DECK: Experiences on Delta Checkpointing for Industrial Recommendation Systems

Xin Gao Sibasish Acharya Sihui Han Yongxiong Ren Yanli Zhao Liang Luo Chucheng Wang Pradeep Fernando Saurabh Mishra Siqi Yan Yicong Du Elzbieta Krepska Intaik Park Min Ni Qunshu Zhang Shen Li
{xingao,sibasish,shenli}@meta.com
Meta Inc.

ABSTRACT

In large-scale industrial recommendation systems, model checkpoints are instrumental in maintaining training goodput and numerical correctness during system failures and job preemptions. The increasing prevalence of multi-terabyte models has rendered frequent regular model checkpoints impractical, resulting in substantial lost progress when recovering from failures. As model sizes continue to grow, researchers and practitioners are compelled to investigate more efficient and scalable solutions. This paper presents DECK, a novel approach to delta model checkpointing designed for real-world industrial systems. Specifically, DECK focuses on extracting delta states with near-zero overhead, staging and streaming delta checkpoints without interrupting the training process, and merging delta checkpoints in an optimal and decoupled manner. Experimental results demonstrate that DECK achieves a 12-fold increase in checkpoint frequency while maintaining negligible impact on training throughput, thereby attaining state-of-the-art (SOTA) production performance.

PVLDB Reference Format:
Xin Gao, Sibasish Acharya, Sihui Han, Yongxiong Ren, Yanli Zhao, Liang Luo, Chucheng Wang,Pradeep Fernando, Saurabh Mishra, Siqi Yan, Yicong Du, Elzbieta Krepska, Intaik Park, Min Ni, Qunshu Zhang, Shen Li. DECK: Experiences on Delta Checkpointing for Industrial Recommendation Systems. PVLDB, 18(12): 4978 - 4990, 2025.
doi:10.14778/3750601.3750621

1 INTRODUCTION

Modern recommendation systems [6, 14–16, 24, 25] deployed in industrial settings operate at unprecedented scales, processing vast volumes of data across distributed computational infrastructure to serve billions of users worldwide. These systems rely on increasingly complex machine learning models that have grown from gigabytes [5] to multiple terabytes in size [9, 13, 23]. As these models expand in complexity and scale, they present significant challenges for traditional training resilience mechanisms, particularly in the domain of model checkpointing which functions as the primary safeguard against various forms of system disruptions, including hardware failures, data anomaly, and job preemptions that occur during resource reallocation. Without effective checkpointing strategies, such interruptions can result in substantial loss of computational work, negatively impacting operational efficiency and recommendation quality.

The conventional approach to model checkpointing involves periodically saving the complete model state to persistent storage. While this methodology provides comprehensive recovery capabilities, it becomes increasingly burdensome as model sizes surpass terabyte thresholds. The time required to serialize, transfer, and persist these massive model states introduces significant training inefficiencies and creates operational bottlenecks. In Large Language Model (LLM) training environments [2, 20], system architects have implemented rapid in-memory checkpointing solutions [21] to maintain or enhance checkpointing frequency. However, these methodologies typically prove incompatible with recommendation systems due to fundamentally different resource allocation profiles. Specifically, recommendation systems typically rely on massive, sparse embedding tables alongside much smaller, compute-intensive layers to handle high workloads at low cost [11, 13, 23, 27]. In such environments, CPU memory and even SSD storage are heavily constrained by concurrent components managing these embeddings, leaving in-memory checkpoints impractical. As a consequence, practitioners usually extend checkpoint intervals to maximize effective training throughput, thereby increasing system vulnerability to progress losses when failure occurs.

This growing tension between checkpoint frequency and training throughput has emerged as a critical challenge in industrial recommendation systems. The ideal solution would enable frequent checkpointing to minimize potential work loss while simultaneously maintaining high training throughput. However, conventional approaches force an undesirable trade-off between these objectives, leading to sub-optimal training efficiency and resilience. In response to these challenges, we present DECK, a production-grade checkpointing solution specifically designed to address the requirements of industrial-scale recommendation systems. DECK introduces a fundamentally different approach to checkpoint models through three key innovations:

- "Zero-Cost" Delta Tracking: DECK implements specialized mechanisms to extract only the modified portions of model state with near-zero computational overhead, dramatically reducing the volume of checkpoints.
- Multi-Layer Staging: Through advanced staging and streaming techniques, DECK enables checkpoint operations to proceed concurrently with ongoing training with little interference, effectively minimizing training interruptions associated with checkpoint procedures.

Proceedings of the VLDB Endowment, Vol. 18, No. 12 ISSN 2150-8097.
doi:10.14778/3750601.3750621

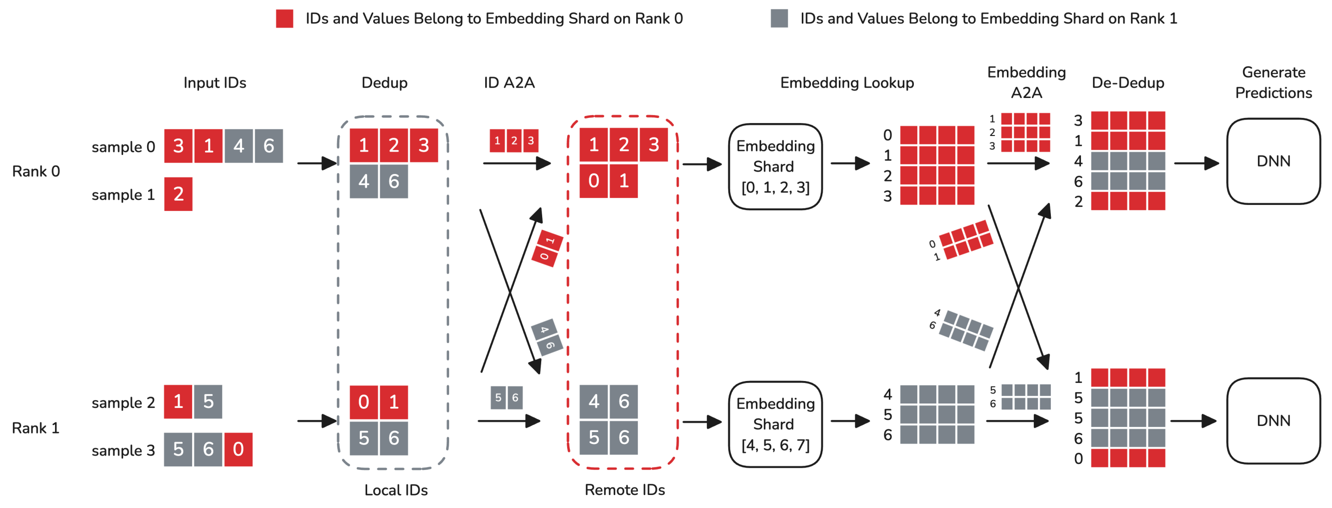

IDs and Values Belong to Embedding Shard on Rank 0
IDs and Values Belong to Embedding Shard on Rank 1
Input IDs
Dedup
ID A2A
Embedding Lookup
Embedding A2A
De-Dedup
Generate Predictions
Rank 0
Rank 1
sample 0
sample 1
sample 2
sample 3
Embedding Shard [0, 1, 2, 3]
Embedding Shard [4, 5, 6, 7]
DNN
Local IDs
Remote IDs

Figure 1: Simplified DLRM Forward Pass

- Optimal Hierarchical Delta Merging: DECK incorporates dedicated merging components to consolidate delta states into fewer and smaller files, balancing resource utilization and recovery efficiency through decoupled checkpointing and consolidation agents.

These innovations collectively enable DECK to achieve significantly higher checkpoint frequency while maintaining negligible impact on training throughput. The implementation of DECK consists of various optimizations throughout the stack, ranging from customized CUDA kernels to extensible adapters and components built on top of TorchRec [7]. Our experimental evaluation conducted on production infrastructure demonstrates that DECK permits a 12x increase in checkpoint frequency compared to prior arts, significantly reducing potential work loss during recovery. This performance improvement establishes DECK as the current industrial state-of-the-art solution for checkpoint management in large-scale recommendation systems.

The remainder of this paper is organized as follows: Section 2 introduces contexts for industrial recommendation systems. Section 3 details the architecture and technical components of the DECK framework. Section 4 reveals implementation decisions tailored for real-world use cases. Section 5 presents our experimental methodology and evaluation results. Section 6 reviews related work in model checkpointing and resilience strategies. Finally, Section 7 summarizes our contributions and concludes the paper.

2 BACKGROUND

This section provides contexts for industry-scale recommendation systems and key observations that motivate our design.

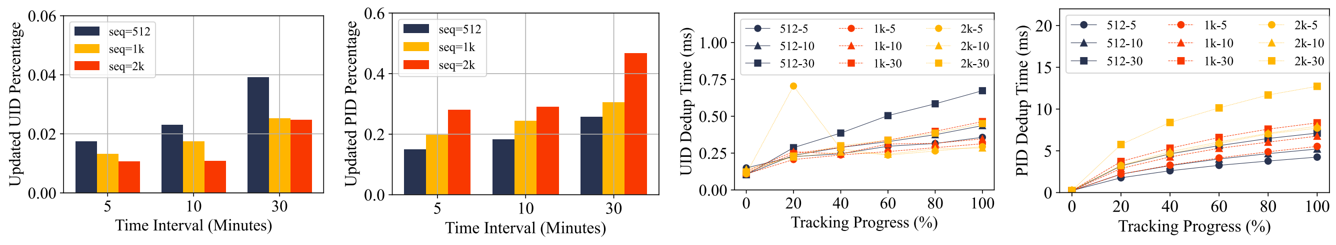
2.1 Industrial Scale Recommendation Systems

Recommendation systems are one of the largest software systems built to date [23], and a significant portion of datacenter capacity is dedicated to training and serving of recommendation models [6].

2.1.1 Deep Learning-based Recommendation Models. Modern recommendation models [23, 24] contain two major components: a collection of embedding tables and dense networks. Embedding tables are the core of recommenders, as they convert discrete entities (*e.g.*, users, movies, books) into continuous representations.

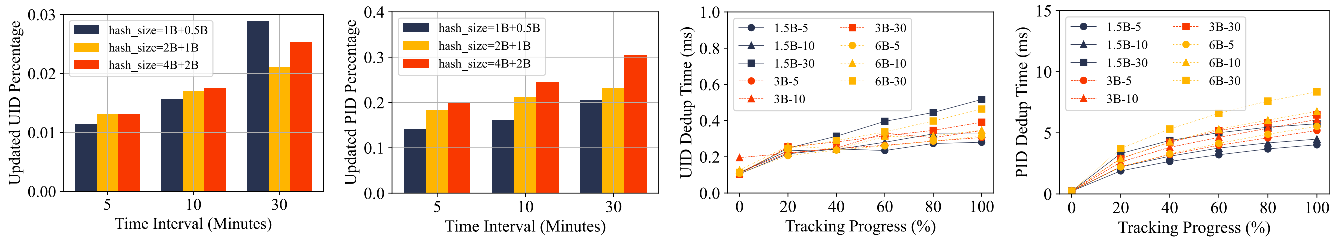
Industry-scale recommendation models are characterized by these huge embedding tables with trillions of parameters [3, 6, 10, 11, 13, 14, 27]. As a result, the state-of-the-art often uses a hybrid strategy to train these models in a distributed, synchronous manner: the embedding tables are sharded by a combination of rows and columns to different ranks [7, 12, 15], where inputs and embedding results are communicated across GPUs (see below); on the other hand, dense components are synchronized via distributed data parallel (DDP) [8] or fully sharded data parallel (FSDP) [28] due to their relatively small sizes. Therefore, most challenges in efficient training of modern recommendation models stem from efficient handling of embedding tables, which will be the focus of this paper.

2.1.2 Training Flow. Figure 1 illustrates a simplified flow of Deep Learning Recommendation Model (DLRM) forward pass running on a cluster of two ranks, where a rank is a logical concept that usually exclusively maps to one GPU device. In this example, the embedding table is sharded in a row-wise manner, where rank `0` owns embedding values for row `[0, 1, 2, 3]`, and rank `1` owns `[4, 5, 6, 7]`. Each rank independently loads its own input batch, where each batch contains multiple samples. Each sample consists of a variable number of float features, embedding values, embedding IDs, weights, etc. To simplify the presentation, Figure 1 only keeps the embedding IDs, as those IDs are the main components that interact with the embedding table. Each rank first removes duplicated IDs locally and then routes IDs to the corresponding rank to lookup embedding values using an `AllToAll` communication operator. This `AllToAll` essentially transposes IDs from a rank-major form (Local IDs) into shard-major form (Remote IDs). Upon receiving the IDs, each rank further de-duplicates global IDs from all ranks and retrieves corresponding values from its local embedding table shard. After that, another `AllToAll` routes the embedding values back to the original ranks requesting corresponding IDs, and those embedding will pass through DNN layers to generate final predictions.



(a) Updated User ID (b) Updated Post ID (c) Dedup User ID (d) Dedup Post ID

Figure 2: Unique Percentage and Dedup Cost vs User History Sequence Length



(a) Updated User ID (b) Updated Post ID (c) Dedup User ID (d) Dedup Post ID

Figure 3: Unique Percentage and Dedup Cost vs Embedding Table Size

2.2 Model Checkpointing

Large-scale distributed training inherently requires faster recovery, which demands frequent checkpointing. However, due to the sheer size of the models, frequent checkpoints of recommendation models incur prohibitively high cost. Fortunately, as we show empirically, full checkpointing at short intervals is unnecessary, due to a unique access pattern in the recommendation models.

2.2.1 Sparsity in Embedding Table Access Pattern. In contrast to dense networks, where all parameters are updated during each training iteration, embedding tables exhibit a sparse access pattern. Since embedding tables contain millions [5] to billions [3, 9, 11, 13, 23, 26, 27] of rows but each training sample accesses up to a few hundred on average, each iteration updates only a tiny fraction of the table. This sparsity is further amplified by the skewed distribution of item popularity. To demonstrate this, we conducted an empirical analysis using real-world data and production models. These models typically generate a full model checkpoint every 40 to 60 minutes. The models comprise multiple embedding tables, with the user table and item table being significantly larger than the rest combined. Figure 2 (a) and (b) illustrate the percentage of updated rows during training within various time intervals and for different maximum user history lengths, where the user history contains items the user has engaged recently. The results show that even with a user history length of 2048, the training process updates less than 5% of the entire user table after 30 minutes. In contrast, the item table sees more frequent updates, with approximately 50% of the rows being updated within 30 minutes at the same user history length. This disparity is driven by the inherent nature of recommendation systems, where users typically engage with many items over time. Nevertheless, the findings suggest that it is still possible to increase checkpoint frequency without incurring additional communication or storage overhead, even for the item table, by *only checkpointing the modified entries (delta).*

Figure 3 (a) and (b) illustrates the percentage change across various embedding table hash sizes, utilizing a fixed user history length of 1024. IDs that exceed the hash size are mapped back to the valid range through modulo operations. The percentage of updated post IDs demonstrates a consistent pattern, where larger hash sizes result in fewer collisions, consequently requiring more unique IDs observed. An intriguing observation emerges regarding the trend of unique user IDs, which exhibits a contrary directional pattern when varying sequence length versus varying hash size. This phenomenon can be attributed to the longer processing time required for each iteration when utilizing long user histories. Consequently, within a constant time interval, longer sequence lengths result in fewer completed iterations, which in turn leads to a reduction in the number of updated user IDs.

We note that this observation is not unique to our setting, as various prior works [6, 14] have adopted delta checkpointing to reduce checkpoint footprints that take advantage of the access pattern. However, existing approaches still incur significant overhead due to additional copies of states as basis for comparison, resulting in longer checkpoint loading and saving time, and quality implications from the need to tune hyperparameters that dictate what and when gets checkpointed. In this paper, we detail how DECK overcomes these challenges to minimize checkpoint overheads to deliver full fidelity checkpointing for industry-scale recommendation.

2.2.2 Storage Reliability. The checkpointing process involves writing trainer states and lineage metadata to persistent storage. In practice, checkpointing mechanisms and storage systems [19] are developed and maintained by separate teams, interfacing through distributed file system APIs with configurable replication levels and read/write checksums. Default API success rates typically range from 99.99% to 99.999%. For use cases requiring higher reliability, storage configurations can be adjusted to include additional replicas and checksums. Since storage reliability is orthogonal to the checkpointing mechanism, the remainder of this paper skips reliability-related considerations.

3 SYSTEM DESIGN

A robust checkpointing mechanism must function as a faithful representation of the training state, enabling seamless resumption from system failures. Furthermore, frequent checkpoint generation is typically desired to minimize progress loss. An optimal checkpointing solution should address three critical requirements:

- Keep training throughput intact, which calls for efficient parallel maintenance of checkpoint states alongside the primary training process.
- Minimize training interruption during checkpoint persistence operations, requiring prompt release of critical computation and storage resources.
- Recover rapidly from failures, *i.e.*, checkpoints can be loaded with high efficiency.

This section introduces DECK, an architecture leveraging *"Zero-Cost" Delta Tracking*, *Multi-Layer Staging*, and *Optimized Hierarchical Delta Merging* techniques to satisfy these essential requirements.

3.1 "Zero-Cost" Delta Tracking

A naive approach to implementing delta checkpoints involves maintaining a complete replica of the embedding tables from the previous checkpoint, computing the difference with the most recent version, and then extracting non-zero rows to construct the content for a delta checkpoint. However, given that even a single local shard of the embedding table can far exceed the capacity of an accelerator's High Bandwidth Memory (HBM), these tables typically reside on UVM and only move fractions of the full shard in an on-demand fashion. Consequently, computing the delta requires either utilizing CPU resources or processing in a chunked fashion on HBM, both of which incur significant memory overhead and execution latency. To mitigate these challenges, DECK employs an optimized strategy that amortizes computations and effectively hides them by overlapping with blocking communications during embedding lookup operations. Simultaneously, DECK maintains only the minimum possible states required to recover the delta content in a lossless manner, thereby minimizing storage requirements.

As illustrated in Figure 2 and Figure 3, each training iteration in DLRM modifies merely a tiny fraction of rows from the entire embedding table. Within a 5 minute window, only less than 15% of the rows are updated. Keeping a set of unique indices of touched rows would be sufficient to faithfully represent the delta content. This approach significantly reduces the memory requirements and computational overhead associated with tracking all rows.

To extract row indices, we first compare different locations to insert tracking logic. As discussed in Section 2.1.1, sharded embedding lookup operations effectively perform a distributed transpose across all ranks, shuffling input indices from rank-major to shard-major. This process relies on `AllToAll` communication to route lookup indices from the rank that loads the input batch (*i.e.*, local form in Figure 1) to the rank that owns the embedding shard (remote form). While tracking in the local form can be decoupled from sharded embedding table design and implementation, it may result in duplicated global indices being tracked across ranks, necessitating additional computations and communications to eliminate these duplicates in a distributed manner. In contrast, tracking in the remote form requires custom design for specific sharded embedding implementations, but the tracked IDs are readily usable for constructing delta checkpoint contents. To minimize computation and memory overhead, DECK employs tracking in the remote form. The engineering cost of implementing specialized tracking logic is typically manageable, as platforms like PyTorch provide a rich set of hooks for installing custom functions to extract internal states. Section 4.2 will further elaborate on how we restrict implementation-specific customization to adopters and generalize the remaining components in DECK.

The tracked indices represent the embedding rows that are accessed during the forward iteration, where gradients will be generated accordingly in the backward iteration. Although it is possible that gradients for some rows may be zero, DECK does not further filter out these rows for three reasons. Firstly, the use of `nonzero` operators is computationally expensive and requires device-to-host synchronizations, which can significantly impact performance. By avoiding these operators, DECK can achieve substantial computational efficiency gains. Secondly, if embedding tables are column-wise sharded, the decision to filter out zero-gradient rows could diverge across different ranks, necessitating additional communication to determine the global set of `nonzero` rows. Thirdly, even without explicit filtering, the number of touched rows typically falls within a reasonable threshold in most use cases, allowing the training job to affordably store all the indices in HBM. For instance, a 5TB embedding table with 256 columns and `FP16` precision requires only approximately 300MB of HBM space to store updated row indices for a 30-minute delta frequency. If storing indices in HBM becomes an issue, it is always possible to offload them to UVM. Therefore, DECK directly tracks all touched indices.

In addition to storage considerations, tracking deltas also incurs additional computational overhead to remove duplicated indices. To quantify this overhead, we utilize `CUDA` events to measure the execution time of `cat` and `unique` PyTorch operators on tracked indices. Figure 2 (c)(d) and Figure 3 (c)(d) illustrate the results, where removing duplicated user indices and post indices takes up to 0.7ms and 13ms, respectively. The cost increases with the total number of tracked indices throughout the iterations, as the delta from the latest iteration must be compared with a larger set of indices. Although the computational cost is non-trivial, these computations are off the critical path of the forward and backward passes. Consequently, we can reorder them to overlap with exposed communications, such as the `AllToAll` operation that shuffles the embedding lookup outputs. This communication is blocking, because subsequent computations depends on the communication results, which forces a bubble in the main `CUDA` compute stream. As long as the length of the exposed communications can overshadow de-duplication computations, tracking unique row indices is effectively free. Section 5.2 presents detailed measurements from real-world production training jobs.

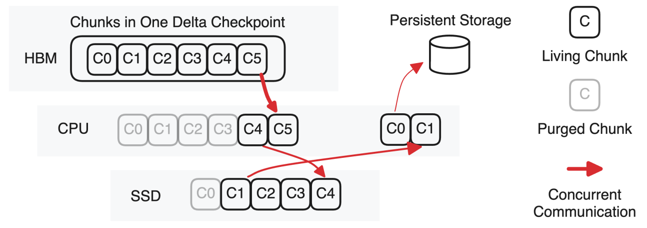


Figure 4: Multi-Layer Staging

With zero-cost tracking, whenever the system needs to generate a delta checkpoint, DECK just need to perform one embedding lookup on local shard to retrieve updated rows using the tracked indices, which only takes a few milliseconds.

3.2 Multi-Layer Staging

Although checkpointing delta states can substantially reduce communication and storage overhead, the sheer size of multi-terabyte models implies that even a 10% delta can result in checkpoints of hundreds of gigabytes, which may require several minutes to write to remote persistent storage. If the trainer frequently generates delta checkpoints, synchronously writing them to remote storage can significantly diminish system throughput. A straightforward alternative is to save checkpoints in an asynchronous manner. Given that HBM is a limited resource, practitioners typically maximize its use to enhance training efficiency. Consequently, retaining delta checkpoints in HBM is not feasible, which calls for separate staging storage solutions. Modern GPU servers are often equipped with ample CPU memory and SSD storage, which can facilitate the staging process. However, utilizing these storage options introduces additional challenges. CPU memory is typically already burdened by UVM embedding tables and model publishing processes, even before asynchronous checkpoints are enabled. Although SSDs offer plenty of space, writing large volumes of data into SSD can rapidly reduce their lifespan. Furthermore, techniques like GPUDirect are not universally available, meaning that SSD offloading still requires data to be copied to CPU memory first. To address these challenges, DECK employs multi-layer staging to optimize for the following four objectives:

- Minimize training pause duration by expeditiously offloading delta checkpoints out of HBM to create enough space for the next training iteration.
- Restrict CPU memory usage during offloading to avoid resource contention with other memory-intensive processes on the server.
- Optimize the effective SSD write volume to promote hardware longevity.
- Maximize network bandwidth utilization to stream checkpoints to remote persistent storage.

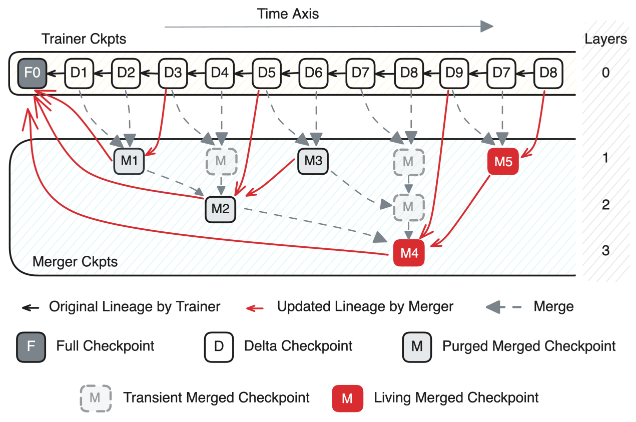
The solution involves segmenting a delta checkpoint into multiple chunks, which are then sequentially written to CPU memory, SSD, and remote persistent storage in a pipelined fashion. This approach allows for overlapping communications between the various storage media. To limit CPU memory usage, DECK prefetches only one chunk ahead into CPU memory relative to the SSD. Consequently, training can only resume once the delta checkpoints have been completely offloaded to the SSD, making the pause duration dependent on the SSD's write speed. Each SSD can support writing speeds of up to several gigabytes per second. In a 128-GPU training cluster, the aggregate bandwidth can easily reach 100GB/s, allowing delta checkpoints to be offloaded to the SSD in mere seconds, thereby ensuring a relatively short pause duration. Once a chunk is stored on the SSD, it is queued for streaming to remote persistent storage, which involves bringing the chunk back into CPU memory for network communication. Similarly, this process also prefetches at most one chunk in advance. A notable advantage of this approach is that the peak CPU memory consumption remains invariant to the model size. Instead, it is deterministically governed by the chunk size, which ensures that no more than four chunks are resident in CPU memory at any given time. This characteristic is exemplified in Figure 4, where concurrent communications across disparate storage media are represented by red arrows, with the thickness of each arrow denoting the corresponding channel bandwidth. By virtue of this design, DECK successfully decouples CPU memory usage from model size, thereby minimizing memory footprint while concurrently optimizing network bandwidth utilization.

The aforementioned multi-layer staging solution satisfies three out of the four objectives, leaving only the optimization of SSD write volume to be addressed. Two orthogonal approaches can be employed to achieve this goal: mitigating physical write amplification and reducing logical write content. Physical write amplification arises from the Program/Erase (P/E) cycle of SSDs, which is a fundamental operation for SSDs and determines their lifespan. When new data needs to be stored, the SSD controller writes the data to a new, blank location on the flash memory. The old data in the previous location is marked as invalid and will need to be erased before that location can be written to again. The P/E cycle operates at a granularity defined by a large page size. Consequently, if the delta checkpoint's chunk does not align with this page size, each P/E cycle will inevitably refresh underutilized pages, resulting in unnecessary physical write amplifications. To prevent underutilized pages during P/E cycles, DECK ensures that each chunk is sized identically to the P/E page, thereby minimizing write amplification and optimizing SSD endurance.

Logical volume reduction can be achieved by circumventing the SSD and transmitting chunks out of order, thereby enabling direct in-memory streaming to remote storage. In the example illustrated in Figure 4, chunk C0 can bypass the SSD and write directly to persistent storage. After C0 is finished, the next in-memory chunk can also skip the SSD as well and so on. All chunks offloaded to the SSD will be communicated to persistent storage after those that bypassed SSD. However, due to the significant speed gap between CPU memory and network, the SSD will only bypass a negligible percentage of chunks. Consequently, this SSD-bypassing technique is not adopted in production environments to reduce system complexity.

3.3 Optimal Hierarchical Delta Merging

Zero-cost delta tracking is an effective approach to minimizing lost progress during failures. However, the process of loading checkpoints can become significantly more expensive when the trainer needs to merge the latest full checkpoint with all subsequent delta



Time Axis
Trainer Ckpts
Layers
F0
D1
D2
D3
D4
D5
D6
D7
D8
D9
D7
D8
0
M1
M
M3
M
M5
1
M2
M
2
M4
3
Merger Ckpts
Original Lineage by Trainer
Updated Lineage by Merger
Merge
F Full Checkpoint
D Delta Checkpoint
M Purged Merged Checkpoint
M Transient Merged Checkpoint
M Living Merged Checkpoint

Figure 5: Hierarchical Merging

checkpoints. For instance, if a training job generates a full checkpoint every 4 hours and a delta checkpoint every 5 minutes, in the worst-case scenario, the trainer would need to merge 47 delta checkpoints into the full checkpoint upon resuming from a failure, resulting in substantial delays.

A natural solution to mitigate this issue is to perform delta checkpoint merging in parallel with the training process, thereby reducing both the number and the total size of pending delta checkpoints. This merging process can be executed on CPUs without occupying valuable GPU resources. For jobs with relatively light CPU workloads, trainers can spawn merging processes in-place on the same server. In cases where the trainer's CPU resources are already heavily utilized for tasks such as monitoring, input data loading, model publishing, and hosting large embedding tables in UVM, it is more suitable to allocate a small group of auxiliary CPU servers specifically for merging. The cost of these additional CPU servers should be negligible compared to the cost of GPU servers.

We propose the Hierarchical Merging Algorithm, where merging is conducted at a fixed but configurable stride s. After the merger process sees s consecutive delta checkpoints in layer i, it will merge them into a larger merged delta in layer $i+1$. Figure 5 illustrates an example where $s = 2$. In this example, the trainer process produces one new delta checkpoint at fixed time intervals. Solid lines highlight the dependency lineage of the delta checkpoints. One merged checkpoint is generated every $s = 2$ delta checkpoints. At step `4`, after merging `D3` and `D4`, the merger has accumulated $s = 2$ delta checkpoints at layer `1`, and hence it will further merge it with `M1` to create the final `M2` that represents the first 4 delta checkpoints. After that, `M1` can be deleted, and `M2`'s lineage dependency directly points to the original full checkpoint `F0`, as `M2` covers all the changes since then. A proper value of stride s can be derived from duplication ratio d and relative merging velocity v, where d indicates the ratio of duplicated rows from two equal-sized consecutive delta checkpoints and v the ratio of delta merging speed over delta producing speed. Suppose the trainer produces one delta checkpoint every 5 minutes and it takes the merger 1 minute to process it, then the relative merging velocity $v = 5$. Under this setup, merging s consecutive delta checkpoints results in a merged checkpoint with relative size $(2-d)^{\log s}$. The overall input volume of the merger process surpasses the the total volume of original delta checkpoints, because the merger also produced merged deltas. The total volume of merged checkpoints across all layers needs to stay below v times the delta producing speed. Hence, we have the following condition.

$$\sum_{i=1}^{\infty} \frac{(2-d)^{i \log s}}{s^i} < v \tag{1}$$

In Inequality 1, both d and v can be measured empirically and s can then be determined based on that.

Although the aforementioned design effectively reduces both the number and the volume of delta checkpoints, the size of pending deltas will continue to increase with the number of merging layers. In the worst-case scenario, the collective volume of merged checkpoints in each layer can approach the size of full checkpoints, ultimately leading to a substantial loading overhead. To mitigate this issue, the merger can be designed to create a full checkpoint every N steps, thereby truncating the lineage graph and preventing the indefinite growth of delta checkpoint lineages. The value of N can be dynamically determined based on the total volume of living delta checkpoints, allowing for adaptability in response to changing system conditions. Alternatively, for more predictable system behavior, N can be computed ahead of time using a predetermined equation:

$$\sum_{i=1}^{\log_s N} \left(\lfloor \frac{N}{s^i} \rfloor - \lfloor \frac{N}{s^{i+1}} \rfloor s\right)(2-d)^{i \log s} < F \tag{2}$$

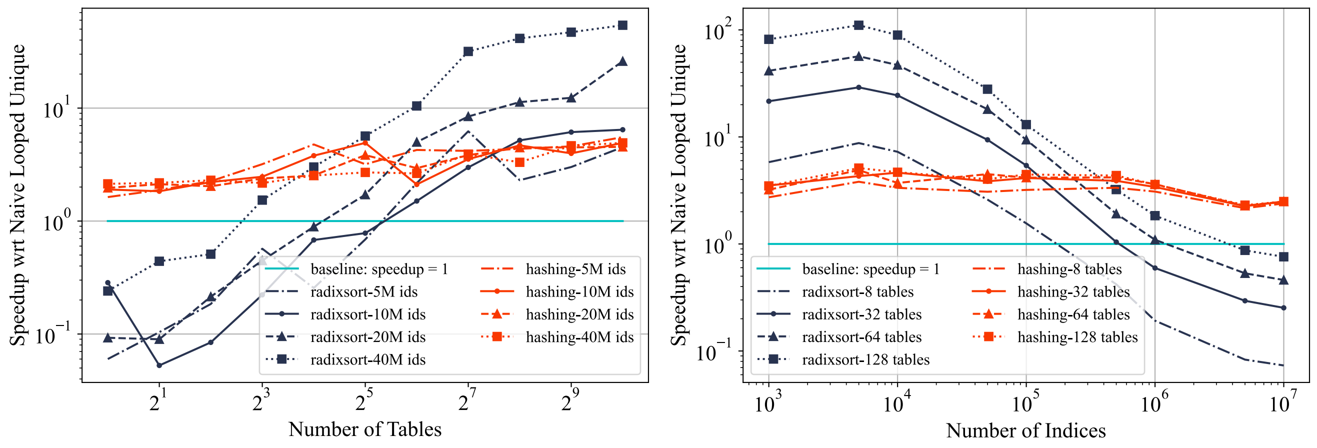
where the term in the first parentheses computes the number of living merged checkpoints in layer i and F denotes the upper bound of relative volume compared to an original delta checkpoint before the merger creates a full checkpoint.

4 IMPLEMENTATION

In this section, we present the technical implementation specifics of DECK, encompassing several key components. First, we elaborate on the CUDA kernel optimizations engineered to facilitate expeditious state tracking. Additionally, we discuss the extensible APIs designed to accommodate diverse sharded embedding table variants. Finally, we detail our checkpoint lineage management system, which provides robust protection against both system failures and data corruption incidents.

4.1 Optimized CUDA Kernels

A critical function in delta tracking involves de-duplicating tracked indices accumulated across training iterations for all embedding tables. A naive implementation would sequentially process each table with independent `unique` operations, resulting in substantial computational overhead, particularly for architectures with many embedding tables and large tracked indices set. To efficiently overlap ID de-duplication with `AllToAll` communications, maintaining consistent ID `unique` computation costs for diverse model architectures becomes essential. Therefore, we propose specialized accelerated CUDA kernels to supersede the naive approach. The optimal design of these unique acceleration mechanisms involves several technical considerations dependent on specific model requirements: (a) preservation of original indices ordering post-deduplication; (b)



(a) Vary number of tables under fixed total indices

(b) Vary number of indices per tables under fixed table cont

Figure 6: Speedup of `jagged_unique` Kernels with Respect to Naive Loop-Unique

necessity of inverse indices for original indices and state reconstruction; and (c) efficacy across diverse model scales and table quantities.

Implementation strategies include both sorting and hashing methodologies, each offering distinct advantages under varying data dimensions and application requirements. The sorting-based approach preserves sequence ordering after de-duplication and demonstrates superior performance for smaller indices collections, though it exhibits significant performance degradation with larger input sizes. To comprehensively address diverse data scales and application scenarios, we have developed two CUDA kernels based on sorting and hashing paradigms respectively. Both implementations perform `unique` operations on aggregated and jagged ID tensors in a single computational pass, eliminating the inefficiencies of sequential table-by-table de-duplication. The sorting-based kernel employs parallel and stable radix sort algorithms, while the hashing-based kernel utilizes hash partitioning techniques with optimized memory locality to mitigate performance deterioration when processing extensive indices collections.

- **Radix sorting based approach**: we first employ device-wide parallel radix sort to batch sort the jagged indices tensor, and then develop a custom jagged flagging kernel to identify the location of first occurrence of each index, followed by a parallel compaction operation to pick out all unique indices. Radix sort and compaction are implemented using high-performance device-wide primitives provided by the *CUB* library [17]. Given that the looped unique implementation also relies on radix sorting, this approach can ensure output order matches the naive approach.
- **Partitioned hashing based approach**: For smaller input size, we construct a hash set in global memory and use hashing to pick out the unique indices. For larger input size where data transfer would become a performance bottleneck, we hash partition the input indices into smaller partitions to increase locality, and then construct a hash set in either global memory or shared memory depending on the size of each partition. Note that hash collision would cause two unique indices colliding onto the same hash value, resulting in mis-deduplication. As such, concurrent hash map data structure provided by *cuCollections* library [18] that can resolve hash collision via linear probing is used to ensure output correctness. However, the hashing process disrupts the index sequence order, which could break ordering parity compared to the naive solution.

Figure 6 (a) illustrates the speedup of the two `jagged_unique` kernels with respect to naive looped `unique` across various table numbers given fixed total input size of 5M, 10M, 20M and 40M respectively. We can observe that the radix sorting-based kernel exhibits significant acceleration with large number of tables where the indices size per table is on the order of thousands. Conversely, it falls short at large input sizes per table, even showing worse performance than the baseline. Figure 6 (b) depicts the speedup in terms of indices size per table under different table number. From both figures, we can see that partitioned hashing based approach consistently delivers 2X-4X performance boost compared to the baseline under various conditions. In contrast, the speedup of the radix sort-based approach is highly sensitive to the input size per table. Therefore, we rely on partitioned hashing based kernel for subsequent experiments.

4.2 Extensible Tracker

As elaborated in Section 2.1.2, tracking in a remote form requires specializations depending on the implementations of distributed embedding tables. Consequently, it is imperative to make DECK extensible to accommodate a diverse range of use cases. Based on the design of DECK, it is evident that it requires two primary types of input:

- The indices of updated rows in the current iteration;
- Timing signals for executing computations to eliminate duplicate indices.

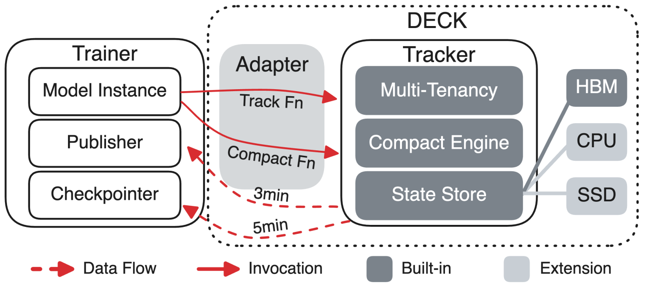


Figure 7: Custom Delta Tracker

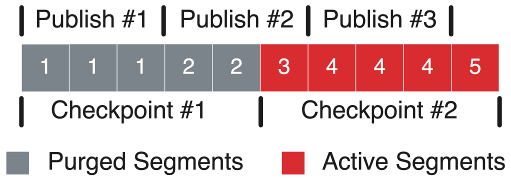
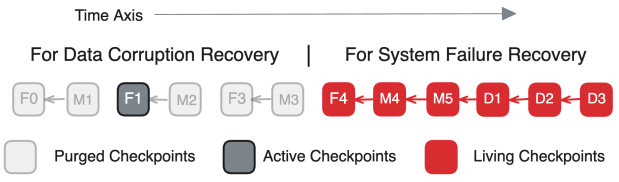


Figure 8: Dual Tenancy Example



Time Axis
For Data Corruption Recovery
For System Failure Recovery
F0 M1 F1 M2 F3 M3 F4 M4 M5 D1 D2 D3
Purged Checkpoints
Active Checkpoints
Living Checkpoints

Figure 9: Lineage Graph

The former ensures correctness, while the latter enhances performance. To facilitate this, we encapsulate these functionalities within an adapter API, wherein a new use case only needs to implement two registration methods that interface with the `track` and `compact` functions, respectively. The `track` function takes as arguments the Fully Qualified Name (FQN) of the embedding submodule, indices of updated rows, and any custom states. It stores these arguments in a memory-efficient format. The `compact` function informs DECK of the appropriate timing to execute computations (*i.e.*, `jagged_unique` in Section 4.1) to effectively remove duplicate indices and states. For instance, in the context of the sharded embedding table in `TorchRec`, the `track` method is configured as a pre-forward hook on the local `Embedding nn.Module`, while the `compact` method is triggered immediately following the invocation of `AllToAll` communication for lookup results.

Figure 7 illustrates the foundational components of the extensible tracker. The trainer operates on a package comprising multiple elements, among which the model instance, publisher process, and checkpoint processor are pertinent to DECK. The trainer shares the model instance with DECK, enabling DECK to apply a customized adapter that installs `track` and `compact` hooks as needed. This setup allows DECK to efficiently monitor updated row indices. In production online training systems, a concurrent publisher process typically streams model delta states to inference machines. Although publishing is beyond the scope of this paper, it is noteworthy that both the publisher and checkpointer utilize the same model instance and operate on the same device. DECK must ensure that its delta states are accessible to both, even if these two tenants retrieve delta states at varying intervals. In Figure 7, the trainer publishes every three minutes and generates checkpoints every five minutes. To prevent redundant state maintenance, DECK responds to requests from both tenants using the same storage, segmenting tracked states to ensure it can serve all tenants with the correct granularity. Figure 8 exemplifies this, showing that compaction is applied only within each segment. In the worst-case scenario, where each segment contains entirely duplicated states, DECK retains at most `N` copies of states, where `N` is the number of tenants. Fortunately, in all observed use cases, there are only up to two tenants, making the upperbound increase in delta state size manageable.

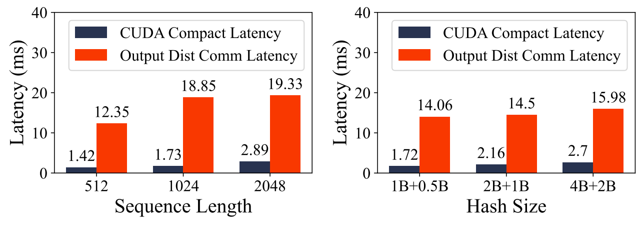
In our observations of existing use cases, DECK has been able to accommodate to the storage of tracked indices within HBM. However, as embedding tables continue to evolve, potentially featuring fewer columns and more rows or dynamic columns with collision-free rows, it is possible that storing all indices in HBM may no longer be feasible. To address this potential limitation, DECK has been designed with an extensible `StateStore` architecture, allowing for the incorporation of additional memory resources such as CPU memory and SSDs in future implementations. This modular design enables DECK to adapt to changing requirements and ensure efficient storage and management of tracked indices.

4.3 Lineage Graph

In Section 3.3, we briefly explored the maintenance of lineages for delta checkpoints, merged checkpoints, and full checkpoints to facilitate expedited failure recovery. In practical online training systems, failures are not solely attributed to system and software errors. A more complex challenge arises from data corruption, which can result from various factors, including silent errors in data logging systems, malicious attacks, or data outliers that violate inherent model assumptions. Unlike system or software errors, which typically lead to immediate crashes, data corruptions are often identified with considerable delay, usually through the observation of model quality regressions. When such an error is detected, it is imperative for the model to revert swiftly to a much earlier state, where data corruption is presumed absent. To accommodate such scenarios, maintaining lineage only up to the latest full checkpoint is inadequate. Instead, the system must also preserve past full checkpoints at relatively broad time intervals. Figure 9 illustrates this concept. The lineage graph comprises two segments: the red segments represent active checkpoints for rapid failure recovery, the dark grey segments denote active full checkpoints for addressing data corruption. The light grey segments indicate purged checkpoints from previous active lineage graphs. This approach enables the lineage system to support both types of recovery effectively.

5 EVALUATION

In this section, we conduct a comprehensive performance analysis of the DECK architecture. Initially, we decompose the system to evaluate each constituent component described in Section 3, analyzing their individual contributions to overall efficiency. Subsequently,



(a) Vary Sequence Length (b) Vary Embedding Table Size

Figure 10: Tracking Overhead

we execute end-to-end experiments utilizing real production infrastructure, models, and datasets to quantitatively assess DECK's performance characteristics under authentic operational conditions.

5.1 Experiment Setup

The experiments detailed in this section utilize up to 160 H100 GPUs, each with 94GB of memory, to host the training jobs for production HSTU [23] models. Each training machine is equipped with 8 GPUs, 2 TB of CPU memory, and 8 SSDs, each providing 4 TB of storage. Within each server, all GPUs are interconnected via NVSwitch, while different servers are linked using 8 200Gbps InfiniBand connections. Communication between HBM and the CPU is facilitated through PCIe 5.0.

The experiments in Section 5.2 are conducted using real production hardware, model implementations, data, and synthetic model configurations to accurately assess tracking latency across various setups. We deliberately avoid limiting ourselves to existing model configurations, as these configurations evolve rapidly due to changes in data volume and continuous improvements in system efficiency. In this way, we ensure that the experiments encompass a range of plausible configurations to verify that the tracker operates efficiently for both current and potential future workloads. In Sections 5.3 and 5.4, the focus is on evaluating the performance of individual components of DECK, namely checkpoint staging and delta merging. These experiments are conducted on real production hardware and model implementations, using synthetic model configurations and input data that accurately reflect real-world traffic patterns. Section 5.5 assesses DECK using entirely authentic production traffic to evaluate end-to-end overhead and performance improvements. Latencies associated with CUDA operations are measured using `CUDAEvents`.

5.2 Tracking Cost

DECK internally maintains a dynamic set of unique indices. To minimize the size of the tracked states, DECK employs an aggressive strategy to eliminate duplicate indices in each iteration through its `compact` function. The efficiency of this `compact` function is therefore critical. We measure the `compact` latency on CUDA GPU across various user engagement history sequence lengths and embedding table hash sizes configurations.

Figure 10 (a) presents the results of the final compact invocation at the end of a 30-minute interval, utilizing different sequence lengths. The implementation of customized CUDA kernels has resulted in significant speed enhancements for the `compact` functions. Compared to the baseline curves using `torch.cat` and `torch.unique` operations shown in Figure 2, the optimized kernels achieved approximately a 4.5x speed increase, reducing the compaction time from 13ms to 2.9ms. This latency is already below 2% of the per-iteration time for majority of the production models. To further mitigate this minor overhead, DECK overlaps the compaction process with the `AllToAll` communication of distributed embedding table outputs, whose latency is also depicted in Figure 10 (a). It is evident that the `compact` computation overhead can be entirely masked by the communication overhead. Although the compaction time increases with sequence length, it remains well below the communication time, which also increases with sequence length. The communication time does not increase proportionally with sequence length due to two factors:

- Variable sequence lengths introduce slight load imbalances across ranks, causing part of the communication time to be spent waiting for stragglers;
- Models typically perform sampling on input sequences before embedding table lookup to enhance efficiency.

Despite these factors, even if the communication overhead does not increase at all, DECK should still be capable of supporting sequences at least 8 times longer without incurring additional costs based on the measured data.

Figure 10 (b) evaluates the same set of metrics by varying the embedding table hash size. The results reaffirm that DECK can indeed achieve zero-cost tracking in terms of computations. An interesting observation is that although all three sets of experiments employed the same sequence length, the `AllToAll` communication time still increases steadily. This is likely because a larger hash size results in fewer index collisions, leading to a greater number of unique indices per iteration, which in turn results in more expensive communications.

5.3 Staging Overhead

In this section, we evaluate the performance of our checkpoint staging methodology. The primary objective of staging is to minimize training interruptions by expediting the checkpoint process. We distinguish between two critical temporal metrics: staging time and total uploading time. The former represents the interval until a checkpoint copy is secured in non-HBM storage, allowing training to resume, while the latter encompasses the complete transfer to remote persistent storage. Our experimental analysis examines CPU staging and SSD staging as separate implementations.

To isolate staging performance, we conducted experiments without delta checkpointing optimizations, using standard full embedding tables as input parameters. Figure 11 (a) presents performance measurements across varying embedding table hash sizes. The baseline (leftmost gray bar) represents conventional synchronous uploading of the entire embedding table to remote persistent storage. As anticipated, synchronous upload duration correlates linearly with embedding table size due to communication bandwidth constraints.

The implementation of staging technology substantially reduces training interruption time by a factor of 3-4, with variation depending on whether SSD or CPU memory serves as the destination. We observed that when embedding table shards exceed 20 billion

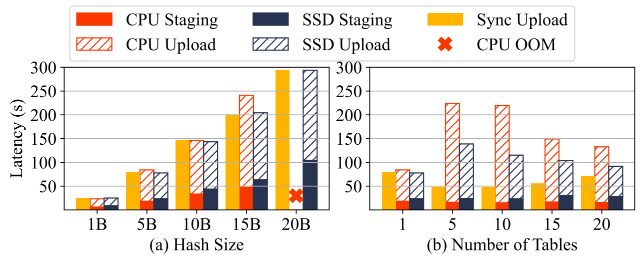


Figure 11: Staging Overhead

rows (approximately 60GB per GPU, *i.e.*, 480GB per host), the staging process triggers CPU Out-Of-Memory errors. Despite the total staging requirement per server being well under CPU memory capacity, available CPU memory remains insufficient due to concurrent resource demands from other system components such as publisher services and UVM embedding tables as elaborated in Section 3.2. Theoretically, CPU memory offloading should demonstrate significantly superior performance compared to SSD storage, given that the HBM-to-CPU communication channel offers bandwidth exceeding SSD transfer rates by more than an order of magnitude. However, our current CPU offloading implementation was not fully optimized due to its limited applicability caused on CPU memory hungry nature.

Figure 11 (b) illustrates the latency metrics associated with saving embedding tables totaling 5 billion hash size entries. The experimental results confirm similar latency reductions as observed in our previous assessment. However, an unexpected phenomenon emerges regarding total uploading duration: the measured time suffers from significant variability and follows a non-intuitive pattern, initially increasing and subsequently decreasing as the number of tables increases.

We hypothesize that this anomalous behavior stems from network resource contention with concurrent jobs running in the same physical cluster. The training infrastructure incorporates dual network architectures: a backend network utilizing Infiniband technology dedicated to inter-trainer communications, and a frontend network that facilitates connectivity between the training cluster and auxiliary systems including remote storage infrastructure, monitoring platforms, and checkpoint repositories. Therefore, concurrent communications within the same job or from other jobs in the same cluster could compete for bandwidth with checkpoint uploading, which could lead to large latency variances.

Since the pause duration is determined solely by staging latency rather than total uploading time, these upload duration fluctuations do not significantly impact training efficiency, provided they do not extend substantially enough to interfere with subsequent checkpoint upload operations.

5.4 Merging Cost

This section examines two distinct merging operations essential to our delta checkpointing architecture: (1) consolidate delta checkpoints into the most recent full checkpoint during training resumption and (2) combining delta checkpoints into merged checkpoints on CPU servers. The first operation quantifies the additional recovery latency introduced by our approach before training can resume, while the second determines CPU resource requirements to process delta checkpoints generated by GPU-based training clusters.

Figure 12 presents latency measurements for merging delta checkpoints into full checkpoints across various configurations. In sub-figure (a), we utilize a 10GB full checkpoint shard and 1GB delta checkpoint shards. The horizontal axis represents varying delta quantities, while the vertical axis decomposes the associated latency components. Our findings indicate that merging a single delta introduces minimal overhead, whereas merging five or more deltas begins to introduce non-trivial delays. However, considering that delta checkpointing reduces potential work loss from approximately 30 minutes to only 2-3 minutes, the additional 30-second merging overhead represents an acceptable performance trade-off.

In Figure 12 (b), we maintain a constant 10GB full checkpoint shard while varying the relative delta sizes. Results demonstrate that our current implementation processes larger but fewer deltas more efficiently than smaller but more more deltas. We attribute this to implementation limitations rather than fundamental algorithmic constraints and anticipate that optimized CUDA kernels could achieve comparable performance across both scenarios, as network communication remains the primary bottleneck. Figure 12 (c) experiments measuring the cost of consolidating a 10GB embedding table shard and 1GB delta shards across different numbers of tables, which reveals that table count does not significantly impact merging performance.

Figure 13 illustrates delta merging overhead on CPU servers using identical configurations to Figure 12. In most scenarios, CPU-based merging requires only marginally more time than GPU-based merging, remaining well below delta generation rates. For the 25% delta case shown in the figure, each CPU server can effectively process checkpoints from approximately 15 GPUs when generating deltas at 5-minute intervals. Notably, loading operations consistently execute faster on CPU compared to CUDA environments as the latter requires an additional host-to-device memory transfer to make checkpoint values available to CUDA operations.

5.5 Effective Training Time

This section presents an assessment of DECK's end-to-end efficiency in real production deployments. The primary objective of DECK is to increase checkpoint frequency while simultaneously minimizing adverse effects on training throughput. To quantitatively evaluate DECK's efficacy in achieving this objective, we conduct a comparative analysis between DECK's staging latency and TorchRec [7] sharded full model checkpointing methodologies. To isolate the effects of delta tracking and multi-layer staging (MLS), we evaluated two configurations: *DECK+TorchRec*, which enables only delta tracking on top of TorchRec, and *DECK+MLS*, which includes all DECK features.

Figure 14 illustrates the performance results obtained from experiments conducted using a 5TB production model distributed across 160 94GB H100 GPUs, with each GPU managing a 32GB UVM embedding table shard. The model employs HSTU with Stochastic Length [23], which downsamples UIH items based on UIH length before embedding table lookup operations. The hyperparameter α

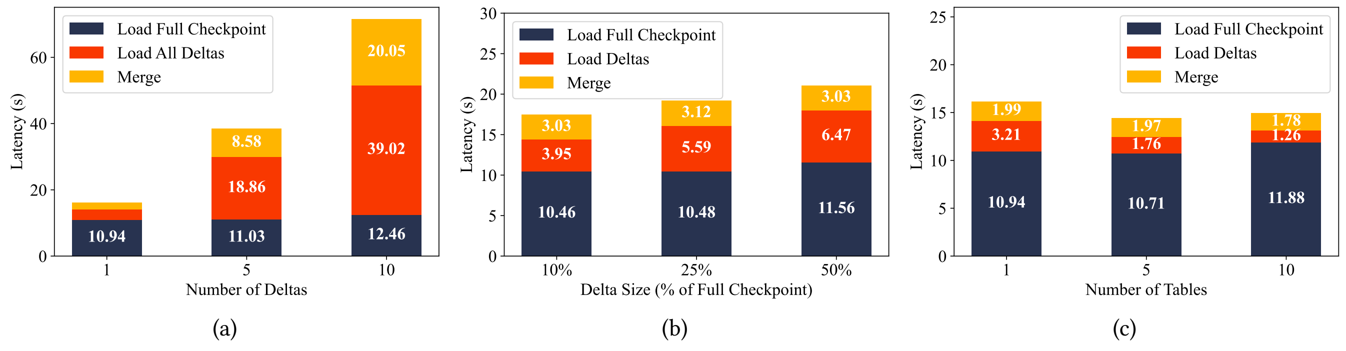


Figure 12: Delta Merging Overhead

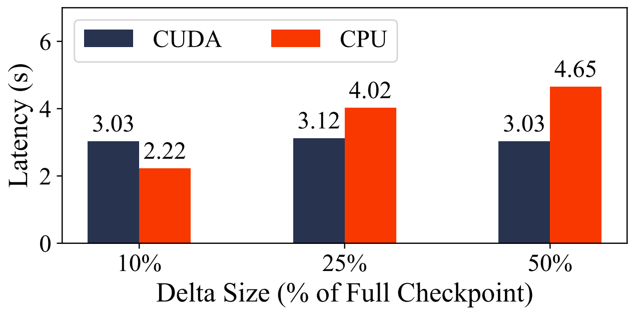
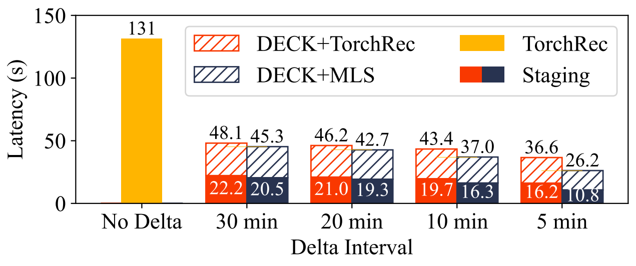
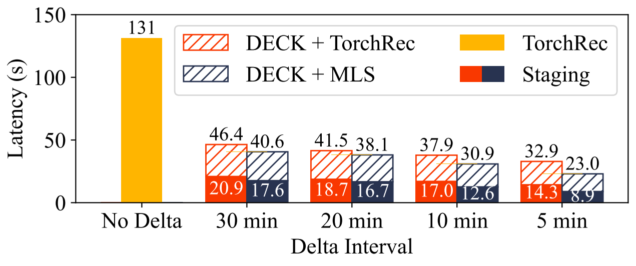


Figure 13: Merge on CUDA vs Merge on CPU



(a) Stochastic Length $\alpha = 1.7$



(b) Stochastic Length $\alpha = 1.3$

Figure 14: Overall DECK Overhead

controls the downsampling probability, where smaller values result in more aggressive downsampling and hence fewer updated rows every iteration. For the majority of use cases we have seen so far, the value of α falls in range `[1.3, 1.7]`. The leftmost data point indicates that synchronous full checkpointing requires 131.4 seconds to complete. Given that these full model checkpoints are generated at 50-minute intervals, the checkpointing process consumes approximately 4.4% of overall training throughput. Our analysis reveals that individual delta checkpointing overhead follows a positive correlation with generation interval duration, as longer intervals result in more unique row modifications. However, the impact on training throughput demonstrates an inverse relationship, as fewer checkpoint operations are performed within a specified time period. When implementing delta checkpoints at 20-minute intervals (representing a 2.5x increase in checkpoint frequency), the throughput reduction decreases to only 1.6%, calculated as 19.3 seconds divided by 1,200 seconds. Most notably, even with the highest frequency configuration of 5-minute delta checkpoint intervals depicted in the figure, the impact on training throughput remains at only 3.6% and 3.0% respectively with *DECK+MLS*, which outperforms TorchRec's sharded full checkpointing methodologies but delivers a 10x reduction in potential progress loss during recovery scenarios.

Figure 15 shows the Effective Training Time (ETT) across different checkpoint intervals for $\alpha = 1.7$ and $\alpha = 1.3$. ETT measures the percentage of time spent on processing training data, excluding checkpoint-related pauses. Checkpoint intervals are typically chosen offline based on trainer overhead, network bandwidth budget, and storage capacity constraints, usually ranging from 40 to 60 minutes. The grey dashed line at 96.3% represents the highest production ETT we have seen using sharded full checkpoints every 60 minutes without DECK. With DECK enabled, the ETT exceeds 96.4% while reducing progress loss by 12x. While it's possible to eliminate the remaining 3.6% staging overhead, *e.g.*, by making staging asynchronous and tracking delta during staging as auxiliary deltas, DECK already surpasses the highest existing ETT with sharded full checkpoints. Given that 3.6% is minor and can be outweighed by typical system noise, further optimizing staging may not justify the added engineering complexity.

In practice, while workloads fluctuate, especially during online training, DECK's overhead remains relatively stable for two key reasons: UIH length variations are mitigated by techniques like Stochastic Length [23] where sequences with longer history has a higher downsample probability, and fluctuations in request volume are absorbed by elastic training (*i.e.*, spatially) or traffic backlog

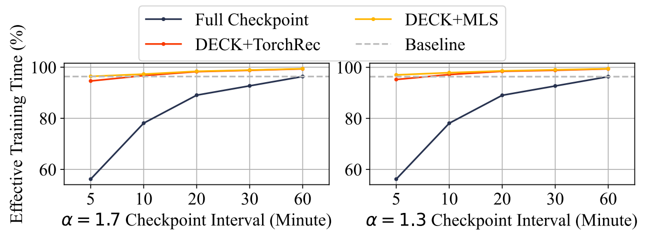


Figure 15: Effective Training Time

systems (*i.e.*, temporally). As a result, per-GPU checkpoint overhead stays relatively consistent over time. More intuitively, training systems usually aim to maximize per-GPU throughput, which depends primarily on hardware and software efficiency. DECK's staging and merging overhead scales with this per-GPU throughput and is largely independent of workload dynamics, because the delta size of a local embedding table shard is determined by how many tokens the corresponding GPU processes per time unit.

6 RELATED WORK

Deep Learning-based Recommendation Models Recent research on dense recommendation model scaling and scaling laws [23, 24] can shift the relative sparse to dense network ratio, however, the sparse components continue to dominate in terms of weight sizes [9, 11, 13]. Further, from a modeling perspective, scaling up both dense and sparse networks will yield the best model quality, future proofing DECK. Additionally, many of DECK's components, including staging and merging, are highly applicable to efficient checkpointing of other parts of the model.

Hierarchical Checkpoints Memory hierarchy is used in industrial recommenders for checkpointing. While not directly applicable to the recommendation setting where no redundant CPU memory is available for checkpointing, advancements have been made in in-memory checkpoints for LLMs for fast failure recovery. In particular, Gemini [21] proposed an optimal placement strategy for maximizing probability of failure recovery and a fine-grained, pipelined approach to transfer checkpoints and reduce interference between checkpointing and training. In AIBox [27] and [26], Baidu adopts a multi-level, HBM-CPU-SSD hierarchy for embedding storage, with novel hashing methods, compression algorithms, managed caches and multi-stage pipelines to minimize latency from lower-tier storage. OpenEmbedding [26] introduces persistent memory to the hierarchy, and uses a co-designed cache replacement strategy with the checkpoint process to minimize checkpointing overheads. Similarly, DECK uses a chunked and asynchronous method to hide checkpoint transfer overheads in SSDs.

Delta Checkpoints DECK improves upon prior arts from multiple aspects, including drastically reduced tracking and storage overheads as well as transparency in deployments.

Compared to Check-N-Run [6], DECK does not require an empirical model to determine whether a differential or a baseline checkpoint should be taken. Compared to QuickUpdate [14], DECK has no quality implications and more ubiquity as it does not use specific optimizer state to approximate model delta and can be applied to all optimizers. Unlike both Check-N-Run and QuickUpdate, DECK incurs virtually no tracking overheads and minimum storage overheads as it only need to remember the changed row IDs, not a snapshot of model state as the basis of calculating delta.

Checkpoint Pruning DECK can work with orthogonal selective checkpoint techniques to further reduce checkpoint storage overheads while maintaining low tracking performance: for example, to support optimizer-state guided selective checkpointing, the delta tracker can filter IDs out based on the current momentum of the optimizer states of the embedding tables.

Checkpoint Compression and Quantization Checkpoint footprints can be further reduced using both standard compression algorithm [4] and quantization approaches on model weights [1], especially when used in synergy with quantized communication [22], as they can simultaneously reduce both network bandwidth and remote storage requirements for checkpoints.

Alternatives to Delta Tracking In specific cases where the `AllToAll` paradigm is not used for ID distribution, efficient data tracking is still possible by tapping into the data service. For example, the (distributed) data loader has a global view that can work in tandem with embedding table sharding metadata that, upon request, instructs the GPU workers which rows of its embedding tables need to be checkpointed, and this notification can still be hidden as part of the training process.

Fault Tolerance and Delta Checkpoints One downside of maintaining multiple delta checkpoints is the restoration of a full checkpoint relies on the availability of all delta checkpoints plus the previous full checkpoint, resulting in fault-tolerance concerns. While this can be mitigated by using a smaller stride to encourage frequent consolidation, we acknowledge that it ultimately requires a trade-off between availability and storage overheads, and we believe the benefits significantly outweighs the associated risks. DECK's current recovery process in face of missing delta checkpoints involve immediately requesting a full checkpoint from the GPU workers, thereby allowing all previous checkpoints to be safely purged.

7 CONCLUSION

This paper presents the design, implementation, and evaluation of DECK, a high-performance model checkpointing system specifically tailored for industrial recommendation systems. The system's efficiency is achieved through the synergistic combination of zero-cost tracking, multi-layer staging, and optimal hierarchical merging techniques. To further optimize performance, key system components are meticulously optimized down to the CUDA kernel level, ensuring maximum efficiency. We conduct comprehensive experiments using real-world production clusters, models, and datasets to validate the effectiveness of DECK. Our results demonstrate that DECK achieves a 12-fold increase in checkpoint frequency while maintaining comparable training throughput costs, significantly outperforming conventional checkpointing solutions.

ACKNOWLEDGMENT

We acknowledge Andrew Briand, Chirayu Garg and Mohammed Karamathulla from Nvidia Corporation for helpful discussion of jagged unique kernels and for providing sample implementation of partitioned hashing kernel.

REFERENCES

[1] [n.d.]. GGUF — huggingface.co. https://huggingface.co/docs/hub/en/gguf. [Accessed 17-03-2025].

[2] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[3] Cheng Chen, Yilin Wang, Jun Yang, Yiming Liu, Mian Lu, Zhao Zheng, Bingsheng He, Weng-Fai Wong, Liang You, Penghao Sun, et al. 2023. Openembedding: A distributed parameter server for deep learning recommendation models using persistent memory. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 2976–2987.

[4] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. 2014. Project adam: Building an efficient and scalable deep learning training system. In *11th USENIX symposium on operating systems design and implementation (OSDI 14)*. 571–582.

[5] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM conference on recommender systems*. 191–198.

[6] Assaf Eisenman, Kiran Kumar Matam, Steven Ingram, Dheevatsa Mudigere, Raghuraman Krishnamoorthi, Krishnakumar Nair, Misha Smelyanskiy, and Murali Annavaram. 2022. {Check-N-Run}: A checkpointing system for training deep learning recommendation models. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 929–943.

[7] Dmytro Ivchenko, Dennis Van Der Staay, Colin Taylor, Xing Liu, Will Feng, Rahul Kindi, Anirudh Sudarshan, and Shahin Sefati. 2022. TorchRec: a PyTorch Domain Library for Recommendation Systems. In *Proceedings of the 16th ACM Conference on Recommender Systems* (Seattle, WA, USA) *(RecSys '22)*. Association for Computing Machinery, New York, NY, USA, 482–483. https://doi.org/10.1145/3523227.3547387

[8] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. 2020. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. arXiv:2006.15704 [cs.DC] https://arxiv.org/abs/2006.15704

[9] Xiangru Lian, Binhang Yuan, Xuefeng Zhu, Yulong Wang, Yongjun He, Honghuan Wu, Lei Sun, Haodong Lyu, Chengjun Liu, Xing Dong, et al. 2022. Persia: An open, hybrid system scaling deep learning-based recommenders up to 100 trillion parameters. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 3288–3298.

[10] Xiangru Lian, Binhang Yuan, Xuefeng Zhu, Yulong Wang, Yongjun He, Honghuan Wu, Lei Sun, Haodong Lyu, Chengjun Liu, Xing Dong, Yiqiao Liao, Mingnan Luo, Congfei Zhang, Jingru Xie, Haonan Li, Lei Chen, Renjie Huang, Jianying Lin, Chengchun Shu, Xuezhong Qiu, Zhishan Liu, Dongying Kong, Lei Yuan, Hai Yu, Sen Yang, Ce Zhang, and Ji Liu. 2021. Persia: An Open, Hybrid System Scaling Deep Learning-based Recommenders up to 100 Trillion Parameters. arXiv:2111.05897 [cs.LG] https://arxiv.org/abs/2111.05897

[11] Zhuoran Liu, Leqi Zou, Xuan Zou, Caihua Wang, Biao Zhang, Da Tang, Bolin Zhu, Yijie Zhu, Peng Wu, Ke Wang, et al. 2022. Monolith: real time recommendation system with collisionless embedding table. *arXiv preprint arXiv:2209.07663* (2022).

[12] Liang Luo, Buyun Zhang, Michael Tsang, Yinbin Ma, Ching-Hsiang Chu, Yuxin Chen, Shen Li, Yuchen Hao, Yanli Zhao, Guna Lakshminarayanan, Ellie Dingqiao Wen, Jongsoo Park, Dheevatsa Mudigere, and Maxim Naumov. 2024. Disaggregated Multi-Tower: Topology-aware Modeling Technique for Efficient Large-Scale Recommendation. arXiv:2403.00877 [cs.LG] https://arxiv.org/abs/2403.00877

[13] Xiao Lv, Jiangxia Cao, Shijie Guan, Xiaoyou Zhou, Zhiguang Qi, Yaqiang Zang, Ming Li, Ben Wang, Kun Gai, and Guorui Zhou. 2024. MARM: Unlocking the Future of Recommendation Systems through Memory Augmentation and Scalable Complexity. *arXiv preprint arXiv:2411.09425* (2024).

[14] Kiran Kumar Matam, Hani Ramezani, Fan Wang, Zeliang Chen, Yue Dong, Maomao Ding, Zhiwei Zhao, Zhengyu Zhang, Ellie Wen, and Assaf Eisenman. 2024. QuickUpdate: a Real-Time Personalization System for Large-Scale Recommendation Models. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 731–744. https://www.usenix.org/conference/nsdi24/presentation/matam

[15] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Zhihao Jia, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, Liang Luo, Jie (Amy) Yang, Leon Gao, Dmytro Ivchenko, Aarti Basant, Yuxi Hu, Jiyan Yang, Ehsan K. Ardestani, Xiaodong Wang, Rakesh Komuravelli, Ching-Hsiang Chu, Serhat Yilmaz, Huayu Li, Jiyuan Qian, Zhuobo Feng, Yinbin Ma, Junjie Yang, Ellie Wen, Hong Li, Lin Yang, Chonglin Sun, Whitney Zhao, Dimitry Melts, Krishna Dhulipala, KR Kishore, Tyler Graf, Assaf Eisenman, Kiran Kumar Matam, Adi Gangidi, Guoqiang Jerry Chen, Manoj Krishnan, Avinash Nayak, Krishnakumar Nair, Bharath Muthiah, Mahmoud khorashadi, Pallab Bhattacharya, Petr Lapukhov, Maxim Naumov, Ajit Mathews, Lin Qiao, Mikhail Smelyanskiy, Bill Jia, and Vijay Rao. 2022. Software-hardware co-design for fast and scalable training of deep learning recommendation models. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 993–1011. https://doi.org/10.1145/3470496.3533727

[16] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. 2019. Deep Learning Recommendation Model for Personalization and Recommendation Systems. https://doi.org/10.48550/ARXIV.1906.00091

[17] Nvidia. 2025. Cooperative primitives for CUDA C++. https://docs.nvidia.com/cuda/cub/index.html.

[18] Nvidia. 2025. cuCollections: an open-source, header-only library of GPU-accelerated, concurrent data structures. https://github.com/NVIDIA/cuCollections.

[19] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, et al. 2021. Facebook's tectonic filesystem: Efficiency from exascale. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 217–231.

[20] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).

[21] Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, TS Eugene Ng, and Yida Wang. 2023. Gemini: Fast failure recovery in distributed training with in-memory checkpoints. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 364–381.

[22] Jie Amy Yang, Jongsoo Park, Srinivas Sridharan, and Ping Tak Peter Tang. 2020. Training deep learning recommendation model with quantized collective communications. In *Conference on Knowledge Discovery and Data Mining (KDD)*. 95.

[23] Jiaqi Zhai, Lucy Liao, Xing Liu, Yueming Wang, Rui Li, Xuan Cao, Leon Gao, Zhaojie Gong, Fangda Gu, Michael He, Yinghai Lu, and Yu Shi. 2024. Actions Speak Louder than Words: Trillion-Parameter Sequential Transducers for Generative Recommendations. arXiv:2402.17152 [cs.LG] https://arxiv.org/abs/2402.17152

[24] Buyun Zhang, Liang Luo, Yuxin Chen, Jade Nie, Xi Liu, Daifeng Guo, Yanli Zhao, Shen Li, Yuchen Hao, Yantao Yao, et al. 2024. Wukong: Towards a scaling law for large-scale recommendation. *arXiv preprint arXiv:2403.02545* (2024).

[25] Buyun Zhang, Liang Luo, Xi Liu, Jay Li, Zeliang Chen, Weilin Zhang, Xiaohan Wei, Yuchen Hao, Michael Tsang, Wenjun Wang, Yang Liu, Huayu Li, Yasmine Badr, Jongsoo Park, Jiyan Yang, Dheevatsa Mudigere, and Ellie Wen. 2022. DHEN: A Deep and Hierarchical Ensemble Network for Large-Scale Click-Through Rate Prediction. https://doi.org/10.48550/ARXIV.2203.11014

[26] Weijie Zhao, Deping Xie, Ronglai Jia, Yulei Qian, Ruiquan Ding, Mingming Sun, and Ping Li. 2020. Distributed hierarchical gpu parameter server for massive scale deep learning ads systems. *Proceedings of Machine Learning and Systems* 2 (2020), 412–428.

[27] Weijie Zhao, Jingyuan Zhang, Deping Xie, Yulei Qian, Ronglai Jia, and Ping Li. 2019. AIBox: CTR prediction model training on a single node. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*. 319–328.

[28] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. 2023. PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel. arXiv:2304.11277 [cs.DC] https://arxiv.org/abs/2304.11277