Efficient Fault Tolerance for Recommendation Model Training via Erasure Coding

Tianyu Zhang
Carnegie Mellon University
Pittsburgh, PA
tianyuz2@andrew.cmu.edu

Kaige Liu
Carnegie Mellon University
Pittsburgh, PA
kaigeliu98@gmail.com

Jack Kosaian
Carnegie Mellon University
Pittsburgh, PA
jkosaian@andrew.cmu.edu

Juncheng Yang
Carnegie Mellon University
Pittsburgh, PA
juncheny@cs.cmu.edu

Rashmi Vinayak
Carnegie Mellon University
Pittsburgh, PA
rvinayak@cs.cmu.edu

ABSTRACT

Deep-learning-based recommendation models (DLRMs) are widely deployed to serve personalized content. In addition to using neural networks, DLRMs have large, sparsely-accessed embedding tables, which map categorical features to a learned dense representation. Due to the large sizes of embedding tables, DLRM training is typically distributed across the memory of tens or hundreds of nodes. Node failures are common in such large systems and must be mitigated to enable training to complete within production deadlines. Checkpointing is the primary approach used for fault tolerance in these systems, but incurs significant time overhead both during normal operation and when recovering from failures. As these overheads increase with DLRM size, checkpointing is slated to become an even larger overhead for future DLRMs, which are expected to grow. This calls for rethinking fault tolerance in DLRM training.

We present ECRec, a DLRM training system that achieves efficient fault tolerance by coupling erasure coding with the unique characteristics of DLRM training. ECRec takes a hybrid approach between erasure coding and replicating different DLRM parameters, correctly and efficiently updates redundant parameters, and enables training to proceed without pauses, while maintaining the consistency of the recovered parameters. We implement ECRec atop XDL, an open-source, industrial-scale DLRM training system. Compared to checkpointing, ECRec reduces training-time overhead on large DLRMs by up to 66%, recovers from failure up to 9.8× faster, and continues training during recovery with only a 7–13% drop in throughput (whereas checkpointing must pause).

PVLDB Reference Format:
Tianyu Zhang, Kaige Liu, Jack Kosaian, Juncheng Yang, and Rashmi Vinayak. Efficient Fault Tolerance for Recommendation Model Training via Erasure Coding. PVLDB, 16(11): 3137 - 3150, 2023.
doi:10.14778/3611479.3611514

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 16, No. 11 ISSN 2150-8097.
doi:10.14778/3611479.3611514

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/Thesys-lab/ECRec/tree/ecrec.

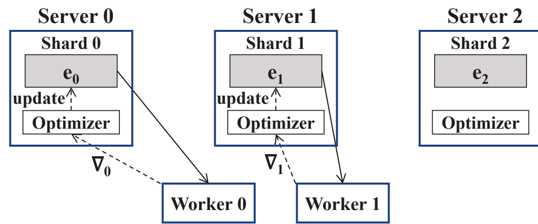
1 INTRODUCTION

Deep-learning-based recommendation models (DLRMs) are key tools in serving personalized content at Internet scale [10, 35]. As the value generated by DLRMs relies on the ability to reflect recent data, production DLRMs are frequently retrained [2]. Due to their widespread use, DLRM training occupies a large fraction of compute cycles, such as over 50% of the training demand at Facebook [34]. Reducing DLRM training time is thus critical to maintaining an accurate and up-to-date model [33], and reducing resource usage.

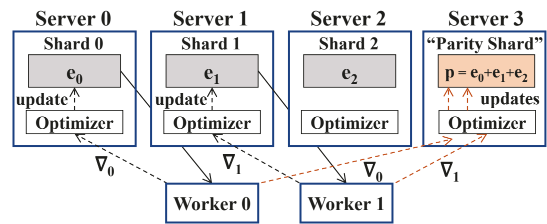
DLRMs consist of embedding tables and neural networks (NNs). Embedding tables map sparse categorical features (e.g., city name) to a learned dense vector. Embedding tables resemble lookup tables in which millions or billions [17, 21] of sparse features each map to a small dense vector of 10s/100s of floating-point values. A small NN processes dense vectors resulting from embedding table "lookups" to produce a prediction. We refer to a single dense embedding vector as an "embedding table entry," or "entry" for short.

Embedding tables are typically large, ranging from hundreds of gigabytes to terabytes in size [21]. Such large models are trained in a distributed fashion across tens/hundreds of nodes [6, 21], as depicted (at a small scale) in Fig. 1a. Embedding tables and NN parameters are sharded across *servers* and kept in memory for fast access. *Workers* operate in a data-parallel fashion by accessing model parameters from servers and sending gradients to servers to update parameters via an optimizer (e.g., Adam [12]).

Since model parameters are stored in memory, any server failure requires training to restart from scratch. Given that DLRM training is resource and time intensive and that failures are common in large-scale settings, DLRM training must be fault tolerant [16, 29]. For example, Facebook reported the median and mean times between failure for production-scale DLRM training to be 8–17 hours and 14–30 hours, respectively [29]. Fault tolerance is particularly important for DLRMs, which are frequently retrained on tight deadlines for deployment [33]. In this work, we focus on tolerating server failures. Server failures are critical to handle because they result in the loss of a fraction of the DLRM parameters. In contrast, worker failures are less critical because workers do not contain training state.



(a) Example of the distributed setup used to train DLRMs.



(b) Naive erasure-coded DLRM with $k = 3$ and $r = 1$.

Figure 1: Example of (a) normal and (b) (naive) erasure-coded training with parameters e_0, e_1, and e_2, and gradients ∇_0 and ∇_1. For simplicity, only accesses and updates for embedding table entries are shown.

Checkpointing is the main approach used for fault tolerance in DLRM training [16, 29]. This involves periodically pausing training and writing the current parameters and optimizer state to stable storage, such as a distributed file system. If a failure occurs, the entire system resets to the most recent checkpoint and restarts training from that point. While simple, checkpointing frequently pauses training to save DLRM state and has to redo work after failure. Thus, checkpointing has been shown to add significant overhead to training production DLRMs, such as at Facebook [29]. Checkpointing also consumes significant network and storage bandwidth in datacenters and has a large storage footprint [16]. Even more concerning, these overheads increase with DLRM size. Given the trend of increasing model size [43, 48] (described in §2.2), *checkpointing is slated to incur even larger overhead for training future DLRMs.*

An alternative to checkpointing that does not require a lengthy recovery process is to replicate DLRM parameters on separate servers. However, replication requires at least twice as much memory as a checkpointing-based system, which is impractical given the large sizes of DLRMs.

An ideal approach to fault-tolerant DLRM training would (1) operate with low training-time and memory overhead, and (2) recover quickly from failures, while (3) not introducing potential accuracy loss (and the associated uncertainty). Finally, such a solution should scale well with increases in DLRM size so as to support emerging DLRMs. Designing a system that meets these requirements is the goal of this paper.

Erasure codes are coding-theoretic tools that leverage proactive redundancy (like replication) but with significantly less memory overhead, and have been widely employed in storage and communication systems (e.g., [37, 41]). Like replication and traditional checkpointing, erasure coding would not alter training accuracy. Due to their low memory overhead, erasure codes offer the potential for efficient fault tolerance in DLRM training. As shown in Fig. 1b, a DLRM training system could potentially construct "parity parameters" by encoding k parameters from separate servers. In this example, a parity p is formed from parameters e_0, e_1, and e_2 via the encoding function $p = e_0 + e_1 + e_2$, and placed on a separate server. If a server fails, lost parameters can be recovered by reading the k available parameters and performing the erasure code's decoding process (e.g., $e_1 = p - e_0 - e_2$).

While leveraging erasure codes in DLRM training appears promising, this setting comes with challenges due to the interaction between erasure codes and unique aspects of DLRMs. In this work, we thoroughly investigate the use of erasure codes in DLRM training systems and uncover these challenges and solutions to overcome them. The result is ECRec,a DLRM training system that achieves efficient fault tolerance through careful system design based on insights into the unique characteristics of DLRM training. We describe these challenges and how ECRec overcomes them below.

Hybrid redundancy. We show in §3.2 that correctly using erasure codes in DLRM training necessitates more communication than replication. Thus, ECRec must determine which parameters should be erasure coded to straddle a tradeoff between memory and network overhead. ECRec approaches this decision based on unique characteristics of DLRMs: while embedding tables account for the vast majority of the memory use of DLRMs, gradients for NN parameters dominate the network bandwidth used in updates. For example, for the DLRM trained on the Criteo dataset [1] in MLPerf [30], embedding tables account for 99% of the DLRM parameters, but their gradients only take up 35% of the network bandwidth, while NNs account for 1% of all parameters, but 65% of the network bandwidth (similar for other datasets/models; see §3.3).

Based on this asymmetry, ECRec takes a hybrid approach to redundancy by *erasure coding embedding tables and replicating NN parameters.* Erasure coding embedding tables maintains low memory overhead, while replicating NN parameters reduces the network bandwidth consumed without adding much memory overhead.

Maintaining correctness. Redundant parameters in ECRec must be kept synchronized with DLRM parameters to ensure correct recovery. We show in §3 that correctly updating parities when using optimizers that store internal state (e.g., Adagrad, Adam) without incurring large memory overhead is challenging. ECRec circumvents these challenges by delegating the responsibility for updating parities to servers, rather than workers, via an approach we call "difference propagation," and by using two-phase commit.

Pause-free recovery. An erasure code's recovery process can be resource intensive because it involves reading all available data to a single server and performing decoding [40, 42]. This can lead to long recovery times during which training is stalled. ECRec recovers quickly by enabling training to continue during recovery. ECRec leverages on-demand reconstruction of lost DLRM parameters to service new training iterations while full recovery proceeds in the background. Meanwhile, ECRec carefully ensures that new training updates do not conflict with the background recovery process.

We implement ECRec atop XDL [21], an open-source, industrial-scale DLRM training system, and evaluate using variants of the

Criteo DLRM in MLPerf [30]. ECRec recovers faster than checkpointing and with lower training-time overhead for large DLRMs. For example, ECRec recovers from failure up to 9.8× faster than the average case for checkpointing and enables training to continue during recovery with only a 7–13% drop in throughput, while checkpointing pauses training during recovery. This is critical for ensuring that DLRM training completes within the tight deadlines needed for frequent production deployment, even when failures occur. ECRec reduces training-time overhead for a large DLRM by up to 66% compared to checkpointing. ECRec's benefits increase with DLRM size, showing its promise to enable efficient fault tolerance for current and future DLRMs.

The primary contributions of this work are:

- Investigating how fault-tolerance mechanisms interact with the unique characteristics of DLRMs.
- Designing ECRec, a fault-tolerant DLRM training system that judiciously uses erasure codes and replication, enables training to continue during recovery and keeps the consistency guarantees of the underlying system.
- Evaluating ECRec in a variety of settings to show its superior recovery performance compared to checkpointing and its ability to gracefully scale training-time overhead for future DLRMs, which are expected to increase in size
- To the best of our knowledge, ECRec is the first DLRM system that employs erasure codes to provide fault tolerance.

2 CHALLENGES IN FAULT-TOLERANT DLRMS

2.1 DLRM training systems

Recall from §1 that DLRMs are large in size due to their use of embedding tables and that DLRM training is typically distributed across a set of servers and workers (Fig. 1a). Model parameters (embedding tables and NNs) are sharded across server memory. In a training iteration over a batch of data, a worker reads embedding table entries relevant to that batch and all NN parameters, performs a forward and backward pass to generate gradients (for NN parameters and embedding table entries), and sends gradients back to the servers hosting the parameters that were read. An optimizer (e.g., Adam) on each server uses gradients to update parameters. Many systems use asynchronous training when training DLRMs (e.g., Facebook [6], Alibaba [21]). We thus focus on asynchronous training, but our work could extend to synchronous training.

Stateful optimizers. Many popular optimizers use per-parameter state in updating parameters (e.g., Adam [12], Adagrad [13], momentum SGD). We refer to such optimizers as "stateful." For example, Adagrad tracks the sum of squared gradients for each parameter over time and uses this when updating the parameter. Optimizer state is kept in memory on servers and is updated when the corresponding parameter is updated. As this state grows with DLRM size, it can consume a large amount of memory.

2.2 Unique characteristics of DLRMs

DLRMs contain unique characteristics compared to other deep networks. First, while many deep networks today leverage large NNs, DLRMs typically leverage small NNs but large embedding tables [20]. For example, a DLRM commonly used to train on the Criteo dataset [1] contains over 100 GB of embedding tables, but less than 1 GB of NN parameters. Second, components of DLRMs have diverse access patterns. Each training sample typically accesses only a few embedding table entries, but all NN parameters. For example, the average number of embedding table entries accessed by a batch of 2048 training samples is only 8900 on the Criteo dataset. This is a small fraction of the roughly 200 million entries in the DLRM. Thus, embedding table entries are updated sparsely, while all NN parameters are updated on every training batch.

Scaling trends. Similar to other deep models, increasing parameter count in DLRMs has led to increased accuracy. Thus, DLRMs have drastically increased in size over the years: whereas in 2020, Facebook used DLRMs with 100s of billions of parameters, today's DLRMs now use well over one trillion parameters [33]. This scaling is heavily driven by the increased number of embedding table entries in DLRMs: Facebook reports that the number of embedding table entries in DLRMs increased by 17.5× from 2017–2021 [43]. This scaling trend is expected to increase in the future [48].

2.3 Checkpointing and its downsides

Recall from §1 that maintaining fault tolerance for servers is critical for large-scale DLRM training, and that fault tolerance for workers is not as much of a concern. Recall also from §1 that checkpointing is the primary approach used for fault tolerance in DLRM training.

Recently, *Facebook reported that overheads from checkpointing account for, on average, 12% of DLRM training time*, and that these overheads add up to *over 1000 machine-years of computation* [29]. Checkpointing has two primary time penalties:

1. Time penalty during normal operation. Writing checkpoints to stable storage is a slow process given the large sizes of DLRMs, and training is paused during this time so that the saved model is consistent. Intuitively, the overhead of checkpointing under normal operation increases with the frequency of checkpointing and the size of the DLRM). This is illustrated empirically in §5.2.

2. Time penalty during recovery. Upon failure, a system using checkpointing must roll back the DLRM to the state of the most recent checkpoint by reading it from storage and redo all training iterations that occurred between this checkpoint and the failure. New training iterations are paused during this time. The time needed to read checkpoints from storage can be significant [29] and grows with DLRM size. The expected time to redo iterations grows with the time between checkpoints: if checkpoints are written every T time units, this time will be 0 at best (failing just after writing a checkpoint), T at worst (failing just before writing a checkpoint), and $\frac{T}{2}$ on average. We show this recovery performance in §5.3.

Takeaway. Checkpointing suffers a fundamental tradeoff between time overhead in normal operation and that in recovery. Increasing the time between checkpoints reduces the fraction of time paused when saving checkpoints, but increases the expected work to be redone in recovery. Low training-time overhead in *both* normal mode and during recovery is needed to meet the tight deadlines for deploying DLRMs in production applications [33] even when failures occur. Facebook has also noted that reducing the storage and network bandwidth used in checkpointing DLRMs is critical for reducing load on these shared resources [16]. These overheads increase with model size. Given the trends of increasing

model size noted in §2.2 *checkpointing is slated to become an even larger overhead in training future DLRMs.*

This calls for alternatives for fault tolerance in DLRM training that scale to large DLRMs without a severe tradeoff between training-time overhead and recovery performance.

2.4 Reducing the overhead of checkpointing

There are multiple classes of techniques for reducing the overhead of checkpointing. We next discuss these at a high level, as well as the challenges of applying them to DLRM training.

Approximation. Several recent approaches aim to reduce the overheads of checkpointing by taking approximate checkpoints or via approximate recovery [9, 16, 29, 38]. However, upon failure, such techniques roll back an approximation of the true DLRM, which can alter convergence and final accuracy. Given the major business value generated by DLRMs, prior work has noted that even small drops in accuracy must be avoided [52]. Furthermore, our personal conversations with multiple practitioners working on large-scale DLRM training indicate that this potential accuracy drop introduces a source of uncertainty that makes debugging production DLRMs difficult. Hence, ideally, one would reduce the overhead of checkpointing without compromising accuracy.

Asynchronous checkpointing. Another way to reduce the overhead of checkpointing is by taking asynchronous checkpoint [5]. Here, training continues while the model state is written to stable storage. However, this results in the model state stored in checkpoints being inconsistent since some parameters in the checkpoint will reflect updates from more recent iterations than others. This can lead to accuracy degradation in the recovered model.

Logging. One may also question whether the overhead of checkpointing could be reduced by logging updates to stable storage as they are generated. This is feasible only if writing to storage can keep pace with the rate at which gradients are generated. As DLRM training systems have many workers operating asynchronously, gradients are generated at a high rate that storage cannot keep pace with. In fact, if storage could keep pace, then DLRM parameters could be kept in storage, rather than in memory. Thus, logging to stable storage is not viable for DLRM training.

2.5 In-memory redundancy

An alternative to checkpointing is to provision extra memory in the system to *redundantly* store DLRM parameters and optimizer state in memory in a fault-tolerant manner.

Replication. As described in §1, one approach to redundantly keeping parameters in memory is to replicate each parameter on separate servers. However, this requires at least twice as much memory as a non-replicated system, which is impractical given the large and growing sizes of DLRMs.

Erasure codes. Erasure codes are coding-theoretic tools that enable redundancy with significantly less overhead than replication [37, 41, 45]. An erasure code encodes k data units to generate r redundant "parity units" such that any k out of the total $(k + r)$ data and parity units suffice for a decoder to recover the original k data units. Erasure codes operate with overhead of $\frac{k+r}{k}$, which is less than that of replication by setting $r < k$. These properties have led to wide adoption of erasure codes in many domains (e.g., [23, 37, 39, 41, 45, 47, 49]).

For example, consider the naive erasure-coded DLRM training system in Fig. 1b in which parameters e_0, e_1, and e_2 are stored on three separate servers. Suppose the system must tolerate one server failure. An erasure code with parameters $k = 3$ and $r = 1$ could do so by encoding a parity unit as $p = e_0 + e_1 + e_2$ and storing this parity unit on a fourth server. Suppose the server holding e_1 fails. The system could recover e_1 using the erasure code's subtraction decoder: $e_1 = p - e_0 - e_2$. This setup can recover from any one of the servers failing by using only 33% more memory, while replication would require 100% more memory.

Takeaway. An ideal approach to fault-tolerant DLRM training would (1) avoid pauses during both normal operation and recovery, (2) have low memory overhead, (3) introduce no potential for accuracy loss, and (4) scale to large DLRMs. Erasure codes offer promising potential for achieving these goals. However, there are several challenges in using erasure codes for DLRM training. We describe these and how they can be overcome in the next section.

3 ECREC: ERASURE-CODED DLRM TRAINING

We propose ECRec, a fault-tolerant DLRM training system that requires no pausing during training nor rolling back during recovery and provides the same accuracy guarantees as the underlying training system. ECRec is designed based on investigating the interplay between erasure codes and unique properties of DLRM training.

3.1 Overview of ECRec

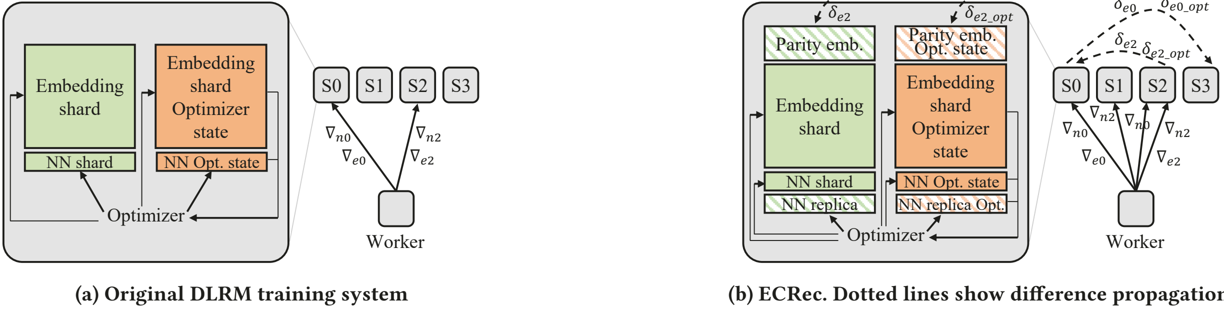
Fig. 2 shows a toy example of the high-level operation of traditional DLRM training systems and that of ECRec, as well as the detailed contents of an individual server in each system. Arrows show the flow of data when performing an update from a single worker.

In the original system (Fig. 2a), a worker sends gradients for NN parameters (∇_{n0}, ∇_{n2}) and for embedding table entries (∇_{e0}, ∇_{e2}) to the servers hosting these parameters. As shown in the inset, the optimizer on a server reads optimizer state for the corresponding entries and NN parameters and uses this with the received gradients to compute updates, which are applied to relevant parameters.

Hybrid redundancy. Fig. 2b shows that ECRec maintains redundant versions (hashed boxes) of embedding tables, NN parameters, and optimizer state. ECRec selects the type of redundancy to use for each type of parameter based on its size in memory and the network bandwidth it uses for updates.

To minimize memory overhead, all parameters in a DLRM would ideally be erasure coded. However, as we show in §3.3, correctly updating parities during training requires more communication than updating a replica. Thus, erasure coding parameters that have a high network-bandwidth footprint may add considerable overhead.

ECRec balances this tradeoff by observing the asymmetry between memory and network-bandwidth footprints for DLRM parameters: embedding tables (and their optimizer state) account for the majority of the memory in DLRMs, while NN parameters (and their optimizer state) account for a minor fraction [20]. On the other hand, gradients for NN parameters account for the majority of network traffic during updates. We discuss this in detail in §3.3.



(a) Original DLRM training system

(b) ECRec. Dotted lines show difference propagation.

Figure 2: Example of an update from one worker with 4 servers (a) in a traditional DLRM training system and (b) in ECRec with $k = 3$ and $r = 1$ (including replication of NN parameters). A detailed view of a single server is shown in the insets to the left.

Based on this asymmetry and the additional network traffic needed for updating erasure-coded parameters, ECRec *erasure codes embedding tables and their optimizer state, and replicates NN parameter and their optimizer state.* Doing so enables ECRec to operate with low memory overhead, as the vast majority of the DLRM's memory footprint is erasure coded, while reducing network bandwidth overhead. We describe specifically how ECRec overcomes challenges in using erasure codes for embedding tables in §3.2, and why ECRec leverages replication for NNs in §3.3.

Updating redundant parameters. Fig. 2b also shows how ECRec keeps redundant parameters up-to-date. Workers send gradients for NN parameters (∇_{n0}, ∇_{n2}) to each server hosting a replica of a NN parameter. However, as will be described in §3.2, this same process is insufficient for correctly updating parities of embedding table entries and their optimizer state. To overcome this issue, ECRec leverages "difference propagation" in §3.2 (dashed lines in Fig. 2b), in which a server hosting an embedding table entry forwards the differences resulting from an update for that entry and its optimizer state to the server holding the corresponding parity.

Recovery and consistency. Finally, ECRec enables training to continue during recovery from a server failure, and maintains the same consistency guarantees as the underlying DLRM training system. Low-overhead recovery is critical for meeting the tight deadlines for deploying DLRMs in production even when failures occur. We describe how ECRec enables each of these features in §3.4 and §3.5, respectively.

We next describe each of ECRec's components in detail.

3.2 Erasure-coded embedding table entries

As described in §2.1, embedding tables and optimizer state are sharded across servers. ECRec encodes k embedding table entries from different shards to produce a "parity entry," and places the parity entry on a separate server. Optimizer state is similarly encoded to form "parity optimizer state," and placed on the same server as the corresponding parity entry.

Parities in ECRec are updated whenever any of the k corresponding embedding table entries are updated. Hence, parities are updated more frequently than the original entries, and must be placed carefully so as not to introduce load imbalance among servers. ECRec uses rotating parity placement to distribute parities among servers, resulting in an equal number of parities per server. An example of

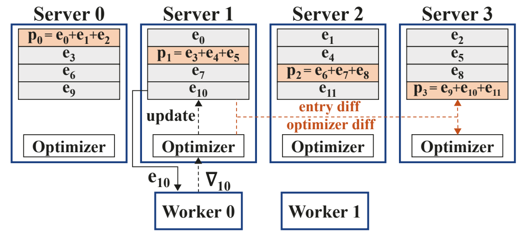


Figure 3: Example of rotating parity placement and difference propagation with $k = 3$, $r = 1$.

this is shown in Fig. 3 with $k = 3$: each server is chosen to host a parity in a rotating fashion, and the entries used to encode the parity are hosted on the three other servers. This approach is inspired by parity placement in RAID-5 systems [37].

Encoder and decoder. We focus on using erasure codes with parameter $r = 1$ (i.e., one parity per k entries, and recovering from a single failure) since it represents the most common failure scenario in datacenters [40]. However, we describe in §4 how ECRec can tolerate multiple failures. Within the setting of $r = 1$, ECRec uses the simple summation encoder shown in Fig. 3, and the corresponding subtraction decoder: with $k = 3$, embedding table entries e_0, e_1, and e_2 are encoded to form parity $p = e_0 + e_1 + e_2$. If the server holding e_1 fails, e_1 will be recovered as $e_1 = p - e_0 - e_2$.

3.2.1 Correctly updating parities. The naive approach to erasure-coded DLRM training in Fig. 1b cannot *correctly* update parity entries when using a stateful optimizer (e.g., Adam, Adagrad).

Consider the system in Fig. 1b with the Adagrad [13] optimizer. Let $e_{i,t}$ denote the value of embedding table entry e_i after t updates, and $\nabla_{i,t}$ denote the gradient for $e_{i,t}$. The update performed by Adagrad for $e_{0,t}$ with gradient $\nabla_{0,t}$ is:

$$e_{0,t+1} = e_{0,t} - \frac{\alpha}{\sqrt{G_{0,t} + \epsilon}} \nabla_{0,t} \tag{1}$$

where $G_{0,t} = \nabla_{0,0}^2 + \nabla_{0,1}^2 + \ldots + \nabla_{0,t}^2$, α is a constant learning rate, and ϵ is a small constant. $G_{0,t}$, which we call e_0's "accumulator," is an example of per-parameter optimizer state.

As described in §3.1, ECRec maintains one "parity optimizer parameter" for every k original optimizer parameters. In the example

above, a "parity accumulator" would be $G_p = G_0 + G_1 + G_2$. This is easily kept up-to-date by adding to G_p the squared gradients for updates to each of the k original entries (e.g., $G_{p,t+1} = G_{p,t} + \nabla^2_{0,t+1}$ after an update to e_0 that produces $\nabla_{0,t+1}$). However, using this parity accumulator to update the parity entry based on $\nabla_{0,t}$ (i.e., replacing e_0 with e_p and G_0 with G_p in Eqn. 1) would result in an incorrect parity entry, since $G_{0,t} \neq G_{p,t}$.

This issue arises for any stateful optimizer, such as the popular Adagrad, Adam, and momentum SGD. Therefore, ECRec must employ some means of maintaining correct parities when using stateful optimizers. This could be overcome by replicating the k original optimizer parameters on the server hosting the parity. However, optimizer state for embedding tables is large and grows with embedding tables, making such replication impractical.

Difference propagation. The challenge described above stems from sending gradients directly to the servers hosting parities: servers holding parities receive only the gradient for the original entry and must correctly update the parity entry and optimizer state. At the same time, an alternative of sending embedding table entries from the original server hosting them to the server hosting the corresponding parity entry would not enable one to correctly update the parity entry because the server hosting the parity entry would need access to the previous version of the embedding table entry to generate a gradient. To overcome this challenge, ECRec leverages *difference propagation.* Under difference propagation, workers send gradients only to the servers holding embedding table entries for that gradient. After applying the optimizer to entries and updating optimizer state, the server sends the *differences* in entry and optimizer state to the server holding the corresponding parity entry. The receiving server adds these differences to the parity entry and optimizer state. This is shown in Fig. 2b, with the worker sending gradients for entries (∇_{e0}, ∇_{e2}) to Servers 0 and 2, which then send differences in entries (δ_{e0}, δ_{e2}) and optimizer state (δ_{e0_opt}, δ_{e2_opt}) to servers hosting the corresponding parity. By sending differences to servers, rather than sending gradients, difference propagation updates parity entries correctly when using stateful optimizers.

3.3 Replicated neural network parameters

We next describe how ECRec applies fault tolerance to neural network parameters and their optimizer state.

Why additional fault tolerance is needed for NNs. Recall from §2.1 that workers in DLRM training pull all NN parameters from servers on each training iteration. Thus, each worker contains an approximate replica of the current NN parameters.[1] This may lead one to question whether ECRec can simply leverage the NN parameters pulled by workers as "natural" replicas of the NN sharded across servers. While this approach could recover NN parameters, it does not provide fault tolerance for optimizer state used for NN parameters. Because optimizer state is kept on servers and is not read by workers, optimizer state for NNs is lost if a server fails. Thus, an alternative that keeps both NN parameters and their optimizer state redundant is needed.

Erasure coding NN parameters results in high overhead. One could keep NN parameters and their optimizer state fault tolerant via erasure coding in a similar fashion to that in §3.2. After all, these parameters are sharded across servers just like embedding tables, so the same technique could be applied.

However, we find that using erasure coding as described in §3.2 for NN parameters leads to significant performance overhead. Recall from §2.2 that, while embedding tables are updated sparsely, all NN parameters and their optimizer state are updated on every training iteration. This leads to an imbalance in the amount of network bandwidth consumed for updating embedding table entries and NN parameters, with NN parameters consuming significantly more bandwidth. For example, for the DLRM used for the Criteo dataset [1], we find that the *network bandwidth consumed in a given training iteration for updating NN parameters is over 1.8× higher than that for embedding table entries.* Similarly, for two variants [7, 8] of a DLRM trained on the Avazu dataset [3], we find that NNs consume 80–90% of the network bandwidth for updating parameters.

Erasure coding with difference propagation requires 200% network bandwidth overhead for a given update, as differences for both the original parameter and its optimizer state (each of which are the same size as the gradient) must be forwarded to the server hosting the parity. Given the dominance of NN parameters on network bandwidth, performing difference propagation for NNs adds considerable training-time overhead. However, as described in §3.2, difference propagation is necessary for correctly updating parities.

Replicating NN parameters. ECRec exploits the asymmetry between the network bandwidth consumed by gradients for NN parameters and the size of NN parameters. While gradients for NNs account for the majority of network bandwidth during updates, NN parameters represent a minor portion of the overall DLRM size: for the DLRMs used for Criteo [1] and Avazu [3], NN parameters and their optimizer state account for less than 1% of the overall DLRM size. Thus, NN parameters and their optimizer state can be replicated with little memory overhead added to the overall system.

Replicating NN parameters and their optimizer state allows ECRec to avoid difference propagation (and its associated network-bandwidth overhead) for updating NNs. In ECRec, gradients for a given NN parameter are sent from workers to both servers hosting replicas of the given parameter. Each server with a replica locally updates the parameter and its replica of the optimizer state. In this way, ECRec incurs half of the network bandwidth overhead for NN parameters as that incurred by erasure coding NN parameters with difference propagation: whereas difference propagation additionally sends both the difference for the NN parameter and the difference for its optimizer state, replication sends *only* the gradient for the NN parameters an additional time.

3.4 Pause-free recovery from failure

We now detail how ECRec recovers without pausing training.

Due to the property of erasure codes used in ECRec that any k out of the $(k+1)$ original and parity units suffice to recover the original k units, ECRec can continue training even if one server fails. For example, a worker in ECRec could access embedding table entry e_1 in Fig. 3 even if Server 2 fails by reading e_0, e_2, and p, and decoding $e_1 = p - e_0 - e_2$. Such read operations that require decoding are referred to as "degraded reads" in erasure-coded systems. Similarly, NN parameters can be accessed via replicas on another server.

[1]This replica is only approximate because workers operate asynchronously. Thus, the NN that one worker has may not reflect the latest updates from other workers.

Challenges in erasure-coded recovery. Despite the ability to perform degraded reads, ECRec must still fully recover failed servers to tolerate future failures. This is simple and efficient for NN parameters, as they are copied from a replica server and are small in size. However, prior work on erasure-coded storage has shown that full recovery can be time-intensive [40, 42]. Full recovery in ECRec requires decoding all embedding table entries and optimizer state held by the failed server. This consumes significant network bandwidth in transferring entries for decoding, and server CPU in performing decoding. Given the large sizes of embedding tables and their optimizer state, fully recovering before resuming training can significantly pause training, making it challenging to meet production training deadlines for deployment when failures occur.

Training during recovery via granular locking. Rather than solely performing degraded reads after a failure or pausing until full recovery is complete, ECRec *enables training to continue while full recovery takes place.* Upon failure, ECRec begins full recovery of lost embedding table entries and optimizer state. In the meantime, the system continues to perform new training iterations, with workers performing degraded reads to access entries from the failed server.

ECRec must avoid updating an entry while the entry is used for recovery. If the recovery process reads the new value of the entry, but the old value of the parity (e.g., because the update has not yet reached the parity), then the recovered entry will be incorrect. ECRec uses granular locking to avoid this. The recovery process locks a set of the lost entries that it will decode. While this lock is held, updates to entries that will be used in recovery for the locked entries are buffered in memory on servers. Workers access updated, but locked entries via the buffer. When a lock is released, buffered updates are applied to the embedding tables, and the next set of entries is locked. The number of entries covered by each lock introduces a tradeoff between time spent switching locks and server memory overhead for buffering, which can be navigated based on deployment-level requirements.

3.5 Recovering a consistent DLRM

As discussed in §2.4, ECRec aims not to introduce new sources of accuracy loss to the DLRM training system. We next describe how ECRec maintains the consistency guarantees of the asynchronous DLRM training system it builds upon.

Under asynchronous training, concurrent updates to parameters from different workers can occur in an arbitrary order and can potentially overwrite one another. The same can occur with the redundant parameters employed by ECRec, matching the consistency guarantees of the original system.

However, one case requires care: server failure while an update is in flight. To better illustrate this, first note that each training iteration updates parameters on different servers. Suppose one server holding an embedding table entry used in the current iteration fails during the update step. The erasure code's decoding function would correctly recover if the corresponding parity entry was updated through difference propagation before the failure occurred. However, if the server failed before propagating its difference, the recovered DLRM would be inconsistent: other parameters would be recovered to the state including the iteration's update, while the recovered entry in question would not reflect the update.

The issue underlying this example is a lack of knowledge of whether a parity entry has been updated through difference propagation before a failure occurs.

Two-phase commit. ECRec uses two-phase commit (2PC) when updating parameters to avoid this inconsistency. The 2PC protocol is coordinated by individual workers on each training iteration they perform. 2PC in ECRec splits the process of updating a set of parameters on separate servers into two phases. In the first phase, updates for parameters (original and redundant) are computed and staged. In the second phase, staged updates are applied to parameters (original and redundant). This ensures that DLRM parameters are in a consistent state before recovery begins.

Example of 2PC. We next walk through a simple example of how it would be applied in ECRec with $k = 2$ and $r = 1$ in Fig. 4.

Fig. 4a shows the first phase of the protocol. (1) The worker sends gradients to servers hosting relevant parameters for a given update. (2) Receiving servers compute the update using the optimizer and stage the updated parameters temporarily, rather than applying the update directly. (3) Receiving servers perform difference propagation, which results in staged versions of the corresponding p parameters. (4) Servers that received updates via difference propagation send an acknowledgment back to the server that sent the difference to acknowledge that the update to the p has been staged. (5) Servers that received the original gradient in step 1 send an acknowledgement back to the worker. Once the worker has received acknowledgments from all servers to which it sent gradients, the first phase is complete and it is safe to begin the second phase.

Fig. 4b shows the second phase of the protocol. (1) The worker coordinating the protocol sends a "commit" message to all servers that staged updates in the previous phase. (2) Receiving servers apply the staged update to the corresponding parameter. (3) Receiving servers send an acknowledgment to the coordinating worker once they have applied the staged update.

Handling failures in 2PC. If a failure happens in the first phase of 2PC, the coordinating worker aborts and restarts the protocol, as is standard. If a failure happens in the second phase, the protocol continues, but with the coordinating worker waiting for acknowledgments only from the non-failed servers involved in the protocol. This is safe because the update for the iteration will have been committed to enough parity/replica and original parameters to enable the recovery process to reconstruct parameters on the failed server.

Tradeoffs. 2PC in ECRec adds an extra round of communication for updating parameters. As will be shown in §5, this adds training-time overhead. Given that ECRec uses 2PC solely to protect against losing portions of an update when a failure occurs, a reader may question how important it is to preserve these full updates. After all, the training system on top of which ECRec is built is asynchronous, which means that concurrent updates from workers can overwrite one another. Nevertheless, we have implemented 2PC in ECRec to adhere to the design goal in §2.4 of introducing no additional sources of inaccuracy in training. While potentially a heavy-handed solution, 2PC ensures that the fault-tolerance technique used in training does not open additional sources of inconsistency (and the related uncertainty when debugging accuracy). We show in §5.2 that the overhead of ECRec can be reduced if one is willing to forgo these guarantees by turning off 2PC.

(a) Phase 1 of 2PC.

(b) Phase 2 of 2PC.

Figure 4: Toy example of the 2PC protocol used in ECRec with $k = 2$ and $r = 1$.

4 DISCUSSION

Tradeoffs in ECRec. ECRec encodes k embedding table entries into a single parity entry (for $r = 1$)). Parameter k results in the following tradeoffs in ECRec:

Increasing k decreases memory overhead and fault tolerance. As ECRec encodes one parity entry for every k entries, less memory is required with increased k. However, since the erasure codes used by ECRec can recover from any one out of $(k + 1)$ failures, increasing k decreases the fraction of failed servers ECRec can tolerate.

Increasing k does not change load during normal operation. As each embedding table entry in ECRec is encoded to produce one parity entry, each update applied to an entry is also be applied to one parity. Thus, the total increase in load in terms of the number of updates performed is 2×, regardless of k. We also show in §5.2 that ECRec balances the overall load for updates across servers.

Increasing k increases the time to fully recover. Recovering embedding tables in ECRec involves reading k entries from separate servers and decoding. Thus, the network traffic used in recovery increases with k, which increases the time to fully recover. However, as described in §3.4, ECRec continues training during this time.

Tolerating multiple failures Recall from §3 that ECRec uses erasure codes with $r = 1$, that is, which can recover from a single server failure. This choice was informed by prior studies of cluster failures that showed that single-node failures are the most common failure scenarios among groups of nodes [40].

ECRec can be easily adapted to tolerate additional faults by using erasure codes with parameter $r > 1$. An alternative to this that still leverages $r = 1$ is to partition the overall cluster used in training into smaller groups of servers over which erasure coding with $r = 1$ is performed, such that more than a single failure within each group is unlikely. Finally, ECRec could also be adapted to take checkpoints at a much lower frequency than normal checkpointing schemes as a second layer of defense against concurrent failures. This final approach bears similarity to multi-level checkpointing [32].

Leveraging fine-grained access frequency Prior works have shown that embedding table entries are accessed with varying frequency during training, with a small number of entries accounting for the vast majority of overall access (e.g., [7]). One could consider adapting ECRec to consider the frequency of access of individual embedding table entries. For example, it could potentially be beneficial to replicate a fraction of the most-frequently-accessed embedding table entries to reduce the network bandwidth overhead of performing difference propagation for these entries. Leveraging such finer-grained access frequency metrics is a promising direction for future work, but outside the scope of the present work due to the additional tradeoffs and system complexity it would introduce.

5 EVALUATION

We next evaluate ECRec. The highlights are as follows:

- ECRec recovers from failure up to 9.8× faster than the average time for checkpointing. Fast recovery is critical for meeting the tight deadlines of production applications that frequently retrain and deploy DLRMs [33].
- ECRec enables training to proceed during recovery with only a 7%–13% drop in throughput, whereas checkpointing requires training to completely pause.
- ECRec reduces training-time overhead on large DLRMs by up to 66% compared to checkpointing. ECRec scales well with DLRM size, showing promise for training current and future DLRMs.
- While ECRec introduces additional load for updating parities, the impact of increased load on training throughput is alleviated by improved cluster load balance.

5.1 Evaluation setup

We implement ECRec in C++ on XDL, an open-source DLRM training system from Alibaba [21].

Datasets and models. We evaluate primarily with the Criteo Terabyte dataset [1], which is commonly used for evaluating DLRM training systems. We randomly draw one day of samples from the dataset by picking each sample with probability $\frac{1}{24}$ in one pass through the dataset, and use this subset in evaluation to reduce storage requirements. This random sampling results in a sampled dataset that mimics the full dataset.

We focus on the Criteo Terabyte dataset because it is one of the most-commonly used public datasets for DLRMs and because its large size more closely emulates production-scale datasets. While other public datasets are available, many leverage small embedding tables (e.g., less than 1 GB), making them impractical for large-scale experimentation. Furthermore, the Criteo Terabyte dataset has similar characteristics to many other public datasets, as shown by Adnan et al. [7]. Our analysis of the memory and network use of DLRM training in §3.3 also reflected this similarity.

We evaluate on various DLRMs based on the DLRM for the Criteo dataset from MLPerf [35], which has 13 embedding tables, for a total of around 200M entries each with 128 dense features. We use SGD

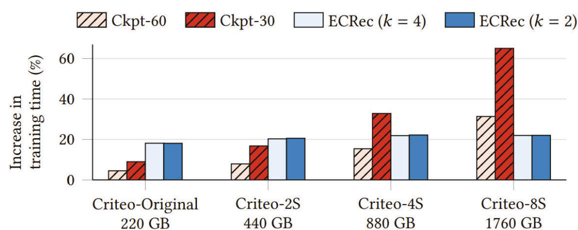


Figure 5: Training-time overhead.

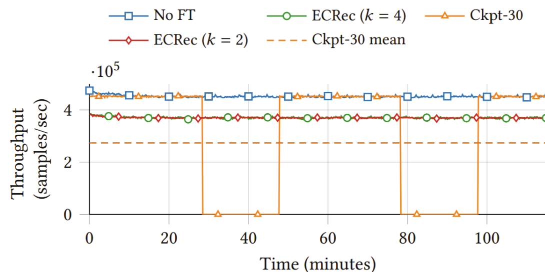


Figure 6: Throughput of training Criteo-8S vs. Ckpt-30.

with momentum as the optimizer, which adds one floating point value of optimizer state per parameter. Any other optimizer can also be used. The total size of the embedding tables and optimizer state is 220 GB. The DLRM uses a seven-layer multilayer perceptron with 128–1024 features per layer as a NN [4].

We evaluate on DLRMs of different size. First, we increase the number of embedding table entries in the DLRM (i.e., sparse dimension). This increases the memory required per server and the amount of data that must be checkpointed/kept redundant and recovered. We consider variants of the original Criteo DLRM described above, with one- , two- , four- , and eight-times more entries. We refer to each of these as *Criteo-Original, Criteo-2S, Criteo-4S*, and *Criteo-8S*, which have size 220, 440, 880, and 1760 GB, respectively. *We primarily focus on scaling the sparse dimension of embedding tables, as this reflects a prominent scaling trend observed today:* Facebook reports that from 2017 to 2021, the number of embedding table entries in DLRMs has increased by 17.5× [43].

For completeness, we also evaluate on a DLRM in which we increase the size of embedding table *entries* (i.e., dense dimension). This increases the memory required per server, the amount of data that must be checkpointed/kept redundant, the network bandwidth in transferring entries/gradients, and the work done by workers and servers. Thus, this form of scaling complements model scaling in the sparse dimension. We consider a variant of the original Criteo-2S DLRM described above but in which each entry is twice as large. The width of the input layer of the NN is also increased to accommodate the larger entry size. We refer to this DLRM as *Criteo-2S-2D*, and it has a total size of 880 GB.

Coding parameters and baselines. We evaluate ECRec with k of 2 and 4, which have 50%, 25% memory overhead, respectively. Though we focus on $r = 1$, these experiments also provide insight into the performance of the technique described in §4 in which one can tolerate multiple failures by partitioning a set of servers into groups in which erasure coding with $r = 1$ is performed within each group. We use one lock during recovery by default (see §3.4), but also evaluate finer locking granularity.

We compare ECRec to checkpointing to HDFS (1) every 30 minutes (Ckpt-30) and (2) every 60 minutes (Ckpt-60). We also compare to running (3) without any checkpointing or fault tolerance at all (No FT). As recovery time for checkpointing depends on when a failure occurs (see §2.3), we also compare against the best-, average-, and worst-case scenarios for checkpointing when evaluating recovery. Checkpointing to HDFS is representative of production DLRM training, which often uses HDFS-like distributed file systems [6, 16]. Furthermore, the checkpointing baselines we use have competitive performance: we find that checkpointing via HDFS is only 7%–27% slower than a (purposely unrealistic) baseline of writing directly to a local SSD. In addition, for Criteo-Original, which is representative of current DLRMs, the checkpoint-writing overhead we report is similar to that reported in production by Facebook [29].

We also compare with approaches that use approximation in checkpointing (and thus, which can result in accuracy loss), in §5.4. Existing works on approximate checkpointing for training do not have open-source code releases (e.g., [9, 16, 29, 38]), so we model their performance analytically and compare it to real system performance of ECRec. Because analytical modeling does not include any system overheads from these approximation-based baselines, this evaluation favors the approximation-based baselines.

We do not compare against replicating embedding tables because, as described in §1, the 2× memory overhead of replicating large embedding tables makes it a non-starter in large-scale settings. In cases in which one is willing to pay the 2× memory overhead of replication, replication is naturally expected to exceed the performance of both ECRec and checkpointing-based approaches.

Cluster setup. We evaluate on AWS with 5 servers of type r5n.8xlarge, each with 32 vCPUs, 256 GB of memory, and 25 Gbps network bandwidth (due to memory requirements, r5n.12xlarge and r5n.24xlarge are used for DLRMs larger than 440 GB and 880 GB). We use 15 workers of type p3.2xlarge, each with a V100 GPU, 8 vCPUs, and 10 Gbps of network bandwidth. While we are unable to scale cluster size even further due to cost, we have chosen the ratio of worker to server nodes inspired by real-world deployments (e.g., XDL [21]), thus capturing the relevant tradeoffs. Workers use batch size of 2048. For checkpointing, we use 15 additional HDFS nodes of type i3en.xlarge, each with NVMe SSDs and 25 Gbps of network bandwidth. All instances use AWS ENA networking.

Metrics. For performance during normal operation, we measure training throughput (samples/second) and training-time overhead, which is the percent increase in the time to train a certain number of samples. For recovery, we measure the time to fully recover a failed server and training throughput during recovery.

5.2 Performance during normal operation

Fig. 5 shows the training-time overhead of ECRec and checkpointing compared to a system with no fault tolerance (and thus no overhead) in a two-hour training run. As DLRM size increases, ECRec's training-time overhead grows very slightly, while that of checkpointing increases significantly. For example, going from Criteo-Original to Criteo-8S, ECRec's training-time overhead with

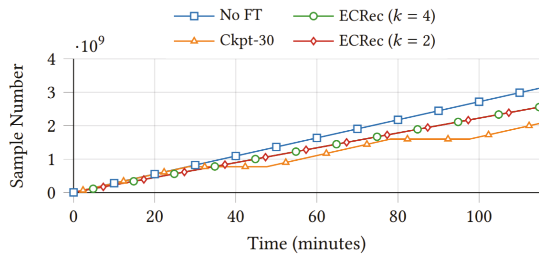


Figure 7: Progress of training Criteo-8S vs. Ckpt-30.

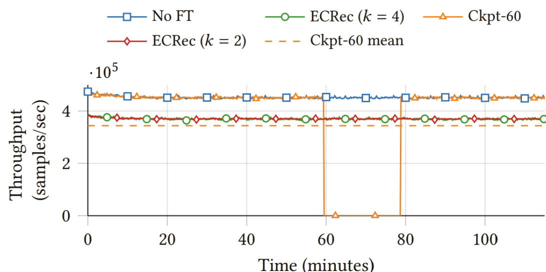


Figure 8: Throughput of training Criteo-8S vs. Ckpt-60.

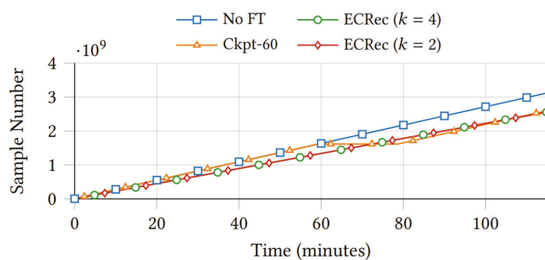


Figure 9: Progress of training Criteo-8S vs. Ckpt-60.

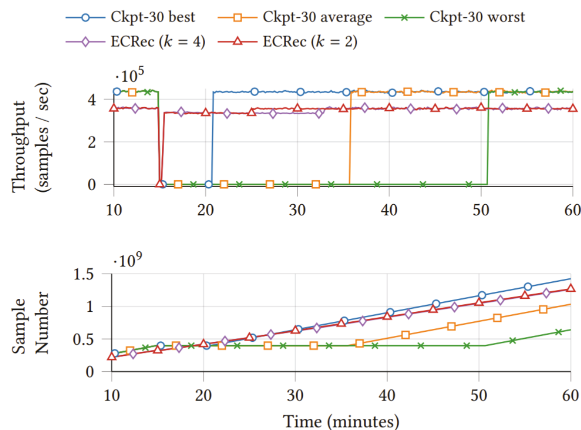


Figure 10: Training throughput (top) and progress (bottom) after a failure at 15 min. on Criteo-4S, compared to Ckpt-30.

k = 4 increases by only 1.2×, while those of Ckpt-30 and Ckpt-60 increase by 7.2× and 7×, respectively. This leads to ECRec significantly reducing training-time overhead for large DLRMs: for Criteo-8S, ECRec has training-time overhead with k = 4 of 22%, while Ckpt-30 and Ckpt-60 have overheads of 65% and 31%, respectively. While one could checkpoint less frequently for such large DLRMs, doing so comes with the adverse effect of prolonged recovery times (as will be shown in §5.3).

ECRec does have higher training-time overhead than checkpointing for smaller DLRMs. However, it is important to note that DLRMs are expected to increase in size [43] (see §2.2). Furthermore, as will be shown in §5.3, ECRec significantly improves performance during recovery compared to checkpointing. Thus, ECRec is poised to remain a scalable solution for future DLRMs, without requiring one to severely trade normal-mode and recovery performance.

Fig. 6 shows that ECRec has slightly lower throughput than No FT, while the throughput of Ckpt-30 fluctuates from that of No FT, to 0 when checkpointing. This makes the mean throughput of Ckpt-30 (dashed line) lower than that of ECRec. This is further shown in Fig. 7: Ckpt-30 progresses slower than ECRec. Figs. 8 and 9 show that the throughput of Ckpt-60 is closer to that of ECRec during normal operation as compared to Ckpt-30 due to the reduced checkpointing frequency. However, this comes at the expense of prolonged recovery times, as will be shown in §5.3.

Effect of parameter k. ECRec has constant network bandwidth and CPU overhead during normal operation regardless of the value of parameter k (see §4). This is shown in Figs. 5, 6, and 7: ECRec has equal performance with k of 2 and 4.

Effect of ECRec on load imbalance. We next evaluate the effect of parity placement in ECRec on cluster load imbalance. We measure load by counting the number of updates that occur on each server when training Criteo-Original.

Without erasure coding, the most-loaded server performs 2.28× more updates than the least-loaded server. In contrast, in ECRec with k = 2 and k = 4, this difference in load is 1.64× and 1.58×, respectively. Thus, the increased load introduced by ECRec is alleviated by *improved load balance.* In ECRec, parities corresponding to the embedding table entries of a given server are distributed among all other servers. Thus, the same amount of load that one server experiences for non-parity updates will also be distributed among the other servers to update parities. While all servers experience increased load, the most-loaded server in the absence of erasure coding will likely have the smallest increase in load due to the addition of erasure coding because all other servers for whom it hosts parities have lower load. Hence, the expected difference in load between the most- and least-loaded servers decreases. Thus, while ECRec doubles the total number of updates on the servers, its impact is alleviated by improved load balancing.

Effect of entry width. We now evaluate ECRec with an increase in the size of each embedding table entry (i.e., the dense dimension). We compare ECRec with k = 4 on Criteo-4S and Criteo-2S-2D, which have the same total size, but with Criteo-2S-2D having half of the entries as Criteo-4S, and with each entry being twice as large.

While ECRec's training-time overhead with k = 4 on Criteo-4S is 22%, that on Criteo-2S-2D is 27%. The higher overhead on Criteo-2S-2D can be explained by the increased network traffic when training Criteo-2S-2D: because each entry in Criteo-2S-2D is twice as large as each in Criteo-4S, transmitting embedding table

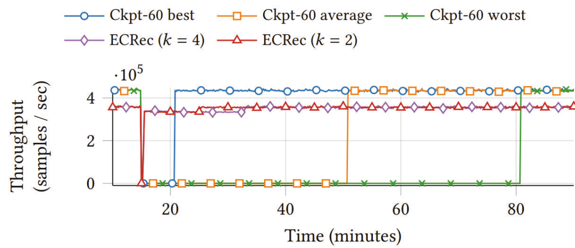


Figure 11: Training throughput after a failure at 15 min. on Criteo-4S, compared to Ckpt-60.

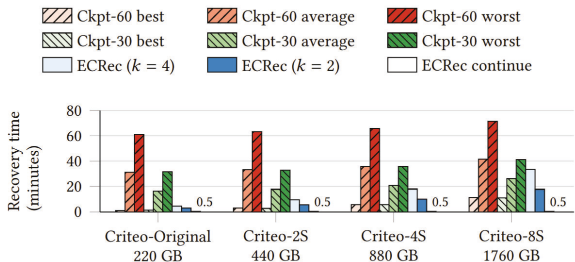


Figure 12: Time to fully recover a failed server. "ECRec continue" shows the time between a failure occurring and ECRec continuing training during recovery. This value is small enough to be imperceptible, so we also indicate it in text.

entries (and their gradients) with difference propagation consumes twice as much network bandwidth in Criteo-2S-2D as in Criteo-4S.

Ablation study. We next investigate the contributions to training-time overhead of ECRec's components.

We first consider the training-time overhead incurred for replicating NN parameters in ECRec. We compare ECRec with $k = 4$ to ECRec-NoRep, a version of ECRec that does not replicate NNs. On Criteo-Original, ECRec has training-time overhead of 18.2%, while that of ECRec-NoRep is only 5.2%. This indicates that the extra network traffic of keeping NN replicas up-to-date in ECRec adds considerable training-time overhead. Recall from §3.3, that replication of NN parameters could be avoided if one uses a stateless optimizer (e.g., SGD). Users leveraging such optimizers can potentially reduce overhead by turning off NN replication in ECRec.

We next consider the training-time overhead incurred by using two-phase commit (2PC) in ECRec (see §3.5). On Criteo-Original, we find that the training-time overhead of ECRec-NoRep in the absence of 2PC further reduces from 5.2% to only 2.6%. As described in §3.5, 2PC is used in ECRec to avoid losing updates that were in-flight when a failure occurs. Applications that are willing to forgo this (likely small) potential hit in accuracy can turn off 2PC.

5.3 Performance during recovery

We next evaluate ECRec and checkpointing in recovering from failure. Recovery is best compared in Fig. 10, which plots the throughput and training progress of ECRec and Ckpt-30 on Criteo-4S after a

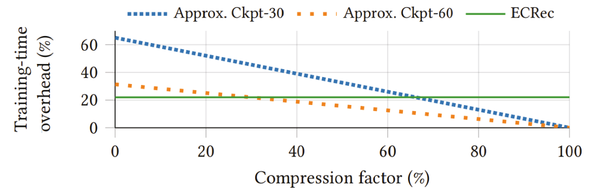


Figure 13: Training time overhead of ECRec ($k = 4$; similar for $k = 2$) and approximate checkpointing-based approaches.

server failure at time 15 minutes.[2] ECRec fully recovers faster than the average case for Ckpt-30, and, critically, maintains throughput within 7%–13% of that during normal operation during recovery. In contrast, Ckpt-30 cannot perform new iterations during recovery. As shown in the bottom of Fig. 10, ECRec's high throughput during recovery enables it to progress faster than the average case for Ckpt-30. Fig. 11 shows that ECRec's improvement in recovery time and throughput over Ckpt-60 is even larger.

Fig. 12 shows the time it takes for ECRec, Ckpt-30, and Ckpt-60 to recover a failed server. ECRec with $k = 4$ recovers 1.2–6.7× and 0.8–3.5× faster than the average case for Ckpt-60 and Ckpt-30, respectively (and up to 9.8× faster with $k = 2$). More importantly, unlike checkpointing, *ECRec enables training to continue with high throughput during recovery within 30 sec. of the failure occurring.*

Effect of parameter k and DLRM size. Fig. 12 shows that it takes longer for ECRec to fully recover with higher value of parameter k. However, ECRec maintains high throughput during recovery for each value of k: after a failure occurs, ECRec resumes training with high throughput within 30 seconds (also see Fig. 10).

Fig. 12 also shows that the time to fully recover increases with DLRM size for both ECRec and checkpointing, as expected (see §2.3 and §4). ECRec's recovery time increases more quickly with DLRM size than checkpointing due to the k-fold increase in data read and compute performed by a single server in ECRec when decoding. This does not significantly affect training because ECRec can continue training during recovery with high throughput.

Effect of lock granularity. We next compare the full recovery time of ECRec with $k = 4$ when using one and ten locks (see §3.4). Using ten locks increases recovery time by 6.5% for Criteo-8S and 24.8% for Criteo-Original. Even when employing locks with finer granularity, and thus having longer recovery time, *ECRec continues to provide high training throughput during recovery*, unlike checkpointing. Switching locks involves (1) momentarily synchronizing workers and servers, and (2) copying updated embedding table entries from buffers to the original entries. Synchronization time is constant regardless of DLRM size, whereas the time to copy buffers grows with DLRM size. Thus, synchronization time is better amortized on larger DLRMs, reducing the overhead of lock switching.

5.4 Comparison to approximate checkpointing

We now compare to approaches that use approximation to reduce the overhead of checkpointing. As described previously, ECRec differs from these approaches in that it maintains the same accuracy

[2] This can be thought of as 15 minutes after the last checkpoint has occurred, since the checkpointing intervals considered are longer than 15 minutes.

guarantees as the underlying training system, whereas approximate approaches can degrade accuracy. This potential accuracy degradation can require training for a longer period of time to reach a target accuracy, and makes it challenging to debug model performance. Existing works on approximate checkpointing do not have open-source code releases (e.g., [9, 16, 29, 38]), so we model their performance analytically and compare to real system performance of ECRec (thus favoring approximation-based baselines).

Compressed checkpointing. We first compare ECRec to approaches that compress checkpoints (e.g., by using lower precision, such as in Check-N-Run [16]). Fig. 13 shows the training-time overhead of ECRec and the checkpointing baselines on Criteo-8S for various degrees of compression. Because ECRec does not use checkpointing, its overhead is constant regardless of the compression factor. The overhead of the checkpointing baselines decreases linearly with increased compression factor (though this analytical model does not account for time overhead to perform compression and hence is more favorable to these approaches). Fig. 13 shows that Ckpt-30 requires a compression factor of over 60% to match the training-time overhead of ECRec. Even when this level of compression can be achieved, checkpointing still requires a lengthy recovery process, and the recovered DLRM from approximate versions may not match the accuracy of the original DLRM.

Partial recovery. Other techniques performs partial recovery, in which only the parameters hosted on a failed server are rolled back to a checkpoint upon failure (e.g., CPR [29]). It is difficult to model the performance of partial recovery. On one hand, recovery time for partial recovery could potentially be modelled as only that required to load the checkpoint for the lost portion of the DLRM. On the other hand, due to the inconsistent state of the recovered DLRM, the total time to reach a particular accuracy using partial recovery may differ from that of ECRec or a non-approximate checkpointing-based system. Because this added training time depends on many factors (e.g., dataset, neural network, when failure occurs, which parameters are lost), it is challenging to model and for practitioners to reason about. In contrast, ECRec does not introduce such questions and has more straightforward recovery semantics.

6 RELATED WORK

DLRM systems. System support for DLRM training and inference has received recent attention. Works on improving DLRM inference range from workload/system analysis [20, 28], model-system codesign [17, 19], and specialized hardware [22, 46]. Work related to training DLRMs includes systems from large-scale organizations (e.g., [6–8, 21, 24, 33, 35, 43, 52]), and model-system codesign [18, 48, 50]. ECRec differs from these works by its focus on fault tolerance for DLRM training and its use of erasure codes therein. ECRec could operate atop many of these works.

Checkpointing. Computer systems have long used checkpointing for fault tolerance (e.g., [11, 25, 32]). Recent works optimize checkpointing in NN training [31, 36], but do not focus on DLRMs. In contrast, ECRec leverages unique characteristics of DLRM training to use erasure codes for fault tolerance. Other works develop approximation-based techniques to reduce the overhead of checkpointing in training [9, 38]. In contrast, ECRec does not change the accuracy guarantees of the underlying training system.

Two works focus on reducing the overhead of checkpointing in DLRM training. Maeng et al. [29] use partial recovery to reduce the overhead of rolling back after failure: only the failed node rolls back to its most recent checkpoint. Eisenman et al. [16] use incremental checkpointing and reducing the precision of checkpointed parameters. Both of these works potentially reduce the accuracy of DLRM training upon recovering from failure. While both works empirically demonstrate only small accuracy drops, they cannot provide the same accuracy guarantees as the underlying training system. ECRec differs from these works in two regards: (1) ECRec maintains the same accuracy guarantees as the underlying training system. This avoids uncertainty about whether the fault tolerance approach will deliver a model with the accuracy needed for deployment, and reduces effort in debugging model accuracy when there are multiple sources of inaccuracy present. (2) ECRec leverages in-memory redundancy to reduce the overhead of fault tolerance.

Erasure-coded systems. Erasure codes are widely used in various domains for fault tolerance, load balancing, and alleviating slowdowns (e.g., [37, 39, 41, 47]). Recent work has also applied coding-theoretic ideas to NN inference [26] and in training certain classes of models (e.g., [15, 27, 44, 51]). In contrast, ECRec focuses on DLRM training, which differs significantly from the settings considered in these works.

7 CONCLUSION

ECRec is a fault-tolerant DLRM training system that employs erasure coding to overcome the downsides of checkpointing. ECRec exploits unique characteristics of DLRM training to take a hybrid approach to in-memory redundancy by erasure coding the embedding tables of DLRMs, while replicating the NN parameters. ECRec maintains up-to-date redundant parameters with low overhead and enables training to continue during recovery, while maintaining the accuracy guarantees as the underlying training system. Compared to checkpointing, ECRec reduces training-time overhead by up to 66%, recovers from failures up to 9.8× faster, and allows training to proceed without pauses. ECRec scales gracefully with increased DLRM size without enforcing a severe tradeoff between training-time overhead and recovery performance. While ECRec's benefits come with additional memory requirements and load on servers, memory overhead is only fractional and load gets evenly distributed. ECRec shows the potential of erasure coding as a superior alternative to checkpointing for fault tolerance in training current and future DLRMs. Exploring the applicability of ECRec to other learning systems is an exciting future direction to pursue.

ACKNOWLEDGMENTS

This work was funded in part by an NSF Graduate Research Fellowship (DGE-1745016 and DGE-1252522), in part by a TCS Presidential Fellowship, in part by Amazon Web Services, in part by a VMware Systems Research Award, and in part by the AIDA project (POCI-01-0247- FEDER-045907) co-financed by the European Regional Development Fund through the Operational Program for Competitiveness and Internationalisation 2020. We also thank CloudLab [14] for providing computational resources used in carrying out part of this research.

REFERENCES

[1] Criteo Labs: Download Terabyte Click Logs https://labs.criteo.com/2013/12/download-terabyte-click-logs/. Last accessed 10 July 2023.

[2] Introducing NVIDIA Merlin HugeCTR: A Training Framework Dedicated to Recommender Systems. https://tinyurl.com/yy82pd2l. Last accessed 10 July 2023.

[3] Kaggle Avazu CTR Prediction Contest. https://www.kaggle.com/c/avazu-ctr-prediction. Last accessed 10 July 2023.

[4] MLPerf Inference Github Repository. https://github.com/mlperf/inference. Last accessed 10 July 2023.

[5] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016.

[6] Bilge Acun, Matthew Murphy, Xiaodong Wang, Jade Nie, Carole-Jean Wu, and Kim Hazelwood. Understanding Training Efficiency of Deep Learning Recommendation Models at Scale. In *2021 IEEE International Symposium on High Performance Computer Architecture (HPCA 21)*, 2021.

[7] Muhammad Adnan, Yassaman Ebrahimzadeh Maboud, Divya Mahajan, and Prashant J. Nair. Accelerating Recommendation System Training by Leveraging Popular Choices, 2022.

[8] Saurabh Agarwal, Ziyi Zhang, and Shivaram Venkataraman. BagPipe: Accelerating Deep Recommendation Model Training. *arXiv preprint arXiv:2202.12429*, 2022.

[9] Yu Chen, Zhenming Liu, Bin Ren, and Xin Jin. On Efficient Constructions of Checkpoints. In *Proceedings of the International Conference on Machine Learning (ICML 20)*, 2020.

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

[11] John T Daly. A Higher Order Estimate of the Optimum Checkpoint Interval for Restart Dumps. *Future Generation Computer Systems*, 22(3):303–312, 2006.

[12] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. In *International Conference on Learning Representations (ICLR 15)*, 2015.

[13] John Duchi, Elad Hazan, and Yoram Singer. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *Journal of Machine Learning Research*, 12(7), 2011.

[14] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The Design and Operation of CloudLab. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019.

[15] Sanghamitra Dutta, Ziqian Bai, Haewon Jeong, Tze Meng Low, and Pulkit Grover. A Unified Coded Deep Neural Network Training Strategy Based on Generalized Polydot Codes for Matrix Multiplication. In *Proceedings of the 2018 IEEE International Symposium on Information Theory (ISIT 18)*, 2018.

[16] Assaf Eisenman, Kiran Kumar Matam, Steven Ingram, Dheevatsa Mudigere, Raghuraman Krishnamoorthi, Krishnakumar Nair, Misha Smelyanskiy, and Murali Annavaram. Check-N-Run: A Checkpointing System for Training Deep Learning Recommendation Models. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022.

[17] Assaf Eisenman, Maxim Naumov, Darryl Gardner, Misha Smelyanskiy, Sergey Pupyrev, Kim Hazelwood, Asaf Cidon, and Sachin Katti. Bandana: Using Non-Volatile Memory for Storing Deep Learning Models. In *The Second Conference on Systems and Machine Learning (SysML 19)*, 2019.

[18] AA Ginart, Maxim Naumov, Dheevatsa Mudigere, Jiyan Yang, and James Zou. Mixed Dimension Embeddings with Application to Memory-Efficient Recommendation Systems. In *2021 IEEE International Symposium on Information Theory (ISIT 21)*, 2021.

[19] Udit Gupta, Samuel Hsia, Vikram Saraph, Xiaodong Wang, Brandon Reagen, Gu-Yeon Wei, Hsien-Hsin S Lee, David Brooks, and Carole-Jean Wu. DeepRecSys: A System for Optimizing End-to-End At-Scale Neural Recommendation Inference. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA 20)*, 2020.

[20] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, David Brooks, Bradford Cottel, Kim Hazelwood, Mark Hempstead, Bill Jia, et al. The Architectural Implications of Facebook's DNN-based Personalized Recommendation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA 20)*, 2020.

[21] Biye Jiang, Chao Deng, Huimin Yi, Zelin Hu, Guorui Zhou, Yang Zheng, Sui Huang, Xinyang Guo, Dongyue Wang, Yue Song, et al. XDL: An Industrial Deep Learning Framework for High-Dimensional Sparse Data. In *Proceedings of the 1st International Workshop on Deep Learning Practice for High-Dimensional Sparse Data*, 2019.

[22] Wenqi Jiang, Zhenhao He, Shuai Zhang, Thomas B Preußer, Kai Zeng, Liang Feng, Jiansong Zhang, Tongxuan Liu, Yong Li, Jingren Zhou, et al. MicroRec: Accelerating Deep Recommendation Systems to Microseconds by Hardware and Data Structure Solutions. In *The Fourth Conference on Systems and Machine Learning (MLSys 21)*, 2021.

[23] Saurabh Kadekodi, Francisco Maturana, Suhas Jayaram Subramanya, Juncheng Yang, K. V. Rashmi, and Gregory R. Ganger. Pacemaker: Avoiding heart attacks in storage clusters with disk-adaptive redundancy. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association.

[24] Dhiraj Kalamkar, Evangelos Georganas, Sudarshan Srinivasan, Jianping Chen, Mikhail Shiryaev, and Alexander Heinecke. Optimizing Deep Learning Recommender Systems' Training On CPU Cluster Architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 20)*, 2020.

[25] Richard Koo and Sam Toueg. Checkpointing and Rollback-Recovery for Distributed Systems. *IEEE Transactions on software Engineering*, (1):23–31, 1987.

[26] Jack Kosaian, K. V. Rashmi, and Shivaram Venkataraman. Parity Models: Erasure-Coded Resilience for Prediction Serving Systems. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP 19)*, 2019.

[27] Kangwook Lee, Maximilian Lam, Ramtin Pedarsani, Dimitris Papailiopoulos, and Kannan Ramchandran. Speeding Up Distributed Machine Learning Using Codes. *IEEE Transactions on Information Theory*, July 2018.

[28] Michael Lui, Yavuz Yetim, Özgür Özkan, Zhuoran Zhao, Shin-Yeh Tsai, Carole-Jean Wu, and Mark Hempstead. Understanding Capacity-Driven Scale-Out Neural Recommendation Inference. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 21)*, 2021.

[29] Kiwan Maeng, Shivam Bharuka, Isabel Gao, Mark C Jeffrey, Vikram Saraph, Bor-Yiing Su, Caroline Trippel, Jiyan Yang, Mike Rabbat, Brandon Lucia, et al. CPR: Understanding and Improving Failure Tolerant Training for Deep Learning Recommendation with Partial Recovery. In *The Fourth Conference on Systems and Machine Learning (MLSys 21)*, 2021.

[30] Peter Mattson, Christine Cheng, Cody Coleman, Greg Diamos, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, et al. MLPerf Training Benchmark. In *The Third Conference on Systems and Machine Learning (MLSys 20)*, 2020.

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

[32] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R De Supinski. Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC 10)*, 2010.

[33] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Zhihao Jia, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, et al. Software-Hardware Co-design for Fast and Scalable Training of Deep Learning Recommendation Models. *arXiv preprint arXiv:2104.05158*, 2021.

[34] Maxim Naumov, John Kim, Dheevatsa Mudigere, Srinivas Sridharan, Xiaodong Wang, Whitney Zhao, Serhat Yilmaz, Changkyu Kim, Hector Yuen, Mustafa Ozdal, et al. Deep Learning Training in Facebook Data Centers: Design of Scale-up and Scale-out Systems. *arXiv preprint arXiv:2003.09518*, 2020.

[35] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G Azzolini, et al. Deep Learning Recommendation Model for Personalization and Recommendation Systems. *arXiv preprint arXiv:1906.00091*, 2019.

[36] Bogdan Nicolae, Jiali Li, Justin M Wozniak, George Bosilca, Matthieu Dorier, and Franck Cappello. DeepFreeze: Towards Scalable Asynchronous Checkpointing of Deep Learning Models. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGrid 20)*, 2020.

[37] David A. Patterson, Garth Gibson, and Randy H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 88)*, 1988.

[38] Aurick Qiao, Bryon Aragam, Bingjing Zhang, and Eric Xing. Fault Tolerance in Iterative-Convergent Machine Learning. In *International Conference on Machine Learning*, pages 5220–5230, 2019.

[39] K. V. Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. EC-Cache: Load-Balanced, Low-Latency Cluster Caching with Online Erasure Coding. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016.

[40] K. V. Rashmi, Nihar B Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. A Hitchhiker's Guide to Fast and Efficient Data Reconstruction in Erasure-Coded Data Centers. In *Proceedings of the 2014 ACM SIGCOMM Conference (SIGCOMM 14)*, 2014.

[41] Luigi Rizzo. Effective Erasure Codes for Reliable Computer Communication Protocols. *ACM SIGCOMM Computer Communication Review*, 27(2):24–36, 1997.

[42] Mahesh Sathiamoorthy, Megasthenis Asteris, Dimitris Papailiopoulos, Alexandros G Dimakis, Ramkumar Vadali, Scott Chen, and Dhruba Borthakur. XORing Elephants: Novel Erasure Codes for Big Data. *Proceedings of the VLDB Endowment*, 6(5), 2013.

[43] Geet Sethi, Bilge Acun, Niket Agarwal, Christos Kozyrakis, Caroline Trippel, and Carole-Jean Wu. Recshard: Statistical feature-based memory optimization for industry-scale neural recommendation. In *Proceedings of the Twenty-Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 22)*, 2022.

[44] Rashish Tandon, Qi Lei, Alexandros G Dimakis, and Nikos Karampatziakis. Gradient Coding: Avoiding Stragglers in Distributed Learning. In *International Conference on Machine Learning (ICML 17)*, 2017.

[45] Hakim Weatherspoon and John D Kubiatowicz. Erasure Coding vs. Replication: A Quantitative Comparison. In *International Workshop on Peer-to-Peer Systems (IPTPS 2002)*, 2002.

[46] Mark Wilkening, Udit Gupta, Samuel Hsia, Caroline Trippel, Carole-Jean Wu, David Brooks, and Gu-Yeon Wei. RecSSD: Near Data Processing for Solid State Drive Based Recommendation Inference. In *Proceedings of the Twenty-Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 21)*, 2021.

[47] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A Chien, and Haryadi S Gunawi. Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, 2017.

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

[49] Juncheng Yang, Anirudh Sabnis, Daniel S. Berger, K. V. Rashmi, and Ramesh K. Sitaraman. C2DN: How to harness erasure codes at the edge for efficient content delivery. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1159–1177, Renton, WA, April 2022. USENIX Association.

[50] Chunxing Yin, Bilge Acun, Xing Liu, and Carole-Jean Wu. TT-Rec: Tensor Train Compression for Deep Learning Recommendation Models. In *The Fourth Conference on Systems and Machine Learning (MLSys 21)*, 2021.

[51] Qian Yu, Netanel Raviv, Jinhyun So, and A Salman Avestimehr. Lagrange Coded Computing: Optimal Design for Resiliency, Security and Privacy. In *Proceedings of the 22nd International Conference on Artificial Intelligence and Statistics (AISTATS 19)*, 2019.

[52] Weijie Zhao, Jingyuan Zhang, Deping Xie, Yulei Qian, Ronglai Jia, and Ping Li. AIBox: CTR Prediction Model Training on a Single Node. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management (CIKM 2019)*, 2019.