AdaptCP: Hotness-Aware Adaptive Checkpointing for Industry-Scale Recommendation Model Training

Junhong Liu
Beihang University
Beijing, China
junhongliu@buaa.edu.cn

Yitang Yang
Beihang University
Beijing, China
yi1tang.yang@gmail.com

Xiaoyang Sun
Kuaishou Inc.
Beijing, China
sunxiaoyang@kuaishou.com

Jiapeng Chen
Kuaishou Inc.
Beijing, China
chenjiapeng@kuaishou.com

Fangzheng Li
Beihang University
Beijing, China
lifangzheng@buaa.edu.cn

Zheng Zheng
Beihang University
Beijing, China
zhengz@buaa.edu.cn

Menghao Zhang
Beihang University
Beijing, China
zhangmenghao0503@gmail.com

Tianyu Wo
Beihang University
Beijing, China
woty@buaa.edu.cn

Chunming Hu
Beihang University
Beijing, China
hucm@buaa.edu.cn

Chengru Song
Kuaishou Inc.
Beijing, China
songchengru@kuaishou.com

Jin Ouyang
Unaffiliated
Beijing, China
oyjmical@gmail.com

Renyu Yang*
Beihang University
Beijing, China
renyuyang@buaa.edu.cn

Abstract

Deep Learning Recommendation Models (DLRMs) are pivotal to modern digital services, yet their terabyte-scale embedding tables pose an unprecedented obstacle to fault-tolerant distributed training. Existing checkpoint systems suffer from an intractable dilemma between training throughput and recovery speed, hampered by blocking I/O, excessive memory pressure on parameter servers, and an inherent mismatch with the power-law distribution of DLRM access patterns. In this paper, we introduce AdaptCP, a high-performance checkpoint system designed for training industry-scale DLRMs. AdaptCP leverages dedicated checkpointing strategies for different types of embeddings based on the awareness of embedding access frequency. A cost model is elaborated to navigate the partitioning of cold and hot embeddings, and pinpoint the proper threshold dynamically when the model training is on the fly. Additionally, a stall-free Save-on-Update (SoU) pipeline is proposed to decouple checkpointing from the training loop through priority-based asynchronous saving, thereby eliminating the training suspension. A high-performance parallel I/O engine is used for tackling storage bottlenecks with SSD buffering and multi-threaded writes. Our extensive evaluation shows that AdaptCP improves training throughput by 13.87× and reduces recovery latency by 13.62×, when compared with the state-of-the-art checkpointing systems. AdaptCP has been deployed into production to successfully support high-throughput and dependable recommendation model training at scale.

CCS Concepts: • **Computer systems organization → Reliability**; • **Computing methodologies → Distributed computing methodologies**.

Keywords: Checkpointing, Fault Tolerance, Distributed Training, Deep Learning Recommendation Models, Embedding Tables

ACM Reference Format:
Junhong Liu, Yitang Yang, Xiaoyang Sun, Jiapeng Chen, Fangzheng Li, Zheng Zheng, Menghao Zhang, Tianyu Wo, Chunming Hu, Chengru Song, Jin Ouyang, and Renyu Yang. 2027. AdaptCP: Hotness-Aware Adaptive Checkpointing for Industry-Scale Recommendation Model Training. In *22nd European Conference on Computer Systems (EuroSys '27), April 19-23, 2027, Rabat, Morocco.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3842654.3848523

1 Introduction

Deep Learning Recommendation Models (DLRMs) [24, 33] are cornerstones of modern digital services, powering applications from e-commerce product recommendations to personalized social media feeds by effectively processing

*Renyu Yang is the corresponding author.

EuroSys '27, Rabat, Morocco

ACM ISBN 979-8-4007-2971-3/2027/04
https://doi.org/10.1145/3842654.3848523

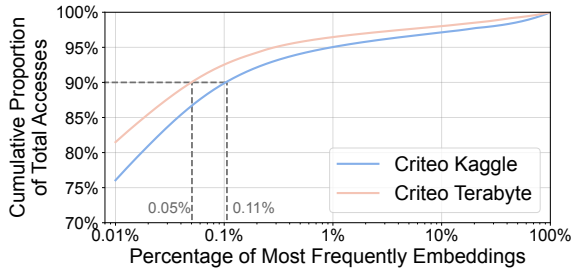


Figure 1. Cumulative distribution function (CDF) of embedding accesses across datasets.

sparse categorical features like user IDs and item IDs into dense and low-dimensional embedding representations. The parameter size of embedding tables can boost to tens of terabytes in production – a single recommendation model may have tens of billions of embedding keys and require large-scale distributed CPU parameter servers and GPU clusters for both storage and training [1].

As distributed training is vulnerable to node failures, network issues, and software faults, checkpointing – a technique to periodically save model parameters and optimizer states – is essential for fault tolerance. However, for large-scale DLRMs, the naive approach of full-state checkpointing necessitates serializing and saving the entire embedding table. This process imposes immense storage I/O pressure and, more critically, requires halting training to ensure consistency, resulting in prolonged stalls that severely impact training efficiency [10].

Prior work [10, 31] reduces training stalls by decoupling checkpointing from the main training process through snapshotting parameters to CPU memory. For example, Check-N-Run [10] keeps embedding parameters entirely in GPU memory and, upon checkpointing, mirrors them from GPU memory to CPU memory, which allows checkpoint persistence to proceed asynchronously in the background on the CPU side. However, this design is infeasible for large-scale recommendation models that maintain tens of TBs of embedding states, which far exceed the capacity of GPU memory.

To reduce I/O overhead, modern training systems use differential checkpointing (storing changes from a baseline) or incremental checkpointing (storing changes since the last save) [10, 19]. However, both of them have key drawbacks: differential checkpoints grow over time, slowing-down the save operations, while incremental checkpointing creates extended dependency chains, which substantially increase system recovery latency.

We present AdaptCP, a high-performance checkpointing system for large-scale recommendation models. The key insight is to customize different checkpointing strategies for embeddings based on their hotness, adaptively classifying them as hot or cold using a dynamic threshold that tracks runtime changes. To do so, AdaptCP leverages the power-law distribution of access frequency and strong locality in DLRM embeddings. As shown in Fig. 1, on the Criteo Terabyte dataset [14], 90% of accesses target only 0.05% of *hot* embeddings, while most *cold* embeddings are rarely accessed during training. Our study shows that the most cost-effective strategy strongly depends on embedding hotness: differential checkpointing minimizes recovery time for hot embeddings, whereas incremental checkpointing reduces I/O costs for cold embeddings. We further prove that combining differential and incremental checkpointing yields a minimal save/recovery cost, which guides our design of specialized checkpointing strategies.

Additionally, unlike typical deep learning models that update nearly all parameters, DLRMs exhibit a sparse update pattern, where each training batch modifies only a tiny fraction of embedding parameters. Sparsity presents a unique opportunity to minimize downtime – instead of a global training pause, data consistency can be ensured by checkpointing only the specific parameters targeted for update. AdaptCP therefore introduces a Save-on-Update pipeline that triggers checkpointing asynchronously at the embedding prefetching stage, decoupling the checkpointing from the training stage. AdaptCP is also enhanced by a parallelized write engine that primarily uses an SSD buffer to bypass I/O bottlenecks, effectively overcoming CPU and storage constraints to minimize persistence time. The same architecture also enables parallel checkpoint loading from remote storage, which substantially accelerates the training recovery. Experiments show that AdaptCP improves training throughput by 13.87× and reduces recovery latency by 13.62×, when compared with the state-of-the-art checkpointing systems. AdaptCP has been deployed into production to successfully support high-throughput and dependable recommendation model training at scale. This paper makes the following contributions.

- A novel hot-cold tiered hybrid checkpointing strategy that combines differential and incremental methods to balance storage efficiency and recovery speed (§ 3.2 and §3.3).
- A Save-on-Update, no-halt checkpointing mechanism that eliminates global pauses by pipelining training and saving processes (§ 3.4).
- A high-performance parallel persistence engine, integrated with an SSD buffering layer, to alleviate CPU memory and remote storage bottlenecks (§ 3.5).

2 Background and Motivation

2.1 DLRMs and Training Systems

Deep Learning Recommendation Models (DLRMs) have become central to modern internet services such as e-commerce, social media, and content distribution [5, 38]. These models analyze user-platform interaction data to learn user preferences and predict future behaviors for providing highly

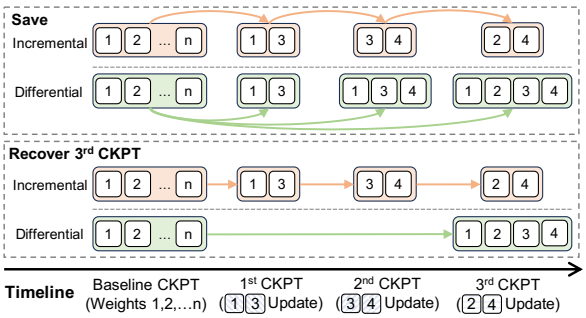


Figure 2. Incremental and differential checkpointing.

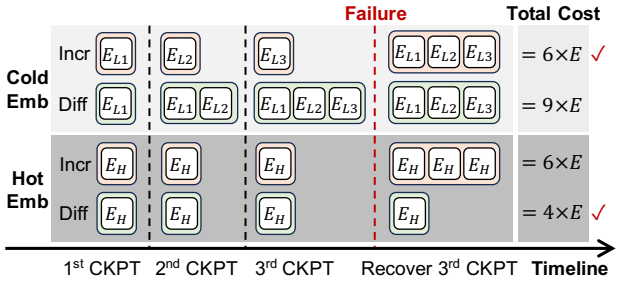


Figure 3. Impact of unbalanced embedding access on incremental and differential checkpoints.

personalized recommendations. Typical recommendation architectures [24] integrate both dense and sparse features, and the embedding layer plays a crucial role. Some models focused on capturing user behavior sequences [37, 38], while emerging generative recommendation architectures [9, 35] aim to directly predict the next item ID. However, the explosive growth of users, items and feature dimensions easily make the size of embedding tables reach tens of terabytes, putting huge pressure on the model storage and training. In industrial settings, distributed training (e.g., model-parallel paradigm) need to be performed for training such massive DLRMs, given that embedding tables cannot fully fit into limited GPU memory. The entire embedding table is distributed across the CPU memory of multiple GPU servers, and managed by Parameter Server (PS) architecture [15]. The DLRM training workflow under this PS-based architecture is as follows:

- **Embedding Pull:** During the forward pass of training, the GPU requests the corresponding embedding vectors from the PS based on the indices required by the current batch.
- **GPU Computation:** The pulled embedding vectors, along with dense features, are fed into the GPU for complex neural network computations (e.g., forward and backward propagation of MLP layers).
- **Gradient Push:** The GPU transmits the computed embedding gradients back to the corresponding PS nodes via the network, where the embedding parameters are subsequently updated.

2.2 Limitations of Existing Checkpoint Strategies

To mitigate the prohibitive I/O cost of full checkpoints in large-scale DLRM training, recent systems exploit the sparsity of embedding updates through two representative optimizations: incremental checkpointing and differential checkpointing. Their typical workflow is shown in Fig. 2. Both approaches start from a full baseline checkpoint but differ in how they capture subsequent changes.

Checkpoint Saving. Incremental checkpointing saves only the subset of embedding parameters that have been updated since the previous checkpoint. This strategy greatly reduces the size and write cost of each checkpoint, though it requires precise tracking of parameter updates. Differential checkpointing, instead, records all changes accumulated since the last full baseline. While this still produces smaller checkpoints than a full snapshot, the checkpoint size grows steadily with training time, necessitating periodic baseline refreshes to bound storage overhead [10].

Checkpoint Loading. Recovery from incremental checkpoints requires replaying a chain – the system first loads the baseline and then applies all subsequent incremental checkpoints in order until reaching the target state. Long checkpoint chains can thus introduce high recovery latency and operational complexity. Recovery from differential checkpoints is simpler, as the system only needs to load the baseline and the corresponding differential checkpoint, thereby avoiding repeated loading of the same embeddings.

As shown in Fig. 1, embedding access exhibits a strong skew pattern. We further analyze the characteristics of incremental checkpointing and differential checkpointing under embeddings with different access frequencies, and we find that both approaches show limitations in handling such skew, as depicted in Fig. 3. To quantify the effectiveness, we compare the total cost of recovery after a failure at the 3rd checkpoint, which includes the cumulative save cost of the first three checkpoints and the load cost at the 3rd checkpoint. The results show that for cold embeddings, differential checkpointing leads to higher total cost because its cumulative strategy repeatedly stores embeddings that are rarely updated, resulting in excessive storage overhead. In contrast, for hot embeddings, incremental checkpointing yields higher total cost due to its chained loading strategy, which repeatedly reloads embeddings that are frequently updated, leading to excessive recovery overhead.

2.3 Opportunities for Checkpointing Optimization

The access skewness of embeddings and the unique locality of updates during DLRMs training present significant opportunities for optimizing checkpointing mechanisms.

As shown in §2.2, neither a standalone differential nor an incremental checkpointing strategy can achieve optimality

for embeddings with varying access frequencies. Differential checkpointing is better suited for frequently accessed embeddings to expedite fault recovery, whereas incremental checkpointing offers a significant storage cost advantage when handling infrequently accessed embeddings. This observation suggests that by differentiating embeddings based on their access frequency and adopting a hybrid strategy, we can judiciously combine the strengths of both approaches, thereby overcoming the inherent performance bottlenecks of any single checkpointing paradigm.

Recommendation models exhibit another critical characteristic during training: update locality. In each training batch iteration, the model only needs to update a small subset of parameters (relative to the entire checkpoint). This unique property provides a significant opportunity for us to design efficient checkpointing strategies. We optimize checkpointing by adopting a "Save-on-Update" (SoU) approach, similar to the "Copy-on-Write" (CoW) [28] technique in operating systems. A checkpoint does not immediately copy all data, but shares the existing parameters with the training process. A physical copy is made only if those parameters are modified in a later stage. This allows asynchronous checkpoint saves in the background, without the overhead from the main training timeline.

2.4 Research Challenges

Efficient, robust checkpointing for online DLRMs is challenging and faces significant practical obstacles.

- **Customizing individual checkpointing strategy for hot and cold embeddings**. Although embedding access frequencies can in principle be derived from historical data for offline training, full-scale analysis is prohibitively expensive for large DLRMs. In online training, constantly changing user behavior further prevents reliable estimation of embedding frequencies in advance, posing a major obstacle to hybrid checkpointing. Even with real-time frequency data, it remains difficult to choose effective thresholds to distinguish *hot* from *cold* embeddings for an optimal hybrid strategy.
- **Pipelining training and saving**. The "Save-on-Update" strategy significantly reduces parameter volume requiring synchronous pauses for saving and allows these parameters to be rapidly serialized into memory. Nevertheless, in practical deployments with limited memory resources and write bottleneck of remote storage, these parameters have to enqueue and await the persistence of preceding data before the release of memory resources. This inevitably introduces training stalls and drastically degrades the overall training throughput. Hence, how to optimize the write pipeline to minimize training obstruction and ultimately achieve complete concealment of save latency within the training process is the key challenge to fulfill non-blocking checkpointing.

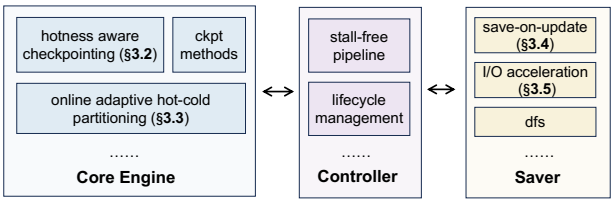


Figure 4. AdaptCP's core components.

3 AdaptCP Design

3.1 Design Overview

AdaptCP is a high-performance checkpointing system designed for DLRM model training, and is compatible to, and integrated with the existing parameter server training paradigm. Fig. 4 shows the core components in AdaptCP.

- **Controller** acts as central coordinator and manages the entire checkpointing lifecycle: triggering checkpoints based on pre-defined policies, controlling the generation of base checkpoints, periodically resetting system states (e.g., clearing the *tracker* and *access counter*), and conducting garbage collection of obsolete checkpoint files.
- **Core Engine** determines what and how to checkpoint. It comprises a *tracker* that precisely identifies the modified embedding vectors within each epoch, and an *access counter* that records access frequency of each embedding. A *threshold solver* is devised to calculate the optimal threshold to partition embeddings into hot and cold sets as per the access frequency. The engine also uses the threshold to navigate the pathway of different checkpointing strategies.
- **Saver** is a runtime worker that fully decouples checkpointing from training. It runs *save* tasks from upper layers in background threads to persist embeddings to remote storage, prioritizing those that will be updated soon to complete critical operations promptly and keep checkpointing non-blocking. A Persistence Writer implements the backend writing pipeline: data is serialized, buffered in memory, then permanently written to remote storage.

3.2 Hotness Aware Checkpointing

Existing checkpointing strategies perform poorly on large-scale embedding tables, due to skewed access hotness (i.e., imbalanced access frequency). In reality, no single policy can optimally handle both high- and low-frequency items, which leads to excessive I/O and storage overhead and suboptimal recovery latency. To this end, our key insights to improve checkpointing performance is splitting embeddings into *hot* and *cold* regions dynamically whilst elaborating individual checkpointing strategies for each of them correspondingly. Through theoretical and simulation-based analysis, we show that a threshold of differentiating cold and hot embeddings exists that minimizes total checkpointing cost.

3.2.1 Checkpointing Strategies. We rank M embeddings by access frequency in descending order and choose the most suitable checkpointing strategy based on a threshold N ($0 \leq N \leq M$). The top-N most frequently accessed embeddings come into the hot set H_N, while the cold set L_N consists of the remaining $M - N$ embeddings.

Given that *hot* embeddings are updated almost every checkpointing interval, using the *differential* strategy can ensure the fastest recovery through solely loading the base checkpoint and a target differential file. On the contrary, we utilize incremental checkpointing for the cold embeddings, considering the nature of sparse update, to reduce the amount of data saved during each checkpoint.

3.2.2 Modeling Total I/O Cost of Checkpointing. To validate the checkpointing effectiveness, we formulate a model of the total system-wide I/O cost, from the time when the training starts to the time when a failure occurs at the n-th checkpointing interval and when the recovery begins.

We define the following notations.

- M: The total number of unique embeddings.
- $P = \{p_1, p_2, \ldots, p_M\}$: The set of access frequencies for all embeddings, in descending order ($p_1 \geq p_2 \geq \cdots \geq p_M$).
- k: Total number of embedding accesses within each checkpointing interval.
- n: The target recovery checkpoint number of interest.
- $E(K, P')$: Expectation function that represents the expected number of unique embeddings in a frequency subset P' that are accessed at least once given a total access count of K. For an embedding with access frequency p, the probability of not being accessed in K independent trials is $(1-p)^K$, so the probability of being accessed at least once is $1-(1-p)^K$. By the linearity of expectation, $E(K, P')$ is the sum of these probabilities for all $p \in P'$, i.e., $E(K, P') = \sum_{p \in P'}(1-(1-p)^K)$.

We decompose the total cost into the loading/recovery cost $L_n(N)$ to restore the n-th checkpoint and the cumulative saving cost $S_n(N)$ from the 1st to the n-th checkpoint.

Loading Cost $L_n(N)$. As loading the base checkpoint represents a fixed cost inherent to any recovery operation, we analyze the costs for retrieving the hot and cold sets upon the base. For the hot set H_N, recovery involves loading the base checkpoint and the n-th differential checkpoint. The additional cost for the hot set is equivalent to the expected number of all hot embeddings accessed up to the n-th interval: $E(n \cdot k, H_N)$. For the cold set L_N, which uses incremental checkpointing, recovery requires loading the base checkpoint and all n incremental checkpoints. The additional cost for the cold set is n times the expected number of cold embeddings contained in a single incremental checkpoint: $n \cdot E(k, L_N)$. Thus, the total loading cost is given by:

$$L_n(N) = E(n \cdot k, H_N) + n \cdot E(k, L_N) \tag{1}$$

Cumulative Saving Cost $S_n(N)$. For the hot set H_N, the size of the differential checkpoint at the s-th save is $E(s \cdot k, H_N)$. Hence, the cumulative cost is $\sum_{s=1}^{n} E(s \cdot k, H_N)$. For the cold set L_N, the size of each incremental checkpoint saved is $E(k, L_N)$, resulting in a cumulative cost of $n \cdot E(k, L_N)$. Therefore, the total cumulative saving cost is:

$$S_n(N) = \sum_{s=1}^{n} E(s \cdot k, H_N) + n \cdot E(k, L_N) \tag{2}$$

Total Cost. Putting together, the total I/O cost $C(N)$ is:

$$C(N) = E(n \cdot k, H_N) + \sum_{s=1}^{n} E(s \cdot k, H_N) + 2n \cdot E(k, L_N) \tag{3}$$

3.2.3 Existence of a Cost-Minimizing Threshold. To investigate the possibility of reaching the minimal cost by using checkpointing strategy, we dive into an in-depth analysis. We decompose $C(N)$ into independent contributions from each embedding p_i:

$$\begin{aligned} C(N) &= \sum_{i=1}^{N} \left(E(n \cdot k, p_i) + \sum_{s=1}^{n} E(s \cdot k, p_i) \right) + \sum_{i=N+1}^{M} 2n \cdot E(k, p_i) \\ &= \sum_{i=1}^{N} C_H(p_i) + \sum_{i=N+1}^{M} C_L(p_i). \end{aligned} \tag{4}$$

$C_H(p_i)$ and $C_L(p_i)$ represent the individual cost contribution when embedding p_i is classified as high-frequency or low-frequency, respectively.

$$C_H(p_i) = (1-(1-p_i)^{nk}) + \sum_{s=1}^{n} (1-(1-p_i)^{sk}) \tag{5}$$

$$C_L(p_i) = 2n \cdot (1-(1-p_i)^k) \tag{6}$$

Assume that the N-th embedding p_N transitions from the cold set to the hot set when the threshold increases from $N-1$ to N. The corresponding change in total cost, or marginal cost, $\Delta(p_N)$, can be expressed as:

$$\Delta(p) = C(N) - C(N-1) = C_H(p_N) - C_L(p_N) \tag{7}$$

We further analyze the behavior of $\Delta(p)$ at both ends of the frequency distribution.

High-Frequency Hot Embeddings (large p values): In large-scale training scenarios, k is typically very large (e.g., hundreds of millions or more). This implies that even for embeddings that are not extremely high-frequency, the condition $p_i k \gg 1$ often holds. Under this condition, we approximate $(1-p_i)^k \approx e^{-p_i k} \to 0$ and hence:

$$C_H(p_i) \approx (1-0) + \sum_{s=1}^{n} (1-0) = 1 + n \tag{8}$$

$$C_L(p_i) \approx 2n \cdot (1-0) = 2n \tag{9}$$

The marginal cost will be:

$$\Delta(p) \approx (1+n) - 2n = 1 - n \tag{10}$$

Given that we are considering recovery scenarios after multiple checkpoints ($n > 1$), it follows that $\Delta(p) < 0$. This indicates that for high-frequency embeddings, assigning them to the hot set can reduce the total cost.

Low-Frequency Cold Embeddings ($p \to 0$): Once p approaches zero, we can use the first-order Taylor expansion approximation $1-(1-p_i)^K \approx Kp_i$. Hence, $C_H(p_i)$ and $C_L(p_i)$ are as follows.

$$C_H(p_i) \approx (nkp_i) + \sum_{s=1}^{n}(skp_i) = kp_i\left(n + \frac{n(n+1)}{2}\right) \quad (11)$$

$$C_L(p_i) \approx 2n(kp_i) \quad (12)$$

As a result, the marginal cost will be:

$$\Delta(p) \approx kp_i\left(n + \frac{n(n+1)}{2} - 2n\right) = kp_i\frac{n(n-1)}{2} \quad (13)$$

Given the conditions $n > 1, k > 0$, and $p > 0$, we derive $\Delta(p) > 0$. This implies that classifying low-frequency embeddings into hot set will increase the total cost.

Concavity Analysis of $\Delta(p)$. We analyze the marginal cost function $\Delta(p)$ in detail. We perform a variable substitution $x = (1-p)^k$, so that $x \in (0,1)$ for $p \in (0,1)$ and $k > 0$.

$$C_H(x) = (1-x^n) + \sum_{s=1}^{n}(1-x^s) = 1 - x^n + n - \sum_{s=1}^{n}x^s \quad (14)$$

$$C_L(x) = 2n(1-x) = 2n - 2nx \quad (15)$$

$$f(x) = C_H(x) - C_L(x) = (1-n) + 2nx - x^n - \sum_{s=1}^{n}x^s \quad (16)$$

As $p \to 0$ (i.e., $x \to 1$), we have $f(x) > 0$, while as $p \to 1$ (i.e., $x \to 0$), we have $f(x) < 0$. By the intermediate value theorem, $f(x)$ has at least one zero in $(0,1)$. Taking the second derivative, we obtain:

$$f''(x) = -n(n-1)x^{n-2} - \sum_{s=1}^{n}s(s-1)x^{s-2} \quad (17)$$

Since $x \in (0,1)$ and $n > 1$, all terms in $f''(x)$ are non-positive, with at least one strictly negative, so $f(x)$ is strictly concave on $(0,1)$ and has a unique zero. Because x and p are in one-to-one correspondence, $\Delta(p)$ likewise has a unique zero for $p \in (0,1)$.

As the embedding frequency p decreases (i.e., as the partition threshold N increases), the marginal cost function $\Delta(p)$ changes smoothly from negative to positive. Thus, there is a unique optimal threshold N^* that minimizes total I/O cost. This N^* is neither $N = 0$ (pure incremental) nor $N = M$ (pure differential), but a hybrid in between, and can be efficiently found via bisection search.

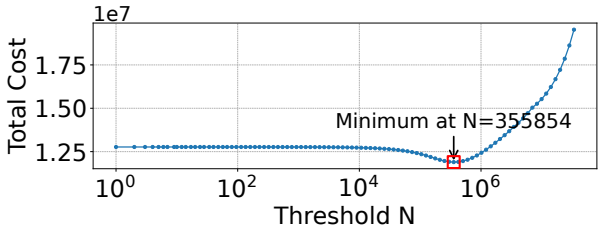


Figure 5. The impact of threshold N on the total cost.

3.2.4 Simulation-based Analysis.

As an illustrative example to showcase how the theoretical analysis works, we conducted simulation experiments using Criteo Kaggle dataset, where each sample contains 26 embedding features. We simulated the training procedure with a batch size of 8,192 and saved the checkpoints every 100 batches. This results in a total access count of $k = 26 \times 8,192 \times 100 = 21,299,200$ per checkpoint interval. To recovery from a training failure, we targeted the 5th checkpoint ($n = 5$) and measured the accumulative total cost of failures.

We varied the partitioning threshold N and, based on the collected embedding access statistics, computed the corresponding total cost $C(N)$ using the cost model in Eq. 3. As shown in Fig. 5, there exists a minimum of $C(N)$, with the lowest cost achieved at $N \approx 355{,}854$. This shows that an adaptive strategy distinguishing hot and cold embeddings can optimally balance storage and recovery cost.

3.3 Online Adaptive Hot-Cold Partitioning

3.3.1 Determining Threshold at Runtime.

Despite the theoretical existence of the optimal N^*, it is impractical to pre-traverse the entire dataset to pinpoint the access frequencies of all embeddings in large-scale training settings, and finding out the threshold N^* dynamically poses even more computational challenges.

We propose an adaptive method for hot-cold partitioning in an online manner, with the aid of three functional components: *tracker*, *access counter*, and *threshold solver*. The detailed process is shown in Algorithm 1. At each training step, the *tracker* records the IDs of updated embeddings, while the *access counter* accumulates their access counts (Lines 5–7). When performing an adaptive checkpoint (Lines 12–21), the *threshold solver* utilizes the theoretical conclusion from §3.2.3 to determine the optimal partition threshold via binary search over the current access frequencies sorted in descending order, locating the point where $\Delta(p)$ transitions from negative to positive. Based on the obtained threshold, the embeddings in *tracker* are partitioned into a hot set (for differential checkpointing) and a cold set (for incremental checkpointing). The cold set is removed from the *tracker* after the hybrid checkpoint to preserve the continuity of the differential chain (Line 20). Threshold solving is performed only at the first hybrid checkpoint after a base checkpoint,

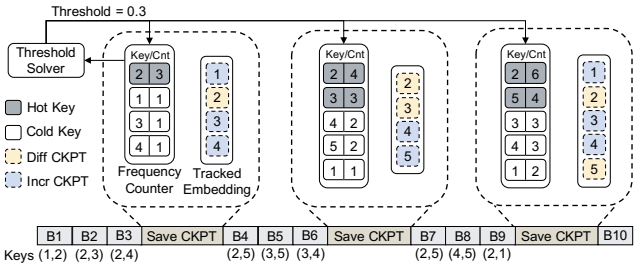


Figure 6. Working example of AdaptCP. It demonstrates how the *tracker*, *access counter*, and *threshold solver* collaboratively operate to dynamically partition embeddings into hot and cold sets across successive training steps.

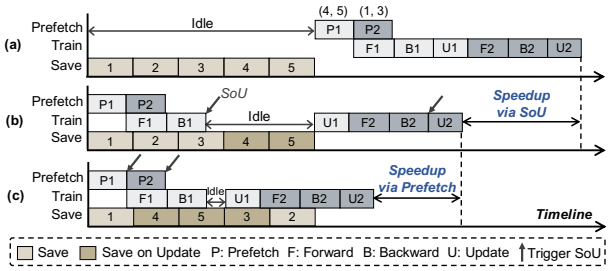


Figure 7. Examples of different saving pipelines: (a) native synchronous checkpointing, (b) our native SoU pipeline, and (c) prefetch-triggered SoU pipeline (our approach).

Algorithm 1: Online Adaptive Hot-Cold Partitioning

```
1  Embedding Tracker T = ∅;
2  Access Counter C = {};
3  Threshold T_p = 0;
4  for each training step do
5      id_updated = TrainOnBatch();
6      T.add(id_updated);
7      C[id_updated] += 1;
8      if need save ckpt then
9          if need save base ckpt then
10             Save(method=base);
11             T = ∅, C = {}, T_p = 0;
12         else
13             if is first ckpt after base then
14                 T_p = ThresholdSolver(C);
15             end
16             S_hot = { id ∈ T | C[id] / Σ_j C[j] ≥ T_p };
17             S_cold = T − S_hot;
18             Save(method=differential, set=S_hot);
19             Save(method=incremental, set=S_cold);
20             T = S_hot;
21         end
22     end
23 end
```

as the access pattern is generally stable within a base checkpoint period. After saving a base checkpoint, both *tracker* and *access counter* are reset to adapt to changes in the access frequency distribution (Line 11).

To resume the training procedure, the system loads the base checkpoint and sequentially applies the incremental parts from each intermediate hybrid checkpoint. The differential parts of the targeted recovery point are finally loaded to completely reconstruct the embedding state.

3.3.2 Working example. Fig. 6 illustrates how the adaptive checkpointing works. In this simplified example, training starts from a base checkpoint. At the first checkpoint, updates from Batches 1–3 are 1, 2, 2, 3, and 2, 4, giving tracker 1, 2, 3, 4. With the optimal threshold 0.3, only key 2 (frequency 0.5) is classified as a hot key (differential), while the others are cold keys (incremental). After removing the cold keys, the tracker becomes 2. After Batches 4–6 (2, 5, 3, 5, 3, 4), the tracker expands to 2, 3, 4, 5. With the same threshold, keys 2 and 3 are hot keys (differential), and the rest are cold keys (incremental). After removing the cold keys, the tracker becomes 2, 3. After Batches 7–9 (2, 5, 4, 5, 2, 1), the tracker grows to 1, 2, 3, 4, 5. Applying the threshold again, keys 2 and 5 are hot keys (differential), and the others are cold keys (incremental). After removing the cold keys, the tracker becomes 2, 5.

3.4 Stall-Free Checkpointing via Save-on-Update

Even with adaptive checkpointing, frequent saves can still stall training and reduce throughput. This is because all traditional schemes — full, incremental, or differential – require halting the training process to ensure data consistency. These interruptions become prohibitively expensive at high frequencies, especially for large-scale DLRMs. As shown in Fig. 7a, saving a checkpoint for keys (1, 2, 3, 4, 5) necessitates a complete pause until all data is persisted, and thus blocks the holistic training progress.

3.4.1 Save-on-Update pipeline. We address these stalls with a Save-on-Update (SoU) mechanism, inspired by the operating-system concept of Copy-on-Write (CoW). SoU exploits the inherent sparsity of DLRM training – where each batch updates only a small fraction of parameters – to perform checkpointing asynchronously. Instead of blocking training for a full saving, SoU operates at a fine granularity: it intercepts parameters only at the moment that they are updated, snapshots their old values, and then allows the update and training to continue. This means the unmodified parameters do not need to be explicitly copied, as they are shared between the checkpoint and the ongoing training process. As shown in Fig. 7b, SoU transforms checkpointing into a fine-grained, asynchronous pipeline. Consider two consecutive batches: Batch 1 accesses keys (4, 5), while Batch 2 accesses keys (1, 3). Training proceeds uninterrupted, and only incurs

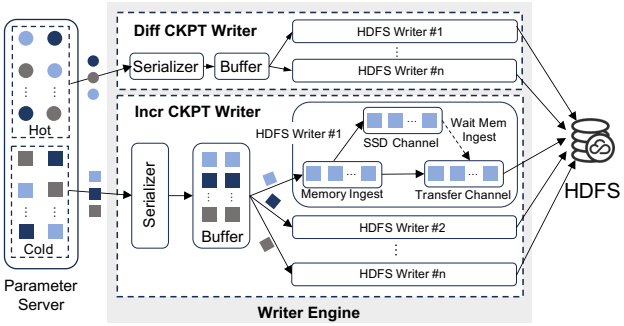


Figure 8. Design of the Writer Engine.

minimal suspension at the parameter update stage after the forward and backward passes (e.g., for keys 4 and 5). This approach dramatically reduces stall time, compared with synchronous checkpointing.

3.4.2 Prefetch-triggered SoU optimization. While SoU can alleviate the stalls derived from synchronous checkpointing, it still requires parameters to be serialized into memory just before they are updated. Parameter serialization, while fast, can cause SoU to block whilst waiting for the previous saves to complete before resource free due to slow remote-storage writes. To hide the residual latency in SoU, we integrate SoU with embedding prefetching. The system triggers a high-priority saving procedure for parameters immediately after they are prefetched for a training batch. This provides a much larger time window for the save to complete in the background and, consequently, the training thread rarely needs to suspend. As illustrated in Fig. 7c, saving keys (4, 5) begins right after prefetching for Batch 1. By the update phase, only key 5's save is ongoing, resulting in a minimal waiting time. This optimization significantly reduces training stalls beyond the native SoU mechanism.

3.4.3 SoU Implementation. When a checkpoint is triggered, the system first collects the embedding IDs to be saved according to the hybrid checkpointing policy and locates their corresponding addresses in the embedding table. At this point, a new background thread is launched to carry out the checkpoint saving, while the foreground training process continues without interruption.

The system manages background saving with two queues. A low-priority queue holds addresses of all embeddings to be checkpointed. When prefetch triggers SoU, the related embeddings move from the low-priority to a high-priority queue. The background saver continuously dequeues entries for serialization and storage, always draining the high-priority queue first; if it is empty, it uses the low-priority queue. Embeddings are processed in batches, enabling smooth asynchronous checkpointing while prioritizing soon-to-be-updated embeddings to minimize training stalls.

3.5 I/O Acceleration for Remote Storage

While SoU prevents direct blocking via asynchronous operations, transferring massive serialized embedding data to a remote DFS like HDFS can still create a bottleneck, constrained by buffer capacity and bandwidth. To overcome this, we introduce a dedicated *Writer Engine* to accelerate I/O. As shown in Fig. 8, its core is a multi-stage, parallel write pipeline that leverages local SSD caching to alleviate memory pressure and accelerate the checkpoint saving. The engine maintains separate pipelines for differential and incremental checkpoints. Each pipeline comprises: a *Serializer* that converts data into byte streams; a *Memory Buffer Channel* that decouples serialization from I/O; and multiple *HDFS Writers* that perform parallel writes to HDFS.

3.5.1 SSD-based Backpressure Mitigation. To mitigate backpressure caused by slower HDFS write speeds, the HDFS Writer employs local SSDs as an intermediate buffer. Data is streamed concurrently to HDFS and the SSD. The SSD absorbs the excess data that HDFS cannot immediately consume, thereby protecting upstream modules from slowdowns. After the initial memory flush, the system sequentially transfers the remaining SSD-buffered data to HDFS. This design ensures that the SSD handles only writes or reads at any given time, eliminating read/write contention and optimizing overall throughput.

3.5.2 Writing Parallelization. To save a checkpoint efficiently, AdaptCP launches multiple HDFS Writer instances in parallel. Each instance operates independently with its own buffers and HDFS connections, aggregating I/O bandwidth to maximize throughput and prevent single-stream bottlenecks. As the resulting multiple files can be loaded in parallel, the parallelization not only ensures rapid checkpoint saving but also inherently improves recovery speed.

4 Experiments

In this section, we present a comprehensive experimental evaluation to demonstrate the effectiveness of AdaptCP in large-scale checkpointing for deep learning-based recommendation systems. We compare AdaptCP against state-of-the-art checkpointing strategies and conduct ablation studies to assess the contribution of its core components.

4.1 Experimental Setup

Baselines. We select two widely used checkpoint strategies as baselines[1]. *Check-N-Run* [10] combines differential checkpointing with periodic full checkpoints, while native *Incremental Checkpointing*(*Incr CKPT*)[2] saves only updated

[1]We re-implemented such strategies in our system for a fair comparison.
[2]We do not directly compare AdaptCP against IncrCP [19] because it requires maintaining a 2-D chunk data structure in storage and updating it at every checkpoint. This approach is incompatible with our HDFS-based storage and, at industry scale, requires managing multi-terabyte data structures.

parameters since the last checkpoint, but requires sequential loading of all prior checkpoints during recovery, which may increase latency. In addition, to strengthen the evaluation, we re-implemented an *Incremental Checkpointing* version of the *Check-N-Run* baseline, denoted as *Incr CKPT-N*, which, like *Check-N-Run*, also periodically performs a base checkpoint.

Workloads. We performed experiments under both offline and online training scenarios. Offline experiments use the Criteo Terabyte dataset [14] with 214,876,851 embedding keys, trained on the *DLRM* model [24] (128-dim embeddings, Adam optimizer [13], FP16), requiring approximately 150 GB for parameters and optimizer states. Online experiments use a production dataset with nearly 4 billion embedding keys, trained on a customized model with the AdamW optimizer [20] in FP16, requiring about 20 TB storage.

Hardware. All experiments run on machines equipped with 126-core CPUs, 2 TB DDR4 RAM, 3.5 TB NVMe SSDs, and two NVIDIA GPUs based on the Ampere architecture. Offline training on Criteo Terabyte dataset is performed on a single node, whereas the online production dataset is trained on a 10-node cluster. For persistent checkpoint storage, we use HDFS in all cases.

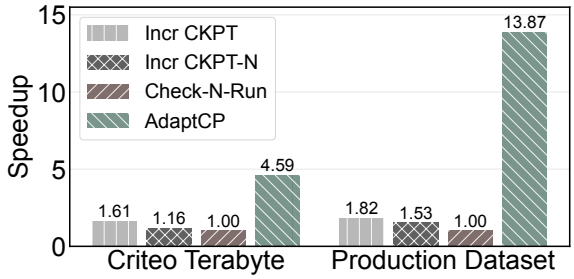
Evaluation Metrics. We assess performance using three key metrics. *Training throughput* measures the number of samples processed per second during training with periodic checkpointing, capturing end-to-end efficiency. *Recovery time* quantifies the duration required to reload parameters from the checkpoint(s) after a training failure. *Total cost* refers to the I/O overhead incurred for a one-time recovery starting from the beginning of training, defined as $\mathcal{S}_{cost} = \sum_{i=1}^{k-1} C_{save}^{i} + C_{load}^{k}$, where k indicates the checkpoint index where failure occurs; C_{save}^{i} and C_{load}^{i} denote the number of parameters saved for or loaded from the checkpoint at i^{th}, respectively.

Parameter Settings. For offline training experiments, we set the batch size per GPU to 131,072. For online training experiments, the batch size per GPU is 120. Checkpoints are performed every 1,000 batches in both scenarios. All of *Check-N-Run*, *Incr CKPT-N*, and AdaptCP save a base checkpoint every 8 checkpoints. Within our Writer Engine, the Memory Buffer Channel is uniformly set to 10 GB, consistent with actual production environments. The HDFS Writer parallelism in the Writer Engine is configured to 8 for offline training experiments and 16 for online training experiments.

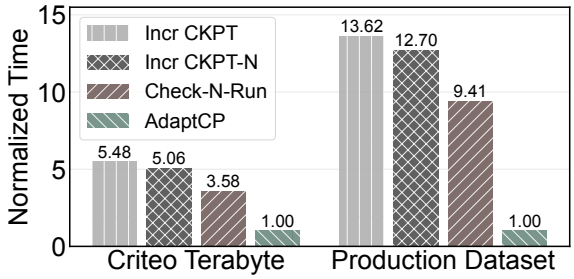
4.2 Overall Performance

This section evaluates AdaptCP for training throughput and failure recovery, compared to standard checkpointing. Experiments include offline training with the Criteo Terabyte dataset and a large-scale online training workload.

Training Throughput. Figure 9a compares the impact of different checkpointing strategies on training throughput.



(a) Training throughput speedup comparison (normalized to *Check-N-Run*).



(b) Failure recovery load time comparison (normalized to AdaptCP).

Figure 9. Overall performance evaluation of AdaptCP.

Check-N-Run suffers from steadily growing differential files between base checkpoints, and its checkpointing mechanism can still block training. In our experimental environment, embedding states fully reside in CPU memory: each machine maintains TB-scale parameters, but only GB-scale CPU memory is available for checkpoint buffering. As a result, most checkpoint data still has to wait for persistence to remote HDFS, so training cannot be fully decoupled from checkpoint persistence as in the original *Check-N-Run* setting. *Incremental checkpointing* reduces file growth by saving only parameters modified since the last checkpoint, improving throughput by 1.61× and 1.82× on Criteo Terabyte and production, but still incurs blocking stalls that cap performance. *Incr CKPT-N* periodically saves a base checkpoint in addition to the native incremental checkpoints, resulting in a larger checkpoint volume than native incremental checkpointing. Consequently, its throughput improves by only 1.16× and 1.53× over *Check-N-Run* on the Criteo Terabyte and production datasets, respectively. Compared to *Check-N-Run*, AdaptCP achieves 4.59× and 13.87× speedup on the two datasets, respectively. Its advantage comes from three synergistic designs: a hybrid strategy that significantly reduces checkpoint volume, the Save-on-Update pipeline that overlaps saves with training, and a parallel I/O engine with SSD buffering that minimizes visible checkpoint latency. Overall, AdaptCP effectively eliminates checkpoint-induced I/O stalls across both offline and online settings, sustaining high throughput under diverse scales and access patterns.

Table 1. Individual contribution to training throughput.

Adapt	SoU	Writer Engine	Throughput (samples/s)	
			Criteo Terabyte	Production Dataset
–	–	–	112040	112.4
✓	–	–	182734 (1.63×)	205.4 (1.83×)
✓	✓	–	366111 (3.27×)	374.3 (3.33×)
✓	✓	✓	514301 (4.59×)	1559.2 (13.87×)

Failure Recovery Load Time. Fig. 9b compares recovery load times normalized to AdaptCP. On Criteo Terabyte, *Check-N-Run*, *Incr CKPT-N*, and native *Incremental Checkpointing* are 3.58×, 5.06×, and 5.48× slower, respectively; in production, they are 9.41×, 12.70×, and 13.62× slower, making them impractical for latency-sensitive services. Although AdaptCP loads slightly more parameters than *Check-N-Run* (Fig. 10), its *parallel loading engine* concurrently fetches multiple shards from remote storage. This design maximizes read bandwidth, removes single-stream I/O bottlenecks, minimizing recovery time even at industrial scale.

Performance Breakdown. As shown in Table 1, enabling the Adapt checkpointing strategy, which adaptively partitions parameters into hot and cold subsets and applies differential and incremental checkpointing, respectively, increases training throughput by 1.63× on the Criteo Terabyte dataset and 1.83× on the production dataset over the unoptimized baseline. Adding the Save-on-Update (SoU) pipeline on top of Adapt further improves throughput to 3.27× and 3.33× of the baseline on the Criteo Terabyte and production datasets, respectively. These enhancements come from fully overlapping checkpoint saves with training, eliminating synchronous stalls and keeping GPUs continuously utilized. Adding the high-performance Writer Engine to Adapt and SoU further boosts throughput to 4.59× on the Criteo Terabyte dataset and 13.87× on the production dataset. The larger production gains show that, for industrial-scale checkpoints, background I/O throughput is the main bottleneck. By parallelizing writes and using SSD buffering, the Writer Engine completes saves within the compute window, minimizing I/O spillover and training downtime. Overall, these results show that AdaptCP's high throughput comes from the combined effect of its optimizations: Adapt reduces the parameters to be saved, SoU removes the architectural cause of stalls, and the Writer Engine eliminates remaining I/O bottlenecks.

4.3 Effectiveness of Hybrid Checkpointing Strategy

We evaluate the I/O characteristics of AdaptCP and three baselines over an entire checkpoint lifecycle, focusing on *per-checkpoint save cost*, *recovery load cost*, and *total I/O cost*. This validates the theoretical model in § 3.2 and examines how AdaptCP balances save efficiency and recovery speed.

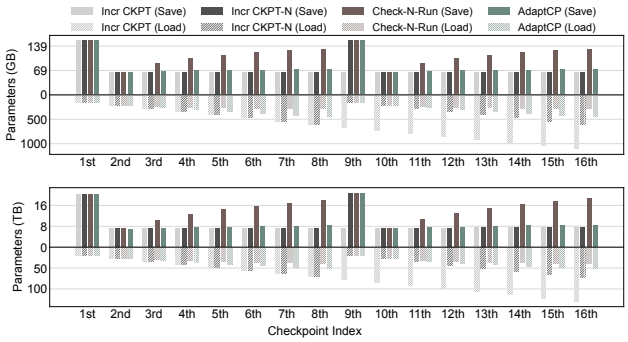


Figure 10. I/O characteristics over a checkpoint lifecycle for different strategies. The top sub-figure shows per-checkpoint save volume (above x-axis) and recovery load volume (below x-axis) on the Criteo Terabyte dataset; the bottom sub-figure shows the same metrics for a production dataset.

Per-Checkpoint Save Cost. The top panels of Fig. 10 plot save data volume by checkpoint index for the Criteo Terabyte dataset and the production dataset, respectively. *Incremental checkpointing* has the lowest and most stable cost since it writes only updated parameters. *Incr CKPT-N* behaves identically to native incremental checkpointing except for periodically saving a base checkpoint, resulting in slightly higher save volumes. *Check-N-Run* shows steadily increasing cost because differential data accumulates against a fixed baseline, growing nearly linearly between baselines. AdaptCP matches *incremental checkpointing* for most cold data while using differential saves for a small hot-data partition, producing an almost flat curve with only a slight increase over time due to hot-data updates. In production experiments, slight variations in baseline checkpoint size result from unavoidable differences in data consumption across strategies, a normal outcome in online training.

Recovery Load Cost. The bottom panels of Fig. 10 show recovery data volumes for the Criteo Terabyte and production datasets. *Check-N-Run* loads only a baseline and one delta, yielding the slowest cost growth. *Incremental checkpointing* must load all increments from the baseline to the target, causing near-linear growth and much higher cost. AdaptCP retains fast hot-data recovery from differential checkpointing, while cold-data increments stay small due to sparse updates. *Incr CKPT-N* lowers recovery cost compared to native incremental checkpointing via periodic base checkpointing, but still must load all increments from the latest baseline to the target, keeping its cost well above *Check-N-Run*. Its load cost is only slightly higher than *Check-N-Run* and far lower than *incremental checkpointing*.

Total I/O Cost. Figs. 11 present the sum of cumulative save cost and current load cost, measuring the total I/O from the start of training to recovery. Results match § 3.2.3: AdaptCP

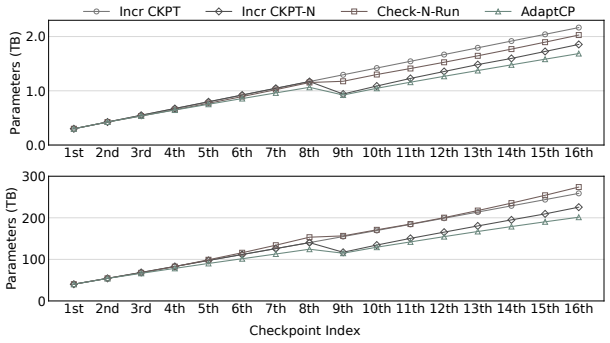


Figure 11. Total I/O cost of different checkpointing strategies. The top and bottom sub-figures present the results for the Criteo Terabyte and for the production dataset.

Table 2. Impact of SoU on per-iteration training time.

Configuration	Time / Iteration
No-Save	544 ms
SoU-Enabled	545 ms (+0.18%)

achieves the lowest total cost for both datasets, while *Check-N-Run* is limited by high cumulative save cost, and *incremental checkpointing* and *Incr CKPT-N* are constrained by high recovery cost. As training progresses, AdaptCP 's advantage grows, demonstrating that dynamically partitioning embeddings into hot and cold sets and applying matching checkpointing schemes avoids the extreme drawbacks of single strategies, achieving a near-optimal trade-off between save efficiency and recovery speed.

4.4 Micro Experiments

Impact of SoU on Training Performance. A potential concern with the SoU pipeline is that, even though it is designed to run asynchronously, its background I/O might still compete with training for shared resources (e.g., CPU cycles and memory bandwidth), which could slow down each training iteration. To measure this impact, we run an isolation experiment on Criteo Terabyte dataset, comparing the average per-iteration training time for: i) No-Save, where training runs without any checkpointing; and ii) SoU-Enabled, where training runs while AdaptCP's SoU continuously saves checkpoints in the background. As shown in Table 2, SoU adds only a 1 ms overhead (about 0.18%), well within normal system noise, demonstrating that AdaptCP preserves performance isolation by confining checkpointing to a background thread pool.

Impact of Checkpointing Frequency. Checkpointing frequency trades off recovery time against training efficiency: more frequent checkpoints reduce data loss but increase I/O overhead. We evaluate AdaptCP's robustness to high-frequency checkpointing on the Criteo Terabyte dataset,

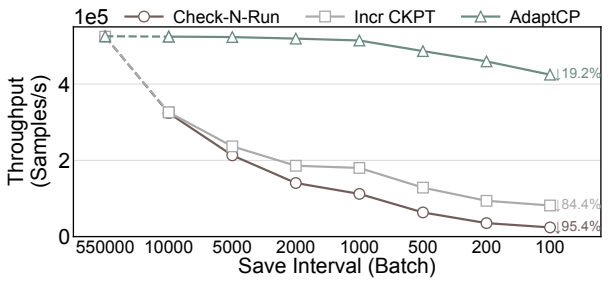


Figure 12. Impact of checkpointing frequency on training throughput.

increasing the frequency from every 550,000 to every 100 batches. As shown in Fig. 12, AdaptCP maintains the highest throughput across all practical checkpointing frequencies. Checkpointing ceases to be a bottleneck only at an interval of approximately 550,000 batches, where it accounts for less than 1% of training time. However, this corresponds to more than 80 hours between checkpoints, defeating the purpose of checkpointing. With checkpointing every 10,000 batches, nearly all embeddings are accessed between checkpoints, making each save effectively a full snapshot. Consequently, *Check-N-Run* and native *Incremental Checkpointing* exhibit similar throughput due to their synchronous checkpointing designs. In contrast, AdaptCP achieves approximately 1.6× higher throughput by overlapping checkpointing with training through its SoU pipeline and parallel writer engine. At every 100 batches, relative to the 550,000-batch baseline, the throughput of *Check-N-Run* and incremental checkpointing drops sharply by 95.4% and 84.4%, respectively. This stems from their synchronous blocking designs, where each save pauses training and higher frequency magnifies cumulative stalls. In contrast, AdaptCP's throughput drops by only 19.2% because its SoU pipeline overlaps saves with computation, and its parallel writer engine finishes most saves within the available window even at high frequency. This resilience enables operators to use more frequent checkpoints to lower recovery point objectives without significantly reducing throughput, improving reliability for large-scale training.

Impact of HDFS Writer Parallelism. We evaluated how write parallelism affects end-to-end training throughput on the Criteo Terabyte dataset, varying concurrent write streams from 1 to 32. As shown in Fig. 13, throughput rises as parallelism increases from 1 to 4, remains near its peak at 8 streams, then declines beyond 8. The initial gains come from better network bandwidth utilization, while the decline at higher parallelism is due to increased CPU scheduling overhead from excessive concurrent threads and context switches, making checkpointing a performance bottleneck.

Validation of the Optimal Frequency Threshold. In §3.2, we theoretically proved that in the hybrid checkpointing

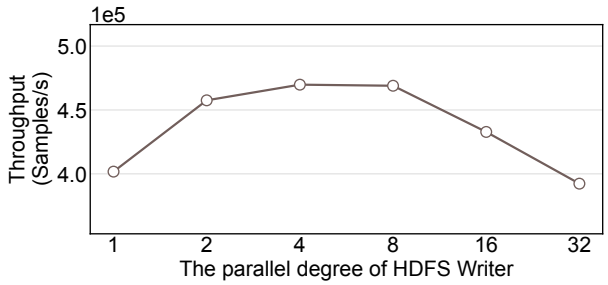


Figure 13. Impact of HDFS Writer parallelism on training throughput.

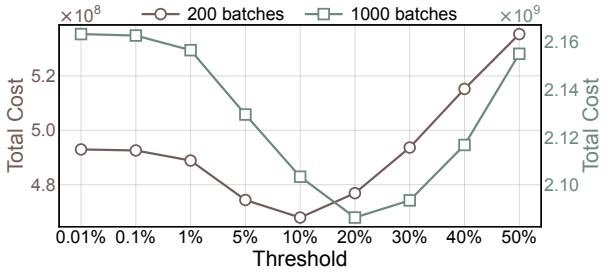


Figure 14. Validation of the optimal frequency threshold for adaptive checkpointing.

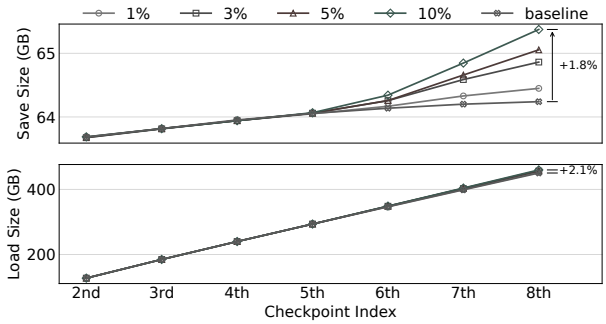


Figure 15. Impact of hotness shifting on checkpoint save and recovery load volumes.

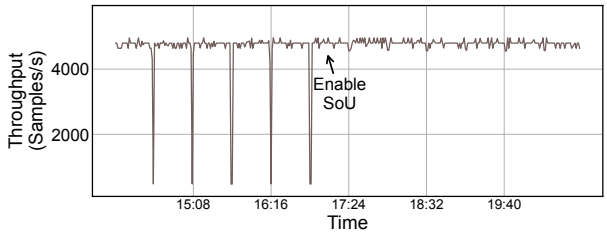


Figure 16. Production-grade online DLRM training with AdaptCP; throughput is reported every 30 minutes.

strategy, there exists an optimal frequency threshold that minimizes the cumulative total I/O cost from the start of training to failure recovery. To validate this key assertion, we conduct an evaluation on the Criteo Terabyte dataset. We test two different checkpointing frequencies—once every 200 batches and once every 1000 batches—and measure the cumulative total cost when a failure occurs at the 5th checkpoint, under varying frequency threshold settings. As shown in Fig. 14, the results reveal a clear valley-shaped performance trend. For the 200-batch checkpointing configuration, the cumulative total cost reaches its minimum at a frequency threshold of approximately 10%; for the 1000-batch configuration, the optimal threshold shifts to around 20%. This strong alignment between empirical results and our theoretical proof confirms the existence of an optimal frequency threshold. Furthermore, the observed shift in optimal threshold under different checkpointing frequencies highlights the practical importance of dynamically or adaptively adjusting the threshold based on the specific training cycle and checkpointing policy, in order to sustain optimal performance in real deployments.

Impact of Hotness Shifting. To evaluate AdaptCP's adaptability to rapidly changing access patterns in online learning, we study hotness shifting. The baseline uses the original access distribution on the Criteo Terabyte dataset. To emulate transient user-interest shifts, after the 5th checkpoint we swap the access behavior of a fraction of the coldest embeddings with the same fraction of the hottest ones, modeling a scenario where previously cold embeddings suddenly become hot. As shown in Fig. 15, AdaptCP remains robust under varying degrees of hotness shifting. Even when 10% of embeddings shift, the save volume before the next baseline checkpoint (the 8th) increases by only 1.8% over the no-shift baseline. This small increase is absorbed by AdaptCP's SoU pipeline and Writer Engine, causing negligible impact on training throughput. The recovery load volume rises by at most 2.1%, showing that hotness shifting barely affects recovery efficiency. Thus, AdaptCP quickly adapts to hotness shifts in online learning, and even if the threshold solver does not update the partition threshold within one baseline-checkpoint interval, AdaptCP shows no noticeable performance degradation.

4.5 AdaptCP in Production

AdaptCP is deployed in production GPU clusters, providing daily training support for critical Deep Learning Recommendation Models (DLRMs) used in our various online services. These models are crucial for maintaining user engagement and enhancing business metrics, thus demanding continuous adaptability and high throughput.

Figure 16 showcases the training throughput performance of a typical online DLRM job under a real-world production workload. This job runs on a cluster of 30 machines equipped

with a total of 60 NVIDIA GPUs based on the Ampere architecture, with the model containing up to 60TB of embedding parameters. We evaluated the throughput variation when using adaptive checkpointing and Writer Engine. As can be seen, before enabling SoU, there was a very significant drop in throughput every time a checkpoint was saved. In contrast, after enabling SoU, no significant drop in training throughput was observed during checkpoint execution.

According to our statistics for this job, after enabling SoU, each checkpoint operation introduces an average pause of only approximately 10 seconds for preparing the data to be saved. Following this brief preparation, training can quickly resume, having virtually no impact on the overall training progress. Compared to the average 3-minute training pause observed when SoU was not enabled, AdaptCP has dramatically reduced the training pause time by 18 times. This extremely low overhead accounts for only 0.5% of the total training time between two consecutive checkpoints. Such a minimal impact is fully acceptable in a production environment, effectively ensuring nearly uninterrupted online training, thereby guaranteeing model freshness, service quality, and maximizing the avoidance of wasted computational resources. These results fully highlight AdaptCP's exceptional capability in achieving high-performance checkpointing in industrial-scale DLRM training scenarios.

5 Discussion

Generalization. AdaptCP can be extended to other checkpointing scenarios involving sparse model parameters. Its Save-on-Update (SoU) mechanism is particularly suitable for models with sparse update patterns, where only a small subset of parameters is modified in each iteration. For example, in Mixture-of-Experts (MoE) models [27], adaptive checkpointing could distinguish between frequently and rarely activated experts, while SoU could persist only updated expert parameters to reduce I/O overhead.

Fault recovery. Although AdaptCP reduces checkpointing stalls during training, fully decoupling checkpoint loading from training during recovery remains challenging. A promising direction is to overlap model loading with training by prioritizing embeddings that will be accessed soon, thereby reducing initialization latency and accelerating failure recovery. Efficient random access to checkpoint data is therefore an important direction for future work.

Adaptive threshold. AdaptCP adaptively determines the hot-cold threshold online to minimize checkpointing I/O cost. Since failures may occur unpredictably, integrating fault prediction techniques [17, 26] could further improve this process. For instance, AdaptCP could favor higher thresholds when the predicted failure probability is low to improve saving efficiency, and lower thresholds when failures are more likely to accelerate recovery.

Asynchronous I/O. As discussed in §4.4, excessive write parallelism may degrade performance. Asynchronous I/O techniques such as io_uring [7, 8] could help overlap I/O with computation, reduce CPU scheduling overhead, and further improve checkpointing efficiency.

6 Related Work

Checkpointing Optimizations for Deep Learning. Checkpointing is the standard fault tolerance mechanism for large-scale training, enabling rapid recovery by periodically persisting model states [21–23, 25, 30, 31, 34]. To mitigate I/O bottlenecks, existing studies primarily focus on hiding checkpointing latency through asynchronous writes or reducing the volume of data to persist. Systems such as CheckFreq [23], DeepFreeze [25], FastPersist [31], ParaCkpt [34], and DataStates-LLM [22] hide checkpointing latency by taking CPU-side snapshots and asynchronously flushing them to storage while training proceeds. However, in DLRMs, massive CPU-resident embeddings make memory snapshotting itself a throughput bottleneck, while multi-level caching approaches [21, 29] are often constrained by scarce DRAM resources. FlowCheck [12] decouples gradient collection from the main training process through switch traffic mirroring, reducing checkpoint interference with training, but it relies on high-bandwidth networks and specific switch functionalities. Other approaches reduce checkpoint size through quantization [3, 10, 16] or model sparsity in MoE architectures [4, 11]. Although effective, these methods may affect model accuracy or are limited to specific workloads. Additionally, Universal Checkpointing [18] and ByteCheckpoint [30] improve recovery portability through hardware- and parallelism-agnostic formats. However, these techniques mainly optimize the loading path and provide limited benefits for reducing checkpoint write latency.

Fault Tolerance in Recommendation Model Training. DLRM training presents unique fault tolerance challenges due to its massive scale, highly sparse embedding tables with non-uniform update patterns, and frequent reliance on remote storage. Memory-based solutions (e.g., ECRec [36], DLRover-RM [32], Bagpipe [2]) reduce recovery latency by replicating parameters within memory or across parameter servers, but at the cost of significant additional memory consumption, higher expense, and the volatile nature of memory. OpenEmbedding [6] leverages Persistent Memory (PMem) to store large-scale sparse embeddings to reduce costs, but PMem's cost remains significantly higher than traditional Hard Disk Drives (HDDs) and other remote storage media. IncrCP [19] and Check-N-Run [10] exploit the localized update characteristics of embeddings in recommendation models to perform incremental or differential checkpointing, thereby reducing the amount of data saved. However, both methods fail to address the prevalent access skewness in embeddings

and are not specifically optimized for the I/O characteristics of remote persistent storage.

7 Conclusion

In this work, we present AdaptCP, a high-performance checkpointing system tailored for large-scale deep learning recommendation models. By leveraging the inherent skewness in embedding access patterns, AdaptCP integrates differential and incremental checkpointing into a hybrid scheme that achieves an optimal balance between storage efficiency and recovery speed. Combined with our Save-on-Update pipeline and SSD-assisted parallel I/O engine, AdaptCP effectively eliminates checkpoint-induced training stalls and significantly reduces recovery latency. Evaluation on industry scale online training workloads shows that AdaptCP improves training throughput by up to 13.87× and accelerates recovery by up to 13.62× compared with state-of-the-art baselines. AdaptCP has been deployed and validated in production to underpin high-throughput and dependable recommendation model training at scale.

Acknowledgments

We would very much like to thank anonymous reviewers for their valuable comments. Special thanks must go to the overall AI platform team at Kuaishou Inc. and RAIDS Lab team at Beihang University, for their collaborative contribution and countless technical discussions. This work is supported in part by National Key R&D Program of China (No. 2024YFB4505604), in part by NSFC (No. 62402024), in part by Beijing Natural Science Foundation (No. L241050), in part by the Fundamental Research Funds for the Central Universities, and, last but not least, by Kuaishou Technology Research Fund. For any correspondence, please refer to the project lead and coordinator Prof. Renyu Yang (renyuyang@buaa.edu.cn).

References

[1] Bilge Acun, Matthew Murphy, Xiaodong Wang, Jade Nie, Carole-Jean Wu, and Kim Hazelwood. 2021. Understanding training efficiency of deep learning recommendation models at scale. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 802–814.

[2] Saurabh Agarwal, Chengpo Yan, Ziyi Zhang, and Shivaram Venkataraman. 2023. Bagpipe: Accelerating deep recommendation model training. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 348–363.

[3] Amey Agrawal, Sameer Reddy, Satwik Bhattamishra, Venkata Prabhakara Sarath Nookala, Vidushi Vashishth, Kexin Rong, and Alexey Tumanov. 2024. Inshrinkerator: Compressing Deep Learning Training Checkpoints via Dynamic Quantization. In *Proceedings of the 2024 ACM Symposium on Cloud Computing*. 1012–1031.

[4] Weilin Cai, Le Qin, and Jiayi Huang. 2025. MoC-System: Efficient Fault Tolerance for Sparse Mixture-of-Experts Model Training. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 655–671.

[5] Jianxin Chang, Chenbin Zhang, Zhiyi Fu, Xiaoxue Zang, Lin Guan, Jing Lu, Yiqun Hui, Dewei Leng, Yanan Niu, Yang Song, et al. 2023. TWIN: TWo-stage interest network for lifelong user behavior modeling in CTR prediction at kuaishou. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 3785–3794.

[6] 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.

[7] Jonathan Corbet. 2019. Ringing in a new asynchronous I/O API. https://lwn.net/Articles/776703/. Accessed: 2025-09-26.

[8] Jonathan Corbet. 2020. The rapid growth of io_uring. https://lwn.net/Articles/810414/. Accessed: 2025-09-26.

[9] Jiaxin Deng, Shiyao Wang, Kuo Cai, Lejian Ren, Qigen Hu, Weifeng Ding, Qiang Luo, and Guorui Zhou. 2025. Onerec: Unifying retrieve and rank with generative recommender and iterative preference alignment. *arXiv preprint arXiv:2502.18965* (2025).

[10] 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.

[11] Swapnil Gandhi and Christos Kozyrakis. 2024. MoEtion: Efficient and Reliable Sparse Checkpointing for Mixture-of-Experts Models at Scale. *arXiv preprint arXiv:2412.15411* (2024).

[12] Zimeng Huang, Hao Nie, Haonan Jia, Bo Jiang, Junchen Guo, Jianyuan Lu, Rong Wen, Biao Lyu, Shunmin Zhu, and Xinbing Wang. 2025. FlowCheck: Decoupling Checkpointing and Training of Large-Scale Models. In *Proceedings of the Twentieth European Conference on Computer Systems*. 1334–1349.

[13] Diederik P Kingma. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[14] Criteo Labs. 2013. Terabyte click logs. https://labs.criteo.com/2014/02/kaggle-display-advertising-challenge-dataset/. Accessed on 01/18/2025.

[15] Mu Li, David G Andersen, Alexander Smola, and Kai Yu. 2014. Communication efficient distributed machine learning with the parameter server. *Advances in neural information processing systems* 27 (2014).

[16] Wenshuo Li, Xinghao Chen, Han Shu, Yehui Tang, and Yunhe Wang. 2024. Excp: Extreme llm checkpoint compression via weight-momentum joint shrinking. *arXiv preprint arXiv:2406.11257* (2024).

[17] Yangguang Li, Zhen Ming Jiang, Heng Li, Ahmed E Hassan, Cheng He, Ruirui Huang, Zhengda Zeng, Mian Wang, and Pinan Chen. 2020. Predicting node failures in an ultra-large-scale cloud computing platform: an aiops solution. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 2 (2020), 1–24.

[18] Xinyu Lian, Sam Ade Jacobs, Lev Kurilenko, Masahiro Tanaka, Stas Bekman, Olatunji Ruwase, and Minjia Zhang. 2025. Universal Checkpointing: A Flexible and Efficient Distributed Checkpointing System for {Large-Scale}{DNN} Training with Reconfigurable Parallelism. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*. 1519–1534.

[19] Qingyin Lin, Jiangsu Du, Rui Li, Zhiguang Chen, Wenguang Chen, and Nong Xiao. 2024. IncrCP: Decomposing and Orchestrating Incremental Checkpoints for Effective Recommendation Model Training. *Proceedings of the VLDB Endowment* 18, 4 (2024), 1049–1062.

[20] Ilya Loshchilov and Frank Hutter. 2017. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101* (2017).

[21] Avinash Maurya, M Mustafa Rafique, Thierry Tonellot, Hussain J AlSalem, Franck Cappello, and Bogdan Nicolae. 2023. Gpu-enabled asynchronous multi-level checkpoint caching and prefetching. In *Proceedings of the 32nd International Symposium on High-Performance*

Parallel and Distributed Computing. 73–85.

[22] Avinash Maurya, Robert Underwood, M Mustafa Rafique, Franck Cappello, and Bogdan Nicolae. 2024. Datastates-llm: Lazy asynchronous checkpointing for large language models. In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing*. 227–239.

[23] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. 2021. {CheckFreq}: Frequent,{Fine-Grained}{DNN} Checkpointing. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 203–216.

[24] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G Azzolini, et al. 2019. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091* (2019).

[25] Bogdan Nicolae, Jiali Li, Justin M Wozniak, George Bosilca, Matthieu Dorier, and Franck Cappello. 2020. Deepfreeze: Towards scalable asynchronous checkpointing of deep learning models. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. IEEE, 172–181.

[26] Paolo Notaro, Jorge Cardoso, and Michael Gerndt. 2021. A survey of aiops methods for failure management. *ACM Transactions on Intelligent Systems and Technology (TIST)* 12, 6 (2021), 1–45.

[27] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. 2017. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538* (2017).

[28] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. 2018. *Operating System Concepts* (10th ed.). John Wiley & Sons.

[29] Foteini Strati, Michal Friedman, and Ana Klimovic. 2025. PCcheck: Persistent Concurrent Checkpointing for ML. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 811–827.

[30] Borui Wan, Mingji Han, Yiyao Sheng, Yanghua Peng, Haibin Lin, Mofan Zhang, Zhichao Lai, Menghan Yu, Junda Zhang, Zuquan Song, et al. 2025. {ByteCheckpoint}: A Unified Checkpointing System for Large Foundation Model Development. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. 559–578.

[31] Guanhua Wang, Olatunji Ruwase, Bing Xie, and Yuxiong He. 2024. Fastpersist: Accelerating model checkpointing in deep learning. *arXiv preprint arXiv:2406.13768* (2024).

[32] Qinlong Wang, Tingfeng Lan, Yinghao Tang, Ziling Huang, Yiheng Du, Haitao Zhang, Jian Sha, Hui Lu, Yuanchun Zhou, Ke Zhang, et al. 2023. DLRover-RM: Resource Optimization for Deep Recommendation Models Training in the Cloud. *arXiv preprint arXiv:2304.01468* (2023).

[33] Ruoxi Wang, Rakesh Shivanna, Derek Cheng, Sagar Jain, Dong Lin, Lichan Hong, and Ed Chi. 2021. Dcn v2: Improved deep & cross network and practical lessons for web-scale learning to rank systems. In *Proceedings of the web conference 2021*. 1785–1797.

[34] Shucheng Wang, Qiang Cao, Kaiye Zhou, Jun Xu, Zhandong Guo, and Jiannan Guo. 2024. ParaCkpt: Heterogeneous Multi-Path Checkpointing Mechanism for Training Deep Learning Models. In *2024 IEEE 42nd International Conference on Computer Design (ICCD)*. IEEE, 183–190.

[35] Jiaqi Zhai, Lucy Liao, Xing Liu, Yueming Wang, Rui Li, Xuan Cao, Leon Gao, Zhaojie Gong, Fangda Gu, Michael He, et al. 2024. Actions speak louder than words: Trillion-parameter sequential transducers for generative recommendations. *arXiv preprint arXiv:2402.17152* (2024).

[36] Tianyu Zhang, Kaige Liu, Jack Kosaian, Juncheng Yang, and Rashmi Vinayak. 2023. Efficient fault tolerance for recommendation model training via erasure coding. *Proceedings of the VLDB Endowment* 16, 11 (2023), 3137–3150.

[37] Guorui Zhou, Na Mou, Ying Fan, Qi Pi, Weijie Bian, Chang Zhou, Xiaoqiang Zhu, and Kun Gai. 2019. Deep interest evolution network for click-through rate prediction. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 33. 5941–5948.

[38] Guorui Zhou, Xiaoqiang Zhu, Chenru Song, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, Han Li, and Kun Gai. 2018. Deep interest network for click-through rate prediction. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*. 1059–1068.