Rabbitail: A Tail Latency-Aware Scheduler for Deep Learning Recommendation Systems with Hierarchical Embedding Storage

Hu Wan
City University of Hong Kong

Yun Huang
City University of Hong Kong

Shuhan Bai
Huazhong University of Science and Technology
City University of Hong Kong

Xuan Sun
The University of Edinburgh

Tei-Wei Kuo
National Taiwan University
Delta Electronics

Chun Jason Xue
Mohamed bin Zayed University of Artificial Intelligence

Abstract

Deep learning-based recommendation systems are critical for many online platforms, but face challenges in managing large embedding tables while meeting strict latency requirements. This paper presents Rabbitail, a novel inference scheduler designed for recommendation systems utilizing hierarchical DRAM-SSD embedding storage. Rabbitail employs a cache-aware approach, classifying inferences into hit and miss categories based on embedding cache lookup results. This allows hit inferences to proceed immediately to top MLPs without waiting for slower SSD retrievals. For miss inferences, Rabbitail implements an on-demand embedding lookup strategy and a reordering mechanism to optimize SSD retrieval. Additionally, it uses dedicated resource allocation for prompt processing of miss queue MLP tasks and employs batch splitting to manage maximum execution times. Evaluations using real-world datasets demonstrate that Rabbitail significantly reduces end-to-end model inference tail latency, achieving a 53.7% lower p99 tail latency compared to the baseline while maintaining throughput.

CCS Concepts

• **Information systems** → **Recommender systems**; • **Software and its engineering** → **Secondary storage**.

Keywords

Recommender systems, Embedding, Solid-state drive (SSD)

ACM Reference Format:
Hu Wan, Yun Huang, Shuhan Bai, Xuan Sun, Tei-Wei Kuo, and Chun Jason Xue. 2025. Rabbitail: A Tail Latency-Aware Scheduler for Deep Learning Recommendation Systems with Hierarchical Embedding Storage. In *The 40th ACM/SIGAPP Symposium on Applied Computing (SAC '25), March 31-April 4, 2025, Catania, Italy*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3672608.3707945

SAC '25, March 31-April 4, 2025, Catania, Italy

ACM ISBN 979-8-4007-0629-5/25/03
https://doi.org/10.1145/3672608.3707945

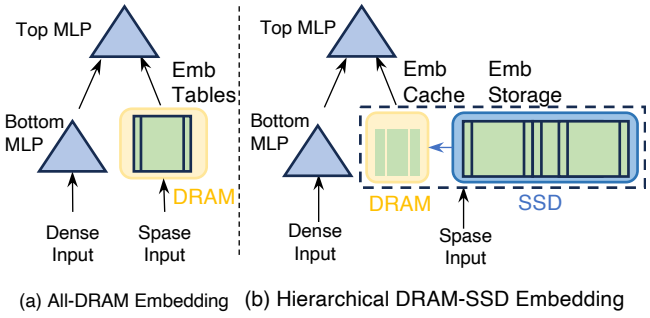

Figure 1: High-level overview of two typical deep learning recommendation model architectures.

1 Introduction

Deep learning-based recommender systems are widely employed by online platforms to deliver personalized recommendations. Modern deep learning recommendation models (DLRMs) employ embedding tables to transform sparse categorical features into dense embeddings. To ensure quick responses to user queries, these tables are typically kept in memory, as depicted in Figure 1(a). Yet, with the expansion of production-scale recommender systems, the size of embedding tables has surged to terabytes. Storing such vast amounts of data entirely in physical memory is impractical [2, 7, 10, 11, 13]. Recent studies propose a hierarchical storage system that combines SSDs with large capacities and a smaller DRAM cache for embeddings [4, 9, 12, 14], as shown in Figure 1(b)). While this solution offers ample storage capacity and good average performance, it also presents the challenge of managing increased tail latency for large-scale recommender systems.

In recommender systems that use a hierarchical DRAM embedding cache and SSD embedding storage, tail latency can be particularly challenging due to the "all-or-nothing" nature of inference operations and the significant difference in access speeds between DRAM cache and SSD storage. To optimize processing and reduce variability in request processing times, existing recommender systems use batching techniques to process multiple requests simultaneously, leveraging parallelism and locality across batched inputs to enhance system throughput and achieve high

resource utilization [5, 6]. However, batching implies that inference requests are not considered complete until all queries within a batch are processed due to the top MLP (multilayer perceptron) layer's dependency on lower layers. Thus, if one inference in a batch fails to hit the embedding cache and needs to access SSD embeddings, the processing time for the rest of the batch may be delayed. Furthermore, as each inference query usually involves several embedding table lookups, if any of these lookups need to access SSD instead of DRAM, it can cause significant latency and block the subsequent top MLP for the entire batch.

To address this issue, we propose Rabbitail[1], a **tail** latency-aware inference scheduler for large-scale recommender systems with hierarchical DRAM embedding cache and SSD embedding storage. While we acknowledge that completely eradicating variability in tail latency is unfeasible due to the vast scale and intricacy of contemporary recommendation services, our objective remains to minimize the extent of latency distribution tails for recommendation inferences. The key insight of Rabbitail is that inferences should be treated differently based on their embedding cache lookup results to mitigate tail latency issues. Accordingly, the inference dispatcher utilizes the embedding vector and block caches to categorize recommendation inferences into two distinct groups: those with cache hits and those with misses. Inferences that hit the cache are immediately processed by the top MLP layer. Conversely, inferences that miss necessitate an SSD lookup to retrieve missing embeddings before they can proceed further. Rabbitail employs an on-demand embedding table lookup strategy and a tail latency-aware reordering mechanism to optimize the retrieval process of SSD embeddings. These strategies aim to decrease long-tail latency by prioritizing and grouping miss inferences based on the number of embeddings they require and the waiting time. Rabbitail optimizes the execution of MLP tasks by proactively allocating specific CPU cores and assigning dedicated worker threads with fixed CPU affinity to handle the miss queue processing exclusively. As soon as the missing embeddings are retrieved from the SSD, the corresponding MLP tasks are processed with minimal delay. It also manages the maximum execution time by dividing large batches of MLP tasks.

Our evaluation of Rabbitail with state-of-the-art recommendation models shows that Rabbitail achieves 53.7% lower end-to-end model inference latency compared to the baseline while maintaining the same throughput.

2 Background and Motivation

2.1 Hierarchical Embedding Architecture

A deep learning-based recommendation model typically consists of a bottom MLP, top MLP, embedding layer, and feature interaction layer [5, 6, 8, 15], as illustrated in Figure 2. The model predicts the click-through rate (CTR) using a combination of dense and sparse features as inputs. Dense features are continuous inputs that undergo processing with MLP layers. Sparse features are transformed into a dense representation by looking up embedding tables in the embedding layer. An embedding table is essentially a lookup table where each row corresponds to a category for a sparse feature and contains a vector of floating-point values. When given an index

[1] The name "Rabbitail" is inspired by the rabbit, known for its short tail. This symbolizes our aim to minimize tail latency in the recommender system.

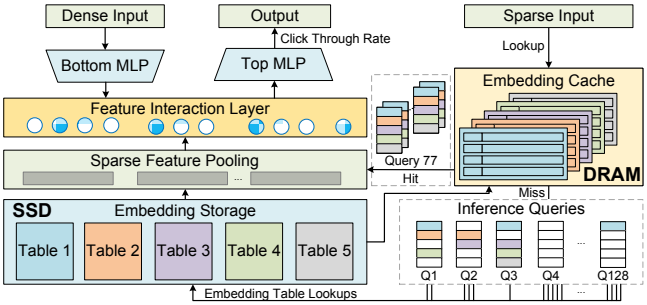

Figure 2: Hierarchical embedding cache and storage.

set, an embedding table lookup reads several vectors from the table. These vectors are then combined into a single vector using element-wise operations such as additions and averages before being concatenated with dense features via feature interaction. Finally, the outputs of the bottom MLP and embedding layers go through the top MLP layers to produce a single value representing the probability of a certain event such as predicted CTR.

Recent studies proposed methods to manage the challenge of large embeddings in recommendation systems by using SSDs for storage and DRAM for caching [4, 9, 12, 14], as illustrated in Figure 2. The hierarchical DRAM-SSD embedding storage system diverges from an all-DRAM embedding storage model due to its layered structure and varying access speeds. The speed of accessing the DRAM cache surpasses that of the SSD storage, creating a pronounced disparity between cache hits and misses. When inference queries hit all embedding caches (e.g., Query 77), it is crucial to apply top MLPs promptly rather than waiting for slow retrieval of embeddings from the SSD for miss queries. For cache-missed inferences, there are variations - some only miss one table (e.g., Query 3), while others miss all tables (e.g., Query 4). Accessing embeddings from the SSD is a time-consuming process that is proportional to the quantity of embeddings. When several embedding tables are queried and an increased number of embeddings are extracted from each, this retrieval duration extends accordingly. Consequently, this leads to longer overall inference times and increases vulnerability to long-tail latency issues.

2.2 Challenges and Opportunities

To gain a better understanding of hierarchical embedding cache and storage in recommendation inference, we conducted studies to identify unexplored challenges and opportunities.

Our analysis begins with a comparison of latency distributions. Figure 3 shows that using all-DRAM embeddings, assuming unlimited memory, achieves peak performance with most latencies under 10ms. In contrast, the hierarchical approach, which combines DRAM and SSD, generally stays below 20ms but exhibits higher p99 tail latency up to 63.5ms. This increases both average and tail latency, highlighting a key challenge in maintaining performance.

Figure 4 explores the impact of missing inferences on batch latency, with a default batch size of 128. As the number of cache-missing inferences increases, latency rises significantly. While larger batches in all-DRAM systems enhance throughput through efficient SIMD utilization and reduced function call overhead, this advantage

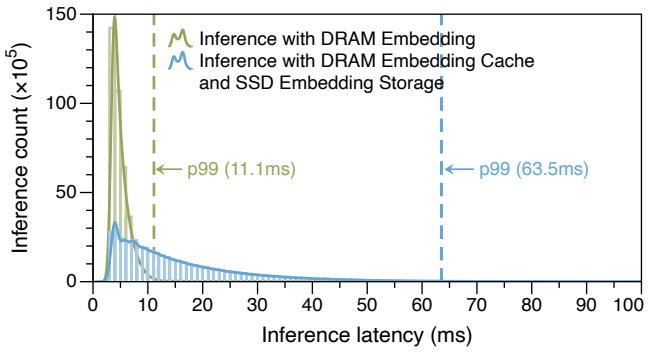

Figure 3: Comparison of latency distribution for different embedding storage.

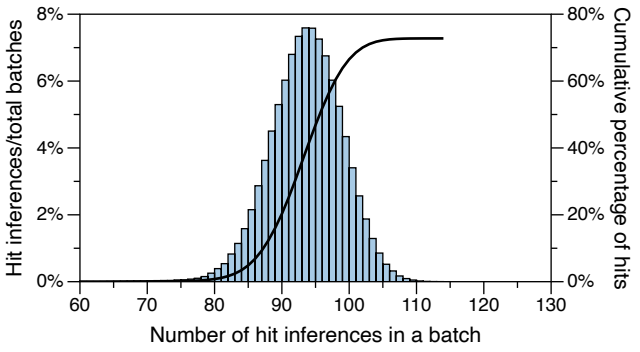

Figure 5: Distribution of batches with a different number of hit inferences.

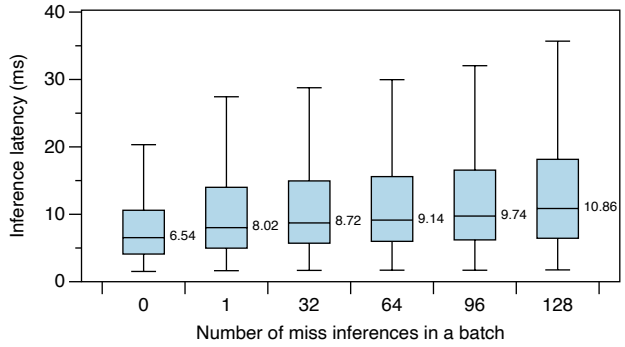

Figure 4: Latency of a batch of inferences with different numbers of miss inferences.

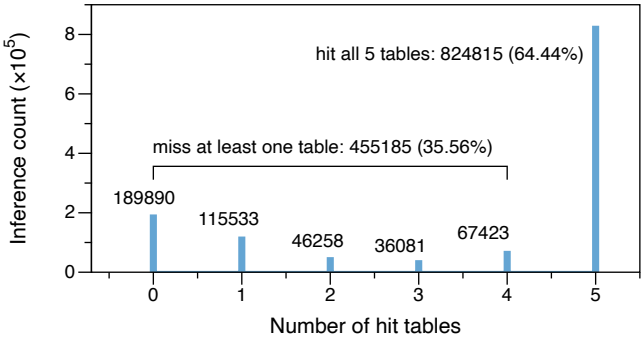

Figure 6: Embedding cache lookup results.

Table 1: Comparison of hit ratio for vector-based and block-based embedding cache.

ID	Table Size	Cache Size	Hit Ratio	
			Vector-based	Block-based
T2	1237MB	40MB	69.79%	88.94%
T3	269MB	40MB	92.60%	97.44%
T11	1019MB	40MB	74.01%	89.77%
T15	667MB	40MB	81.74%	92.26%
T20	860MB	40MB	77.25%	90.61%

diminishes in systems with hierarchical caches. Strict service level agreements (SLAs) in recommendation systems impose practical limits on batch sizes, making current batching schemes sub-optimal for mixed embedding lookup queries and leading to inefficiencies. Even a single missed inference can adversely affect the entire batch's performance, akin to Liebig's law of minimums, where slow SSD lookups act as bottlenecks.

Figure 5 demonstrates the distribution of cache hits per batch. Most batches achieve a hit count between 90 and 100, indicating high cache efficiency in handling queries without resorting to slower SSD storage. However, current scheduling does not optimize based on caching efficacy, treating all queries uniformly despite potential speed gains for cached data.

The utilization of cached embedding tables is illustrated in Figure 6. It reveals that 64.44% of inferences accessed all five caches, while 35.56% missed at least one, with a significant portion missing all five caches. This indicates room for improvement in cache management strategies.

Table 1 presents the hit ratio of the embedding cache managed at vector and block granularity. For the five largest embedding tables, each is assigned a cache size ranging from 3% to 15% of the table size. The block-based cache improves the hit rate by 4.84% to 19.15%, with an average increase of 12.73%. This suggests the potential to leverage temporal reuse and enhance spatial locality in block accesses. In recommender systems, embeddings are typically retrieved as vectors and managed in the DRAM cache, but when stored on SSD, they are accessed at the block granularity.

In summary, increased latency is a key issue when comparing all-DRAM to hierarchical DRAM-SSD methods. Cache misses significantly affect batch inference latency under strict SLAs limiting batch sizes. Despite high cache hit rates, current scheduling does not optimize caching efficiency. Many inferences miss at least one cache, suggesting potential for improved management. Addressing latency and refining cache strategies are vital for enhancing recommendation systems' performance and efficiency.

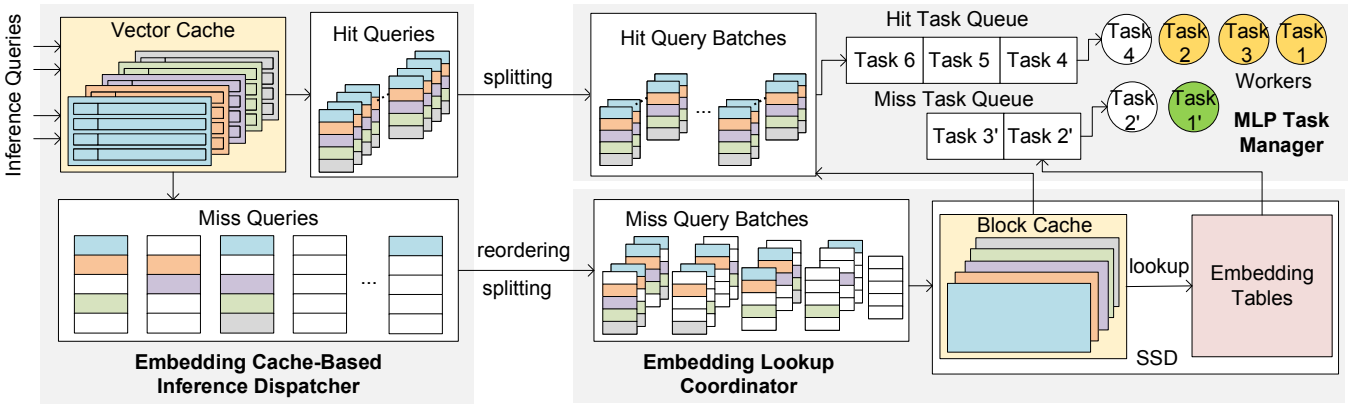

Figure 7: Rabbitail design overview.

3 Design

3.1 Design Overview

We introduce Rabbitail, an inference scheduler designed for large-scale deep learning recommender systems that use hierarchical DRAM embedding cache and SSD embedding storage. This scheduler is tailored to minimize tail latency and reduce SLA violations for inference requests. At its core, Rabbitail operates on the principle of treating each recommendation inference in a batch as a separate entity, independent of others. This approach allows for flexible scheduling of inferences, which can then be consolidated into smaller batches based on their respective embedding access patterns. Figure 7 depicts the design overview of Rabbitail. Rabbitail consists of three primary components:

1. **Cache-based inference dispatcher** classifies inferences into two categories: hit inferences and miss inferences. These categories are handled differently.
2. **Embedding lookup coordinator** further prioritizes and regroups miss inferences based on the number of embedding they require.
3. **MLP task manager** applies top MLP operations to both hit inferences and returned miss inferences that retrieve embeddings from the SSD embedding table.

3.2 Cache-Based Inference Dispatcher

The inference dispatcher, leveraging the embedding vector and block caches, distinguishes between service classes for cache hit inferences (for which embeddings are available in the embedding cache) and cache miss inferences (for which embeddings are not in the embedding cache).

3.2.1 Embedding Vector and Block Cache. The use of an embedding vector cache, as depicted in the top left corner of Figure 7, has been shown to have a high hit ratio by caching frequently accessed embeddings vectors. Most lookup operations retrieve relevant embedding vectors from the host-side cache, which improves performance by reducing slower device access times. Our motivational experiments indicate that we can optimize spatial locality from block accesses. We have enhanced this by introducing an additional layer of embedding cache using the SSD's internal DRAM to cache embeddings at a granularity of 4KB blocks, as depicted in the bottom right corner of Figure 7. In situations where the cache is full, we utilize the traditional LRU (Least Recently Used) caching algorithm to discard those vectors and blocks that have not been accessed recently.

3.2.2 Inference Dispatching. Figure 7 illustrated the progress of inference dispatching. Rabbitail divides a set of inference queries based on embedding vector cache hit/miss and assigns the corresponding top MLP tasks to the priority queue. When Rabbitail receives a batch of queries for inference, it retrieves and loads the corresponding embedding vectors from the embedding vector cache. The process of completing a single recommendation inference query necessitates a thorough search of all embedding tables, which inherently involves examining all SSD embedding table caches. Given that the bottom MLP is already completed when the cached embedding vector returns, the top MLP is triggered as soon as an inference query successfully accesses all caches. Rabbitail labels these successful inference queries as 'hit' queries and queues their corresponding higher MLP tasks in the hit task queue. Conversely, if any caches are missed, these inference queries are designated as 'miss' queries and are redirected to the SSD for further embedding lookup. Any 'miss' queries are then scrutinized to ascertain whether the missed embeddings are present within the embedding block cache. If located, these inferences are reclassified as 'hit' queries, and their corresponding higher MLP tasks are added to the hit task queue. The newly fetched embedding vectors loaded from SSD are stored inside the embedding cache and sent back to upper layers for further processing.

3.3 Embedding Lookup Coordinator

Retrieving missing embeddings from SSD results in large variances in query results, leading to high tail latency. These variances mainly come from the fact that the number of embeddings missed by each inference is not always consistent. Some inferences may miss only one embedding table, while others may miss all embedding tables. To this end, an embedding lookup coordinator is developed to handle miss inferences. It employs on-demand embedding table lookup strategy and reordering mechanism to optimize the retrieval process of embeddings stored on SSD.

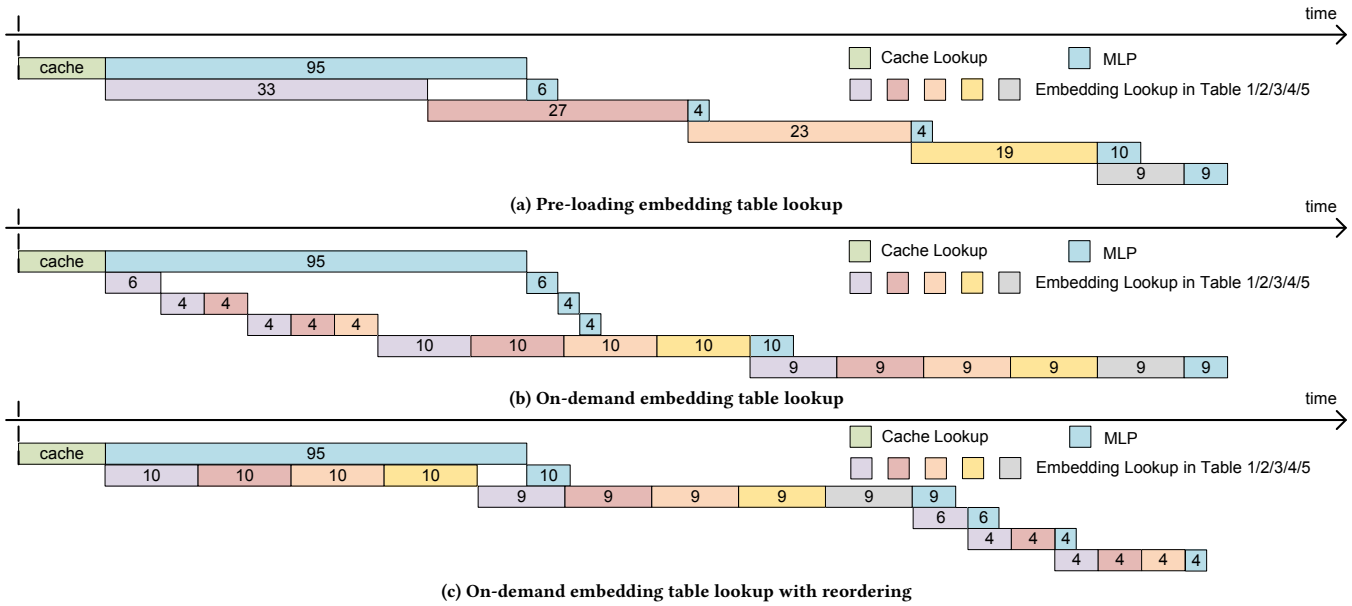

Figure 8: Comparison of embedding table lookup strategies.

3.3.1 Pre-loading Embedding Table Lookup. When an embedding cache lookup fails, misses are batched based on the number of distinct tables they could not find. Multiple queries often miss embeddings from the same table as shown in Figure 2. For example, Query 1 and 2 fail to retrieve data from overlapping sets of tables. Missed embeddings are retrieved by querying each table sequentially—a method we term pre-loading embedding table lookup. Queries missing data from only one table can proceed directly to applying the top MLP after retrieval, while those missing multiple tables must wait until all their required embeddings are fetched. This sequential access ensures that more queries become ready for progression at every step until all necessary embeddings are obtained.

Figure 8a shows the pre-loading process for embedding table lookup. Initially, 95 inference queries accessed all five caches and immediately executed top MLP operations. Subsequent missing embeddings were fetched sequentially from respective tables. Specifically, out of 33 inferences with missing embeddings from table 1, only six exclusively required this table before proceeding with their MLP operations. As queries moved through tables 2 to 4, more inferences obtained necessary embeddings until the final nine retrieved theirs from table 5, completing the batch.

A major issue with the pre-loading embedding table lookup strategy is its tight coupling of inferences during SSD embedding lookups due to bulk retrieval. This method retrieves all missing embeddings at once, even if some aren't immediately needed because data from other tables is absent. For instance, although only four inferences required immediate embeddings from table 2, they had to wait for an additional 23 retrievals due to this collective fetching approach, resulting in a total of 27 embedding lookups and unnecessarily delaying execution times.

3.3.2 On-demand Embedding Table Lookup. To resolve the challenge of pre-loading embedding table lookups strategy, we propose an on-demand embedding table lookup strategy. This method involves fetching only the necessary embeddings from relevant tables as needed, instead of loading entire tables at once. Consequently, each table is accessed multiple times to acquire specific embeddings. The advantage of this technique is that it allows for rapid retrieval of required embeddings even if one or a few tables are missed during inference, facilitating a swift transition to subsequent MLP stages. The time taken for embedding lookups on SSDs increases with the number of embeddings fetched; therefore, dividing missed embeddings into several smaller queries reduces both the size and frequency per query and minimizes delays in batch processing.

Figure 8b demonstrates our proposed on-demand embedding table lookup strategy. For example, six inferences that required only the embeddings from table 1 were able to swiftly acquire them and move on to MLP operations. Meanwhile, four inferences that needed data from both tables 1 and 2 had to sequentially fetch this information before they could proceed with additional processing steps. When multiple tables are implicated, the inferences efficiently request the requisite embeddings from each relevant table as needed, thus avoiding overburdening any individual query.

3.3.3 On-demand Embedding Table Lookup with Reordering. We have noticed that the order in which we perform lookups on embedding tables can greatly affect tail latency. This is because the number of embeddings to be retrieved during inference varies depending on the sequence of table lookups. Retrieving embeddings from SSDs is done sequentially due to hardware limitations, resulting in a queue for multiple requests for lookup. Therefore, we must consider the waiting time for these requests as well. Inferences that stall by more embedding lookups have a high chance of becoming the "tail" end of the distribution of latency. With this in mind, the idea is to minimize the number of miss inferences that have longer waiting times in the end. We take into account the number

of missed embedding tables and the number of missed embedding in each table. We prioritize inferences that lack a greater number of embeddings to retrieve embeddings first, while those missing fewer embeddings are scheduled subsequently. As a trade-off, we choose to sacrifice some average waiting time to enable inferences that require more time to execute as soon as possible. Because inferences that miss few embedding tables need to wait for longer times before their embeddings can be retrieved. Figure 8c shows the timeline of on-demand embedding table lookup with reordering mechanism. Compared to other strategies, fewer inferences will be executed for the operation of retrieving embeddings at the end. Currently, only four inferences are executed, whereas nine were executed previously.

3.4 MLP Task Manager

Hierarchical embedding cache and storage in recommendation systems introduce complex MLP processing, resulting in variable response times and a risk of long delays compared to traditional DRAM-only architectures. This variability arises firstly because MLP operations depend on whether they access data from the cache or must retrieve it from SSDs when the cache is missed. When embeddings are directly available in the cache, MLP operations proceed swiftly; otherwise, they're delayed as embeddings are fetched batch by batch from SSDs. Secondly, the inconsistency in batch sizes for MLP operations stems from fluctuating hit rates of the embedding cache across different batches. Within a single inference batch, those that miss are split into sub-batches of various sizes, which can further complicate processing times—especially since hits typically outnumber misses due to efficient caching.

To mitigate these latency issues, we propose a tail latency-aware MLP task manager with two key strategies: dedicated resource allocation for prompt processing of miss queue MLP tasks; and managing maximum execution time by dividing larger batches into smaller ones for processing. The task manager is illustrated in Figure 7, located at the upper right corner of the diagram.

3.4.1 Dedicated Resource Allocation Strategy. Rabbitail employs a dedicated resource allocation strategy to effectively manage tasks that experience cache misses, ensuring minimal latency and maintaining system responsiveness. The framework differentiates between two task queues based on cache lookup results: the hit task queue for tasks successfully retrieving embeddings from DRAM, and the miss task queue for tasks requiring data from SSD storage. This separation allows us to tailor resource allocation strategies to the specific needs of each task type.

The primary latency challenge for cache miss tasks stems from the time taken to access data from SSDs. However, once data retrieval is complete, immediate processing is crucial to minimize overall latency. By dedicating specific CPU cores and assigning worker threads with fixed CPU affinity to handle the miss queue, Rabbitail ensures these tasks are processed without delay. This setup prevents them from competing with other tasks for CPU resources, thus avoiding additional scheduling delays and reducing variability in response times.

To implement this strategy, we reserve specific CPU cores exclusively for processing miss tasks. Worker threads are bound to these cores through fixed CPU affinity, reducing context-switching overhead and ensuring consistent processing speeds. Additionally, the system proactively manages these resources, anticipating cache misses and preparing the necessary computational power in advance. This proactive approach ensures that once data is retrieved, the tasks can be processed immediately.

3.4.2 Threshold-based Batch Splitting. The size of successful cache hit inferences can be considerably larger than unsuccessful ones, owing to the high hit ratio of the embedding cache. Generally, using larger batch sizes can enhance throughput by optimizing hardware SIMD architecture utilization and minimizing software overhead. However, when considering tail latency, increased batch sizes add computational load and may lead to slower response times. Real-time recommendation systems must balance this against SLA requirements when selecting an optimal batch size.

To address this challenge, we propose dividing large batches into smaller segments once they exceed a certain threshold. This strategy involves identifying batches prone to contributing to tail latency and breaking them down for processing while leaving less problematic batches intact. The subdivided batches are then processed concurrently in a task pool, which helps spread out the computational demands more evenly among threads and reduces the risk of congestion from several large batches. The effectiveness of this method depends on setting an effective threshold, which is determined by monitoring the ongoing 99^{th} percentile performance metric. We start with a threshold equal to the initial inference request's batch size and adjust it dynamically over time based on observed performance trends.

4 Implementation

We implemented the top and bottom layers of the multi-layer perceptron (MLP) using LibTorch, Pytorch's C++ library. To manage the embedding vector cache efficiently, we utilized FlashEmbedding's interface for embedding lookups [12]. We designed an embedding block cache within the SSD's onboard DRAM to avoid typical SSD processing bottlenecks. The cache is distributed across 16 individual chips, creating 16 separate caching areas. This arrangement ensures that blocks are stored in their respective chip-related regions. Each area can hold up to 16 blocks or four pages from its designated chip.

Rabbitail handles inference requests asynchronously, using a separate thread to dispatch them into an inference queue. It generates these queries in batches based on a Poisson distribution, which is used to simulate real-world traffic patterns of recommendation services in production data centers [5]. Rabbitail processes these requests using a thread pool that allows for non-blocking task execution in FIFO order. Upon receiving an inference request, Rabbitail concurrently processes the bottom MLP and embedding layer. If all embeddings are cached, it proceeds with the top MLP; otherwise, it handles cache misses by sending SSD embedding lookup requests to its dedicated thread pool. A priority queue manages these tasks based on their urgency, with worker threads executing tasks from the front of the queue before moving onto subsequent ones.

To prevent interference, we allocate specific CPU cores to compute-intensive MLP tasks and to embedding lookup, ensuring they remain on their designated cores. Additionally, we assign dedicated worker threads with fixed CPU affinity to exclusively handle the miss queue processing.

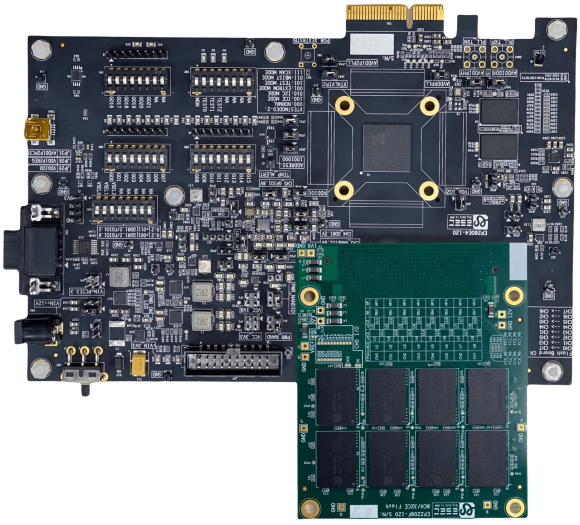

Figure 9: Hardware prototype of SSD embedding storage.

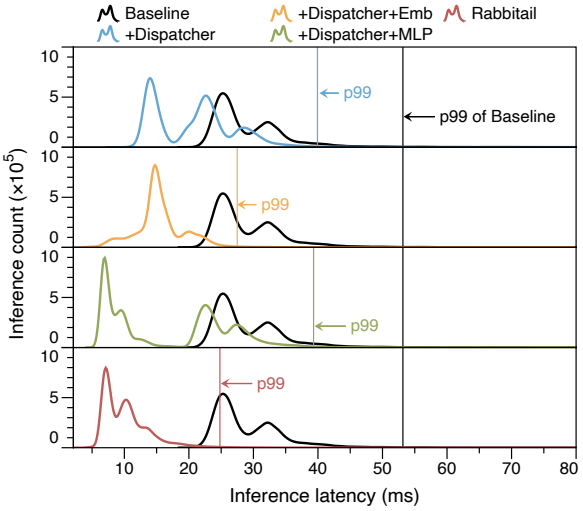

Figure 10: Latency distribution of Baseline and proposed approaches.

Table 2: Hardware specification of SSD embedding storage.

Item	Description
Host Interface	PCIe Gen.3 x 4
Protocol	NVMe 1.2
Module Capacity	477GB
SSD Architecture	8 channels, 8 ways, 2 cores
Storage Medium	SLC NAND flash, MLC NAND flash, and TLC NAND flash
Mapping Region	64MB
Max DDR size	4GB

5 Evaluation

5.1 Experimental Setup

Testbed. We evaluate Rabbitail on a server equipped with an eight-core Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz and 64GB DRAM. We utilize the Pipette's YS9203 NVMe SSD development board [1] to store embedding tables. Pipette is a framework that leverages the byte-accessibility of modern SSDs through PCIe interconnection, allowing programs to seamlessly access data across the SSD and host DRAM with little overhead, and significantly reducing read amplification. We modify Pipette's SSD firmware and extend the NVMe command set to support fine-grained reads from embedding tables in an embedding vector granularity. The details of our prototype can be found in Figure 9 and Table 2.

Datasets and models. We evaluated Rabbitail's effectiveness using publicly available datasets from Criteo AI Labs [3]. The dataset includes 26 embedding tables; we allocated the five largest tables (98.3% of total size) to SSD storage, keeping the smaller tables in main memory. We used DLRM [6] as our primary model due to its accurate mimicry of industrial models' execution flows.

Baselines. Our primary baseline, FlashEmbedding [12], supports hierarchical embedding systems and processes requests synchronously, causing inference delays until all operations complete. We modified it to a non-blocking mode for fair comparison, denoted as ***Baseline*** in our recommender system prototype. Enhancements include ***+Dispatcher***, using an embedding cache-based dispatcher; ***+Dispatcher+Emb*** and ***+Dispatcher+MLP***, which optimize SSD embedding lookups and MLP executions respectively; and ***Rabbitail***, which integrates all optimizations into the Baseline.

Inference requests follow a Poisson distribution, occurring on average every 30 milliseconds. We present the results of tail latency and throughput using real-world workloads when comparing Rabbitail with alternative methods.

5.2 Tail Latency

We start by analyzing the latency differences between the Baseline and our proposed methods. The latency distributions for both the baseline and Rabbitail's techniques are shown in a histogram, as seen in Figure 10. In this graph, the X-axis represents recommendation inference latency, and the Y-axis indicates their frequency. Additionally, a vertical line marks the p99 latency for each method. Based on this data, we make the following observations.

First, it is evident that all methods typically adhere to a log-normal distribution pattern. Notably, each method's distribution displays a bimodal characteristic with one pronounced peak and another less significant one. This dual-peak phenomenon is especially marked in the +Dispatcher and +Dispatcher+MLP methods. This observation underscores the significant influence of embedding cache hits and misses on the distribution shape.

Second, Rabbitail significantly outperforms the Baseline in terms of both average and tail latency. While the Baseline typically exhibits a latency around 25ms for most inferences, it extends up to 54ms for the slowest 1%. In contrast, Rabbitail consistently achieves inference times below 10ms, with its p99 latency reduced by an impressive 53.7%, bringing it down to just 25ms.

Third, all techniques employed in Rabbitail contribute to the lower tail latency. The embedding cache-based inference dispatcher technique reduces p99 latency by 26%. This efficiency primarily results from Rabbitail's approach of distinguishing between cache hits

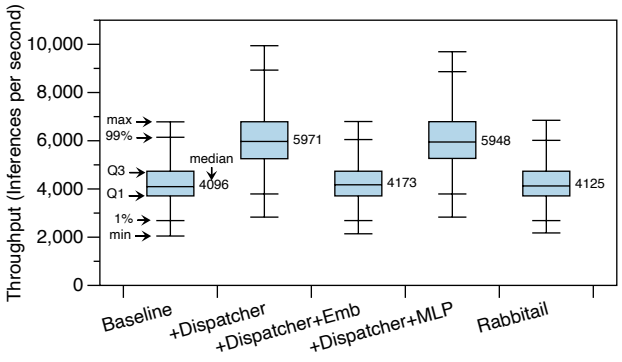

Figure 11: Throughput of Baseline and proposed approaches.

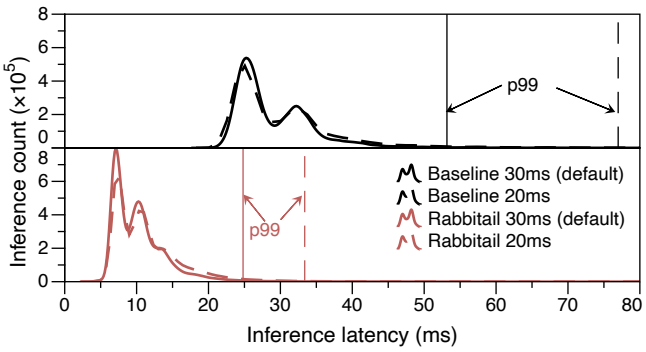

Figure 12: Latency distribution of Baseline and Rabbitail.

and misses during inference, enabling immediate processing of top MLPs upon a cache hit without waiting for slower SSD embedding lookups to finish. Our motivational experiments with the Criteo dataset show a high hit ratio, highlighting substantial opportunities for accelerated execution. In contrast, the conventional baseline method does not differentiate based on embedding cache status and processes all inferences uniformly, leading to extended delays even when embeddings are already cached. Additionally, our refined approach to embedding lookup further reduces p99 tail latency by 22%. This improvement stems from an on-demand retrieval method that limits the number of queried embeddings and an optimized sequence of lookups that shortens waiting times during inference processes. Enhancements in MLP technology have a minimal impact on reducing tail latency—only a 3% reduction—as the primary bottleneck remains with slow SSD embedding retrievals rather than CPU performance during MLP computations.

5.3 Throughput

We also evaluate the throughput of Baseline and our proposed Rabbitail approaches. Figure 11 shows the throughput in terms of inferences per second for each approach. We use a background thread to periodically measure performance with one second interval time. The box plot depicts the distribution of throughput and skewness. For each approach, we present various metrics: the minimum throughput, 1st percentile, first quartile (Q1), median, third quartile (Q3), 99th percentile, and the maximum throughput. Our findings indicate that Rabbitail and +Dispatcher+EMB perform on par with Baseline in terms of throughput. However, the +Dispatcher and +Dispatcher+MLP methods show a marked improvement in inference speed – averaging an increase from 4209 to 6030 inferences per second – which translates to a significant enhancement of approximately 43.26%.

The speed at which inferences are made largely depends on the time it takes to retrieve embedding tables from SSDs. Both the +Dispatcher and +Dispatcher+MLP techniques employ a method that pre-loads vectors in these tables, resulting in lengthy load times but fewer SSD queries. During this loading phase, top MLPs that hit the embedding cache can be immediately forwarded to the MLP Task Manager for execution without waiting for prior requests to finish. This strategy enhances throughput by allowing more top MLPs to process concurrently as new inference requests come in, despite potential fluctuations in performance.

On the other hand, +Dispatcher+Emb and Rabbitail use an on-demand lookup strategy that breaks down all missed embeddings into smaller batches. By querying fewer embeddings at once, this approach minimizes delays caused by processing large batches of data during inference sessions. Although this method only marginally increases overall throughput, it effectively reduces the likelihood of long-tail associated with extensive embedding retrieval tasks.

5.4 Sensitivity Analysis

5.4.1 Impact of Query Arrival Delay. We examine the impact of reduced inference query arrival delays on Rabbitail's performance. Figure 12 depicts the latency distribution comparison between Baseline and Rabbitail under varied inference query arrival delays. By shortening the average interval from 30 milliseconds to 20 milliseconds, we observed that a quicker arrival rate increases system workload. This can lead to competition for computational resources and potentially extend processing times for inference requests. The slower SSD embedding lookup is particularly affected, resulting in increased p99 tail latency for both baseline and Rabbitail methods. Figure 13 illustrates how throughput changes when the query arrival delay decreases. Our findings show that while the baseline method experienced a 42.59% rise in p99 tail latency, Rabbitail saw only a 32% increase—demonstrating its superior performance under these conditions.

5.4.2 Impact of Recommendation Model Size. We investigated the impact of different sizes for Rabbitail's recommendation model, focusing on three variations: small, medium, and large, based on trainable parameters. The latency distribution of Baseline and Rabbitail with different MLP sizes is presented in Figure 14. Our analysis revealed that as we scaled down the top MLP model size, both average latency and p99 tail latency tended to decrease. This trend aligns with expectations since the MLP constitutes only a minor portion of the entire execution process. However, it is important to highlight that reducing the MLP size has an additional benefit in our system design. A smaller MLP enhances the speed at which hit embedding cache inference occurs. Consequently, this acceleration significantly reduces inference latency overall, enabling quicker responses to recommendation queries.

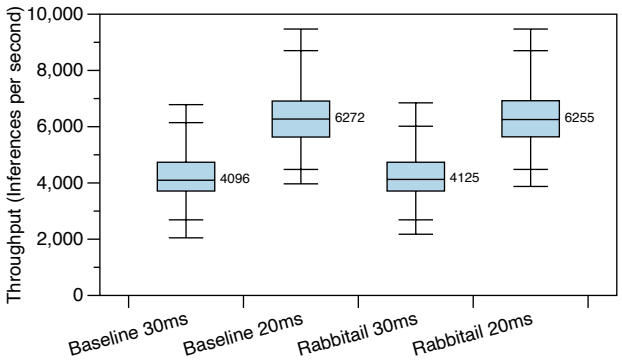

Figure 13: Throughput of Baseline and Rabbitail.

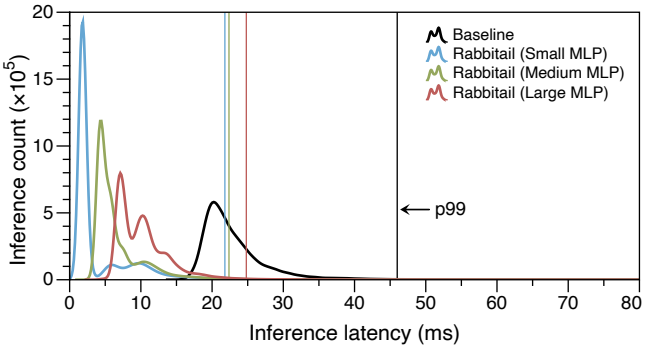

Figure 14: Latency distribution of Baseline and Rabbitail with varying MLP sizes.

5.5 Discussion

Our goal in this paper is to minimize latency distribution tails for recommendation inferences, improving response times and user experience, while acknowledging that eliminating all variability may not be possible. This paper focuses on enhancing recommendation models that rely heavily on embeddings by integrating hierarchical embedding caches and storage systems. MLP-dominated models can benefit from specialized devices such as GPUs. However, as model dimensions—width, depth, or embedding layer size—expand, they frequently exceed the memory capabilities of production servers and even more so the constrained memory capacity available on GPUs. The concepts discussed here have potential applications for large-scale recommendation systems that utilize GPU technology, a prospect we aim to explore in future research.

6 Conclusion

This paper presents Rabbitail, an inference scheduler for deep learning-based recommendation systems with hierarchical architectures utilizing DRAM embedding cache and SSD embedding storage. Rabbitail optimizes inference by differentiating cache hits from misses, allowing faster processing of cached data while managing slower SSD fetches efficiently. Evaluation shows that Rabbitail demonstrated a reduction in end-to-end model inference latency compared to the baseline without compromising throughput.

References

[1] Shuhan Bai, Hu Wan, Yun Huang, Xuan Sun, Fei Wu, Changsheng Xie, Hung-Chih Hsieh, Tei-Wei Kuo, and Chun Jason Xue. 2022. Pipette: Efficient Fine-Grained Reads for SSDs. In *Proceedings of the 59th ACM/IEEE Design Automation Conference* (San Francisco, California) *(DAC '22)*. Association for Computing Machinery, New York, NY, USA, 385–390.

[2] Yujeong Choi, Jiin Kim, and Minsoo Rhu. 2024. ElasticRec: A Microservice-based Model Serving Architecture Enabling Elastic Resource Scaling for Recommendation Models. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, Los Alamitos, CA, USA, 410–423.

[3] Criteo. 2014. Kaggle Display Advertising Challenge Dataset. https://labs.criteo.com/2014/02/kaggle-display-advertising-challenge-dataset/

[4] 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 Machine Learning and Systems (MLSys)*, Vol. 1. Systems and Machine Learning Foundation, Indio, CA, USA, 40–52.

[5] Udit Gupta, Samuel Hsia, Vikram Saraph, Xiaodong Wang, Brandon Reagen, Gu-Yeon Wei, Hsien-Hsin S. Lee, David Brooks, and Carole-Jean Wu. 2020. DeepRecSys: A System for Optimizing End-To-End At-Scale Neural Recommendation Inference. In *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, Los Alamitos, CA, USA, 982–995.

[6] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, David Brooks, Bradford Cottel, Kim 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 *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, Los Alamitos, CA, USA, 488–501.

[7] Kim Hazelwood, Sarah Bird, David 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 *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, Vienna, Austria, 620–629.

[8] Liu Ke, Udit Gupta, Benjamin Youngjae Cho, David Brooks, Vikas Chandra, Utku Diril, Amin Firoozshahian, Kim 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 *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, Valencia, Spain, 790–803.

[9] Minsub Kim and Sungjin Lee. 2020. Reducing Tail Latency of DNN-Based Recommender Systems Using in-Storage Processing. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys)*. ACM, Tsukuba, Japan, 90–97.

[10] 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* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 281–294.

[11] Xuan Sun, Hu Wan, Qiao Li, Chia-Lin Yang, Tei-Wei Kuo, and Chun Jason Xue. 2022. RM-SSD: In-Storage Computing for Large-Scale Recommendation Inference. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, Los Alamitos, CA, USA, 1056–1070.

[12] 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* (Hong Kong, China) *(APSys '21)*. Association for Computing Machinery, New York, NY, USA, 9–16.

[13] 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)*. ACM, London, United Kingdom, 839–848.

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

[15] 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)*. ACM, Lausanne, Switzerland, 717–729.