, we implement EVMix in C++ which utilizes multi-threading 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.

6.3 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. Thus, 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 embedding vector data management, we exploit that embedding vector values range only from -1 to 1 rather than an arbitrary range of values. The typical value distribution is a Gaussian bell-shaped curve where the occurrence/frequency is most highly concentrated near 0. Therefore, we optimize the use of every single bit to better represent this “dense region”. We design the coding procedure for simplicity to ensure that decoding remains computationally cheap and does not become a bottleneck.

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. If the last digit is odd, then the value will be negative, and *vice versa*. The decoding phase will convert each value into a corresponding floating-point value proportionally. **(b) 8 bit:** Here, 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:** We 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 [37] 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 $3\times$ 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.

CHAPTER 7
PREMILIMINARY EVALUATION

EVSTORE is a practical system. For our evaluation, 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 cache layer placement policies. We structure our evaluation as a sequence of experimental questions.

7.1 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.

Implementation: EVSTORE is built within the popular Facebook PyTorch-based DLRM framework [35] that supports both recommendation model training and inference. We believe EVSTORE is the first system to support substantial caching capabilities for the EV Table lookups in this DRS framework. 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. Next, the data can be fetched from the file using either a memory map (mmap) or direct IO. Furthermore, we send the data to PyTorch as bytestream without any serialization overhead. We added a module in PyTorch to convert the bytestream into a Tensor format. 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 (set and hashmap) to replace the default tensor lookup.

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) [2] and the Kaggle version (Criteo-Kaggle) [4]. 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 [3].

Default values: We omit redundant lines and numbers on some of the graphs for improved readability. Here are our default values, unless otherwise noted. Cache size: 5% of the total working set (the total size of all tables); Dataset/workload: Criteo-Kaggle [4]; Embedding precision: `fp32`; Latency measurement: average latency in ms.

Machine specification: We use Chameleon cloud's `gpu_rtx6000` and `gpu_v100` nodes [1, 30] 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.

7.2 EVCache Evaluation

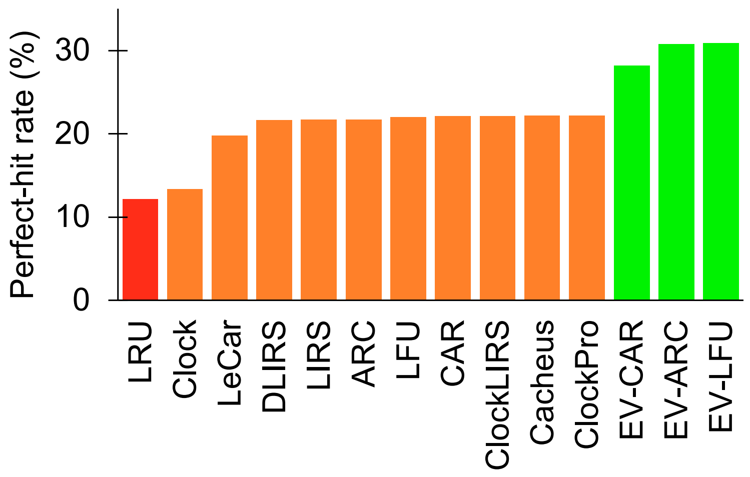

Figure 7.1: **Exp. #1: Perfect hit rates across caching algorithms.** *EVCache algorithms (EV-CAR, EV-ARC, EV-LFU) have the highest perfect hit rate.*

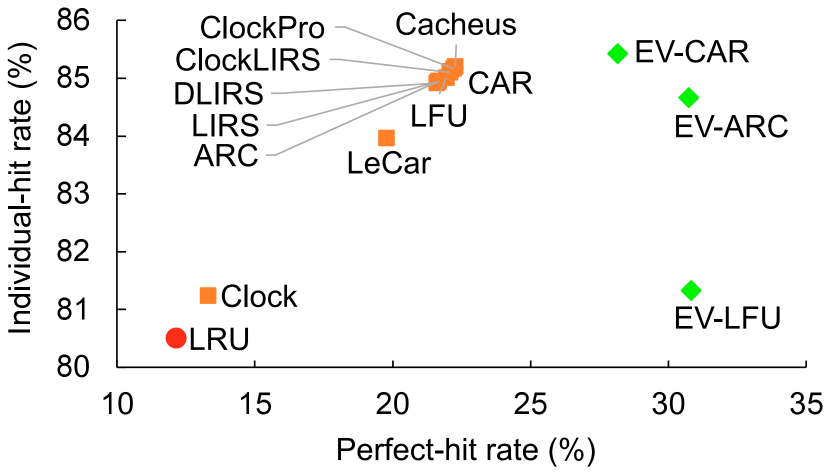

Figure 7.2: **Exp. #1: Individual and perfect hit rates across algorithms.** *EV-LFU increases perfect hits by sacrificing on individual hits.*

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

Experiment #1: How much does the EVCache algorithm affect perfect hit rates? Figure 7.1 shows that EVCache (EV-$*$) algorithm extensions improve upon state-of-the-art algorithms such as LRU [18], CLOCK [36], LeCar [45], LIRS [26], ARC [33], LFU [31], CAR [12], ClockLIRS [26], Cacheus [39], and ClockPro [27]. The perfect hit rates are increased by up to 18%, lending

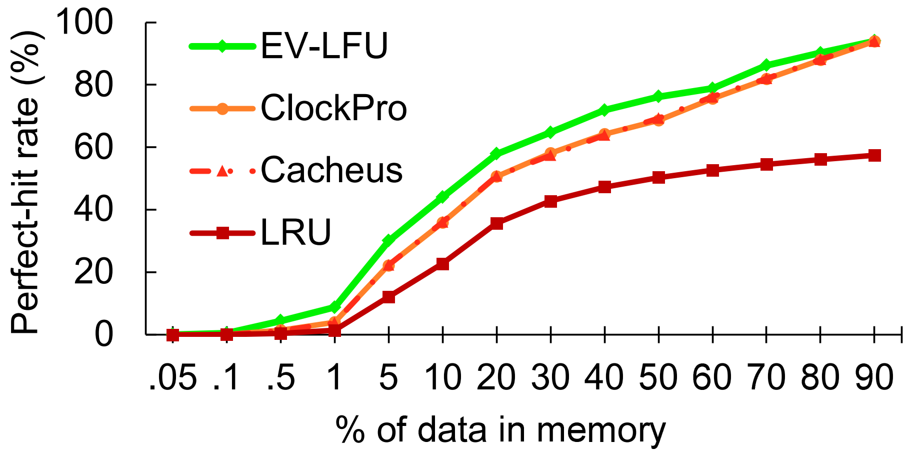

Figure 7.3: **Exp. #2: Perfect hit rates across various cache sizes.** *Our EV-LFU has the highest perfect hit rate compared to other representative algorithms across various sizes.*

support to the need for groupability for EV-based caches. **Figure 7.2** breaks the result down further to compare the perfect and individual hit rates (as defined in Section 5.1). Here, EV-CAR and EV-ARC both improve the perfect hit rates without compromising on individual hit rates, 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 crucial for DRS' embedding lookup workloads).

Experiment #2: How does EVCache perform across various cache sizes? In **Figure 7.3**, 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, we see EVCache (specifically EV-LFU) outperforming others across differing 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 7.4** 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.

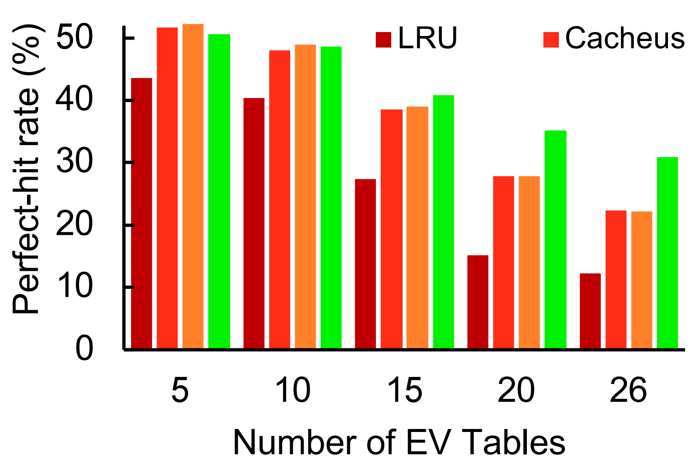

Figure 7.4: **Exp. #3: Perfect hit rates on different number of EV tables.** *EV-LFU shows steeper benefit as the number of EV tables grows (e.g., 5 to 26).*

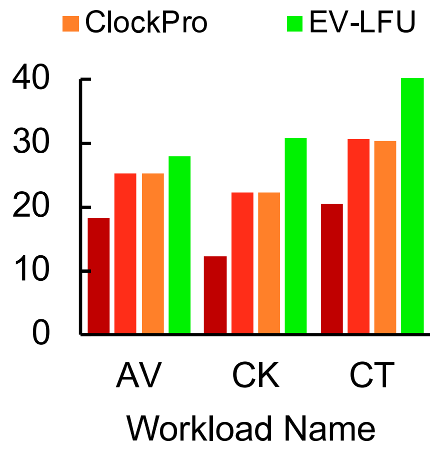

Figure 7.5: **Exp. #4: Perfect hit rates across various datasets.** *EV-LFU has the best perfect hit rate across all datasets.*

Experiment #4: How does EVCache perform across various datasets? **Figure 7.5** again 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) [3], Criteo-Kaggle (CK) [4], and Criteo-Terabyte (CT) [2].

7.3 EVMix Evaluation

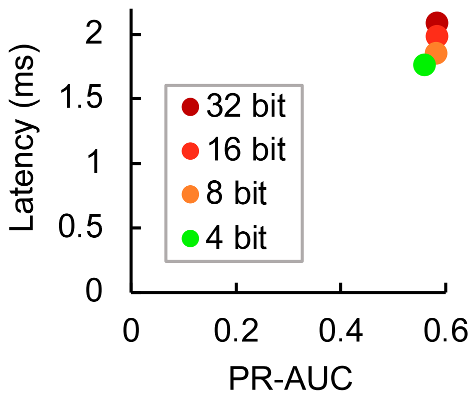

Figure 7.6: **Exp. #5: Trade-off between latency and accuracy.** *Reducing the precision from 32 to 4 bits decreases the accuracy only slightly while greatly improving the latency.*

Experiment #5: What are the latency-accuracy tradeoffs in floating point resolution (32 to 4 bits)? Figure 7.6 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 [19, 25]. Intuitively, PR-AUC measures the extent to which a classifier correctly identifies all positive labels without mistaking too many others as positive.

Future Evaluations:

Now that we know the potential of having EVMix, we will integrate L1 and L2 implementation into the DRS platform and evaluate the end-to-end inference latency. We will also test various optimization that we can apply to EVMix, such as porting the implementation to C/C++ and experimenting with different transport layer (Socket, Ctypes, or Cython).

CHAPTER 8

OTHER PROJECTS

Besides the EVStore project, here is the list of other projects that I work on.

Past Projects

1. PBSE: A Robust Path-Based Speculative Execution for Degraded-Network Tail Tolerance in Data-Parallel *(published at SOCC'17)*

   We are improving the speculative execution mechanism in various distributed storage systems. My contribution is integrating the novel speculative policy in Apache Spark.

2. FlyMC: Highly Scalable Testing of Complex Interleavings in Distributed Systems *(published at EUROSYS'19)*

   This project introduces a novel technique to minimize space-explosion problems when running model checking at scale. I was working on finding the concurrency bugs in Cassandra and testing the model-checker on the real Cassandra deployment.

3. E2E: Embracing User Heterogeneity to Improve Quality of Experience on the Web *(published at SIGCOMM'19)*

   We introduce a novel request scheduling technique for cloud-scale web services in order to improve users' quality of experience. I am contributing to analyzing 1TB of Microsoft's cloud traces to evaluate the E2E claims against the production traces.

4. Let's Cut the Tail Together with LIBROS: Library, Runtime and OS Supports for Multi-Resource Storage *(published at CLOUD'22)*

   We propose a software architecture to better cut the tail in multi-resource storage. My work is focusing on implementing the request-rejection mechanism at Java Virtual Machine.

Current Projects

1. FlashNet

   We are designing and developing an open ML training ground for IO latency prediction. This work aims to improve systems performance predictability. The ML model can be integrated to OS or any storage systems for better request scheduling.

2. MLforStorage

   We study how to improve the internal decision-making inside storage systems with the help of the ML method. One of the directions is to reduce the Erasure coding metadata by using a Lightweight ML model.

CHAPTER 9

RESEARCH PLAN

Task	Description	Planning
	EVStore Improvement	
T1	Find another caching optimization method	To be completed by December 2022.
T2	Integrate new method with the L1 and L2 layers	To be completed by February 2023.
T3	Analyze and evaluate the end-to-end system	To be completed by April 2023.
	FlashNet: ML for Storage	
T4	Implement ML to improve IO latency prediction	To be completed by January 2023.
T5	Implement ML for erasure coding and request scheduling	To be completed by March 2023.
T6	Perform end-to-end system evaluation	To be completed by May 2023.
T7	Write FlashNet paper and submit to conference	To be completed by July 2023.
	Dissertation Writing	
T8	Write the dissertation document.	To be completed by August 2023.
T9	Defense.	To be completed by September 2023.

Table 9.1: **Research Planning.** *The table reflects the planning for improving* EVSTORE *and also finishing FlashNet project. All the deadlines presented in the last column are approximations.*

CHAPTER 10

SUMMARY

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. In the shorter term, we will continue to explore several methods to improve EVSTORE, such as applying data compression and data approximation to reduce the embedding table size. 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 size arrangement?). It also spurs questions around the role of emerging memory technologies and GPU-accelerated caching on future recommendation systems.

REFERENCES

[1] Chameleon. https://www.chameleoncloud.org.

[2] Download Terabyte Click Logs. https://labs.criteo.com/2013/12/download-terabyte-click-logs/, 2013.

[3] Click-Through Rate Prediction: Predict whether a mobile ad will be clicked. https://www.kaggle.com/c/avazu-ctr-prediction, 2014.

[4] Display Advertising Challenge. https://www.kaggle.com/c/criteo-display-ad-challenge, 2014.

[5] Notes from the ai frontier insights from hundreds of use cases. https://www.mckinsey.com/featured-insights/artificial-intelligence/, 2018.

[6] Use cases of recommendation systems in business current applications and methods. https://emerj.com/ai-sector-overviews/use-cases-recommendation-systems/, 2019.

[7] How machine learning powers Facebook's News Feed ranking algorithm. https://engineering.fb.com/2021/01/26/ml-applications/news-feed-ranking/, 2021.

[8] Server Memory Prices: Server RAM Module Price List & Trends. https://memory.net/memory-prices/, 2022.

[9] Alaa R Alameldeen and David A Wood. Adaptive cache compression for high-performance processors. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, 2004.

[10] 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. Supporting Massive DLRM Inference Through Software Defined Memory. https://arxiv.org/abs/2110.11489, 2021.

[11] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural Machine Translation by Jointly Learning to Align and Translate. In *Proceedings of The 3rd International Conference on Learning Representations (ICLR)*, 2015.

[12] Sorav Bansal and Dharmendra S. Modha. CAR: Clock with Adaptive Replacement. In *Proceedings of The FAST '04 Conference on File and Storage Technologies*, 2004.

[13] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. LHD: Improving cache hit rate by maximizing hit density. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018.

[14] Nathan Beckmann and Daniel Sanchez. Modeling cache performance beyond LRU. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[15] Nathan Beckmann and Daniel Sanchez. Maximizing Cache Performance Under Uncertainty. In *Proceedings of the 23rd international symposium on High Performance Computer Architecture (HPCA-23)*, February 2017.

[16] 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. Wide & Deep Learning for Recommender Systems. In *Proceedings of The 1st Workshop on Deep Learning for Recommender Systems (DLRS@RecSys)*, 2016.

[17] Paul Covington, Jay Adams, and Emre Sargin. Deep Neural Networks for YouTube Recommendations. In *Proceedings of The 10th ACM Conference on Recommender Systems (RecSys)*, 2016.

[18] Asit Dan and Don Towsley. An approximate analysis of the lru and fifo buffer replacement schemes. In *Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, 1990.

[19] Jesse Davis and Mark H. Goadrich. The relationship between Precision-Recall and ROC curves. In *Proceedings of the 23rd international conference on Machine learning*, 2006.

[20] Assaf Eisenman, Maxim Naumov, Darryl Gardner, Misha Smelyanskiy, Sergey Pupyrev, Kim M. Hazelwood, Asaf Cidon, and Sachin Katti. Bandana: Using Non-Volatile Memory for Storing Deep Learning Models. In *Proceedings of The 2nd Conference on Machine Learning and Systems (MLSys)*, 2019.

[21] Pankaj Gupta, Ashish Goel, Jimmy Lin, Aneesh Sharma, Dong Wang, and Reza Zadeh. Wtf: the who to follow service at twitter. In *Proceedings of the 22nd international conference on World Wide Web (WWW)*, 2013.

[22] 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. The architectural implications of facebook's dnn-based personalized recommendation. In *Proceedings of The 26th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2020.

[23] 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. Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective. In *Proceedings of The 24th IEEE International Symposium on High-Performance Computer Architecture*, 2018.

[24] Seokin Hong, Bulent Abali, Alper Buyuktosunoglu, Michael B Healy, and Prashant J Nair. Touché: Towards ideal and efficient cache compression by mitigating tag area overheads. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019.

[25] László Jeni, Jeffrey Cohn, and Fernando De la Torre. Facing imbalanced data - recommendations for the use of performance metrics. 2013.

[26] S. Jiang and X. Zhang. 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)*, 2002.

[27] Song Jiang, Feng Chen, and Xiaodong Zhang. CLOCK-Pro: An Effective Improvement of the CLOCK Replacement. In *Proceedings of The 2005 USENIX Annual Technical Conference*, 2005.

[28] 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. Ten lessons from three generations shaped google's tpuv4i: Industrial product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture*, 2021.

[29] Anne Kao and Steve R. Poteet. *Natural language processing and text mining*. 2007.

[30] 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. Lessons learned from the chameleon testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*, July 2020.

[31] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong-Sang Kim. On the existence of a spectrum of policies that subsumes the least recently used (lru) and least frequently used (lfu) policies. In *Proceedings of the 1999 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, 1999.

[32] Adam Lerer, Ledell Wu, Jiajun Shen, Timothée Lacroix, Luca Wehrstedt, Abhijit Bose, and Alex Peysakhovich. Pytorch-BigGraph: A Large Scale Graph Embedding System. In *Proceedings of The 2nd Conference on Machine Learning and Systems (MLSys)*, 2019.

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

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

[35] 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. Deep learning recommendation model for personalization and recommendation systems. arXiv:1906.00091, 2019.

[36] Victor F Nicola, Asit Dan, and Daniel M Dias. 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*, 1992.

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

[38] 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. Deep Learning Inference in Facebook Data Centers: Characterization, Performance Optimizations and Hardware Implications. https://arxiv.org/abs/1811.09886, 2018.

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

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

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

[42] Zhenyu Song, Daniel S Berger, Kai Li, Anees Shaikh, Wyatt Lloyd, Soudeh Ghorbani, Changhoon Kim, Aditya Akella, Arvind Krishnamurthy, Emmett Witchel, et al. Learning relaxed belady for content distribution network caching. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2020.

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

[44] Uresh Vahalia. Unix internals: The new frontiers, 1996.

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

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

[47] Hu Wan, Xuan Sun, Yufei Cui, Chia-Lin Yang, Tei-Wei Kuo, and Chun Jason Xue. FlashEmbedding: storing embedding tables in SSD for large-scale recommender systems. In *Proceedings of The 12th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys)*, 2021.

[48] Jizhe Wang, Pipei Huang, Huan Zhao, Zhibo Zhang, Binqiang Zhao, and Dik Lun Lee. 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)*, 2018.

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

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

[51] Mark Wilkening, Udit Gupta, Samuel Hsia, Caroline Trippel, Carole-Jean Wu, David Brooks, and Gu-Yeon Wei. 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)*, 2021.

[52] Xing Xie, Jianxun Lian, Zheng Liu, Xiting Wang, Fangzhao Wu, Hongwei Wang, and Zhongxia Chen. Personalized recommendation systems: Five hot research topics you must know. *Microsoft Research Lab-Asia*, 2018.

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

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

[55] Da Zheng, Xiang Song, Chao Ma, Zeyuan Tan, Zihao Ye, Jin Dong, Hao Xiong, Zheng Zhang, and George Karypis. 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)*, 2020.

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

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