IncrCP: Decomposing and Orchestrating Incremental Checkpoints for Effective Recommendation Model Training

Qingyin Lin
Sun Yat-Sen University, Peng Cheng Laboratory
linqy35@mail2.sysu.edu.cn

Jiangsu Du*
Sun Yat-Sen University
Guangzhou, China
dujiangsu@mail.sysu.edu.cn

Rui Li
Peng Cheng Laboratory
Shenzhen, China
lir@pcl.ac.cn

Zhiguang Chen
Sun Yat-Sen University
Guangzhou, China
chenzhg29@mail.sysu.edu.cn

Wenguang Chen
Tsinghua University, Peng Cheng Laboratory
cwg@tsinghua.edu.cn

Nong Xiao
Sun Yat-Sen University
Guangzhou, China
xiaon6@mail.sysu.edu.cn

ABSTRACT

Training large models for modern recommendation systems requires a substantial number of computational devices and extended periods. Since it is essential to store model checkpoints throughout the training progress for accuracy debugging or mitigating potential failures, checkpointing systems are widely used. However, given that recommendation models can scale to hundreds of gigabytes or more, existing solutions often introduce significant overhead in terms of both storage and I/O.

In this paper, we present IncrCP, a checkpointing system specifically designed for recommendation models. Given that only a small fraction of model parameters are modified in each iteration, IncrCP creatively leverages the incremental checkpointing strategy and overcomes the inherent slow recovery problem. To support recovering all states throughout the training process, while also ensuring efficient storage utilization and rapid recovery, IncrCP proposes the 2-D chunk approach. It proactively records changed parameters in the training process as well as their indexes, extracts parameters according to duplicated indexes as independent chunk files, and then orchestrates these chunks in the 2-dimensional linked list. In this way, IncrCP achieves fast recovery by loading less unnecessary parameters and performing less deduplication during recovery. Furthermore, IncrCP includes a selective extraction approach to reduce I/O by avoiding worthless extractions and a concatenate approach to reduce random disk access when recovery. Evaluations show that IncrCP achieves up to 6.6× recovery speedup compared to the naive incremental strategy and saves storage space by 60.4% with slight overhead compared to another recovery-friendly strategy.

PVLDB Reference Format:
Qingyin Lin, Jiangsu Du, Rui Li, Zhiguang Chen, Wenguang Chen, and Nong Xiao. IncrCP: Decomposing and Orchestrating Incremental Checkpoints for Effective Recommendation Model Training. PVLDB, 18(4): 1049 - 1062, 2024.
doi:10.14778/3717755.3717765

*Corresponding author.

Proceedings of the VLDB Endowment, Vol. 18, No. 4 ISSN 2150-8097.
doi:10.14778/3717755.3717765

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/linqy71/IncrCP_paper.git.

1 INTRODUCTION

Our daily life has been powerfully assisted by recommendation systems, including daily usage of services from social media platforms [18], e-commerce applications [42, 44] and video entertainment websites [7, 15], etc. As reported by Meta [19], over 50% of machine learning training demands are attributed to recommendation model training at Meta's datacenter. Similar demands can be found at other companies, such as Google, Amazon, and Alibaba.

Training large recommendation models often requires excessive computing resources, e.g. occupying a large-scale high-performance GPU clusters for days or even months. During such a long time, the training process is inevitably interrupted by unexpected failures, including hardware crushes, software malfunctions and network disconnections. Frequent failures would even prevent the training from succeeding since once a failure occurs the whole training process aborts. To mitigate this, checkpointing becomes a fault tolerance mechanism and a fundamental component of deep learning training frameworks [41]. Through periodically storing model states to persistent storage, it enables training to recover from the last saved state despite failures. In addition to failure recovery, where the last checkpoint is saved, checkpoints are also needed in many cases, such as debugging and transfer learning [41], where all checkpoints should be kept. However, the large size of recommendation models, often reaching hundreds of gigabytes or more, challenges current checkpointing systems, making them inefficient due to slow recovery and unacceptable storage usage.

It is imperative for checkpointing systems to optimize three key metrics, checkpoint recovery time, checkpoint construction time, and checkpoint storage usage, to ensure efficiency and practicality. Firstly, the recovery process should be fast in case of a failure and therefore reduce the system blocking time. Secondly, constructing checkpoints blocks the training process, and the system can benefit from a shorter construction time. Although asynchronous checkpointing systems [32, 34, 35, 46] allow the I/O time of persisting checkpoints from CPU memory to storage to overlap with GPU computation, the training process is still blocked by the GPU-CPU data transfer operation. Thirdly, storing checkpoints throughout

the training [26, 30, 41, 46] demands significant storage space and incurs substantial costs. For example, saving 154 checkpoints when training Pythia-12B requires 11TB storage and would cost $5000 a month on a general cloud server [26]. Therefore, an efficient checkpointing system should minimize overhead in both blocking time and storage space, so as to support high-frequency checkpointing. Related works put efforts on improving recovery performance [32, 34, 35, 46] and reducing checkpoint size [3, 6, 20, 22], but overlook the unique property of recommendation models.

During recommendation model training, only a small fraction of the model parameters are updated after each iteration. This feature provides a unique opportunity for checkpoints to save part of the model instead of all parameters as in DNN model training, where all parameters are updated after each iteration. Given the unique feature, two classical approaches, incremental checkpointing and differential checkpointing are potentially suited for recommendation models. The two methods have been widely adopted in high performance computing applications [1, 12, 23, 25, 37, 43] and operating systems [14, 31], yet seldom in deep model checkpointing.

Incremental checkpointing strategy records updates since last checkpoint, capturing only changes between successive checkpoints. In comparison, differential checkpointing strategy tracks the differences relative to the baseline checkpoint (typically the first checkpoint) and records parameters changed from the initial state. Owing to the shorter interval of tracking, hence a reduced data volume, incremental checkpointing inherently surpasses differential checkpointing in terms of construction time and storage consumption. However, the incremental strategy tends to be very slow in recovery for it requires loading all checkpoint files and processing excessive duplication among these files. On the contrary, recovering from differential checkpoints is faster for it only needs to load two files, i.e. the baseline checkpoint and the file that records the difference, with less data volume and avoiding duplicated updates. Check-N-Run [10] uses differential checkpointing for recommendation models, which achieves fast recovery and storage saving compared to the full checkpoint strategy. Further, it intermittently resets the differential interval to reduce storage consumption but still consumes substantial storage. In comparison, the incremental checkpointing strategy takes less storage space, while it is generally overlooked in previous works due to poor recovery performance.

In this work, we reexamine the incremental checkpointing strategy for recommendation model training and introduce a novel checkpointing system IncrCP. IncrCP leverages the advantages of incremental checkpointing, i.e. fast construction and low storage consumption, while overcoming the inherent shortcomings in recovery speed and providing the capability to recover from any arbitrary checkpoint. The key idea behind IncrCP is the 2-D chunk approach, which decomposes successive checkpoints into chunks and orchestrates these chunks in a 2-dimensional linked list. One dimension of the linked list records changes in parameters throughout the training process, while the other dimension tracks duplication between checkpoints. As a result, IncrCP avoids loading unnecessary parameters and eliminates the need for deduplication during recovery, thereby addressing the inherent limitations in recovery speed. In details, the 2-D chunk approach is inspired by the famous Log-Structured Merged Tree (LSM-Tree) used in many modern systems, such as Cassandra [24], RocksDB [11] and PebblesDB [40]. It proactively records all indexes of changed parameters in the training process, and stores these indexes and corresponding parameters into a chunk file when checkpointing. Then it compares this newly-added chunk file with all previous chunks and extracts duplicated indexes as independent chunks, so as to avoid loading and processing duplicated data in recovery. In this way, a checkpoint is composed of multiple chunks and different checkpoints share some chunks. To better manage these chunks, the 2-D chunk approach orchestrates them into a 2-dimensional linked list and achieves fast recovery by reading chunks in a specific route.

Furthermore, IncrCP solves two major problems in the extraction of duplicates. First, the former chunks of previous checkpoints need to be loaded into memory and subsequently written back to storage after retrieving duplicates, introducing excessive overhead. We mitigate this problem by setting three level of filtering to avoid small extractions where there is minimal duplication between chunks. Second, frequent extractions can generate many small fragmented files, resulting in large random reads when recovery, which is not disk friendly. We address this problem by concatenating small chunks into large files during extractions, while ensuring the correctness of subsequent extractions. We summarize contributions of our paper as follows:

- We reexamine the incremental strategy for fast and space-efficient recommendation model checkpointing, by coordinating the sparse update feature of models. To the best of our knowledge, we are the first to apply incremental checkpointing strategy for recommendation model training.
- We propose the 2-D chunk approach to overcome the inherent slow recovery of the incremental checkpointing strategy, by managing checkpoint chunks in a 2-dimensional linked list.
- We propose a selective extraction approach that reduces the I/O overhead of the background chunk management, enabling fast and frequent foreground checkpointing.
- We propose a concatenation approach that repairs the fragmented storage caused by the chunk management, improving the recovery performance with less random disk access.

Extensive evaluations demonstrate that IncrCP significantly improves recovery time over the naive incremental strategy and greatly reduces storage consumption compared to the differential strategy, with only a minor trade-off in other metrics. For example, when checkpointing the popular recommendation model DLRM and using a hard disk drive, IncrCP achieves a 6.6× faster recovery time than the naive incremental strategy and reduces storage space by 60.4% compared to the differential strategy.

2 BACKGROUND

2.1 Recommendation Models

Deep recommendation models are widely-used in existing recommendation systems. They predict user interests by analyzing users' prior interactions. Figure 1 illustrates the architecture of a popular recommendation model, DLRM [33], proposed by Meta. Since the recommendation scenario usually takes continuous inputs (e.g. user's age, login time) and categorical inputs (e.g. user's rating for an item), recommendation models use combinations of the multi-layer perceptron (MLP) and the embedding tables to process them

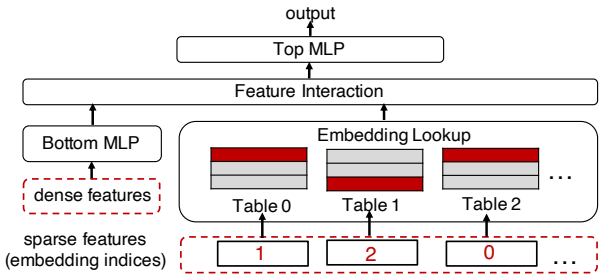


Figure 1: The architecture of DLRM.

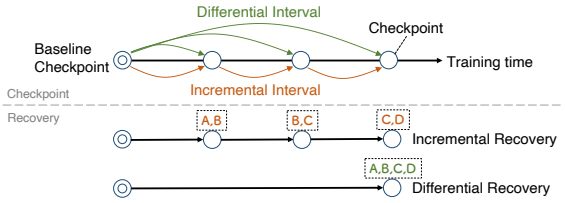


Figure 2: The working process of differential checkpointing and incremental checkpointing strategies.

separately. The dense features, representing continuous inputs, are processed by the bottom MLP directly. The sparse features, representing categorical inputs, are actually embedding indices indicating user's preferences to items in the embedding tables. The sparse features are converted to embeddings by looking up corresponding rows in the embedding tables [47]. In this way, the categorical inputs are mapped from sparse to dense representations, which can be combined and computed with the output of the bottom MLP. Since recommendation systems mostly process sparse features, such as hundreds of billions of products and billions of users, parameters of embedding tables generally dominate the recommendation models (> 99%) and can scale to hundreds of gigabytes or more. For instance, the DLRM model in the popular MLPerf benchmark [29] consists of around 100GB of embedding tables and 10MB of other parameters.

In recommendation models, only a minor proportion of embedding parameters are updated after each iteration [10]. The embedding layers comprise massive embedding tables and each table might contain millions of embedding vectors, where every vector corresponds to one category in the sparse feature. In the forward phase, the embedding table looks for the matching embedding vectors of categorical inputs to be further processed by the upper MLP. Simultaneously, in the backward phase, only the vectors that participated in the forward phase will undergo updates. Thus, considering that embedding tables largely contribute to the overall model size, there are relatively few updates involved in each training iteration of a recommendation model. For example, when training a 25GB DLRM model with a batch size of 1024, only 0.02% of embedding vectors are updated after an iteration.

2.2 Incremental and Differential Strategies

The checkpoint mechanism in computing, particularly in the context of training large models, is a vital technique used to manage and safeguard ongoing computations. This mechanism involves periodically saving the state of a system or model to persistent storage. In general model training, such as CNN or Transformer models, parameters are fully stored to persistent storage after several iterations of training. In comparison, recommendation models only update a small fraction of embedding tables after iterations, there is no necessity to checkpoint all the embedding tables each time. Thus, we tend to explore checkpointing strategy that better suited for recommendation models, i.e. differential and incremental checkpointing strategies.

Both incremental and differential checkpointing strategies begin with the storage of a baseline checkpoint, which represents parameters of the entire model, as depicted in Figure 2. Subsequent to the initial step, both approaches proceed to capture snapshots of the updated parameters at each following checkpoint. The primary distinction between these two approaches lies in the intervals they track. Differential checkpointing tracks changes over an interval from the baseline checkpoint to the current checkpoint, whereas the incremental checkpointing captures the changes that occur from one checkpoint to the next, tracking only the updates made since the last saved state. Consequently, incremental checkpointing monitors a shorter span and diminishes the checkpoint volume, presenting great advantage in reducing both I/O and storage space.

However, incremental checkpointing strategy generally presents much slower speed [10] compared to the differential strategy when recovering from a target checkpoint. During the recovery process, the baseline checkpoint is loaded initially. Given that differential checkpoints encompass all parameters updated in relation to the baseline checkpoint, only two files are necessary for recovery. Conversely, in incremental recovery, all files between the target checkpoint and the baseline checkpoint are needed. Thus, lots of files are required to be loaded and these files include massive repeated parameters. In the circumstance of checkpointing recommendation models, there exists an abundance of duplicated embedding indices among incremental files, indicating the same embedding vectors updated during different incremental intervals. Redundant data loading and deserialization, as well as complex deduplication processing lead to slow recovery process, compared to the differential strategy and even the fully checkpointing strategy. In our experiment of DLRM training, the data loading volume of one incremental recovery can be easily more than 3× compared to that of one differential recovery, and the gap will be further enlarged as the training progresses.

To summarize, the incremental checkpointing strategy naturally saves I/O and storage space, while it fails in achieving fast recovery compared with other strategies.

3 MOTIVATION

3.1 Decomposing Incremental Checkpoints

Existing checkpointing systems generally provide interfaces to save user-given content into single file when storing checkpoints. However, this approach loses the opportunity to manage checkpoint data at a fine-grained level. As is mentioned above, the embedding tables of recommendation models are composed of embedding vectors,

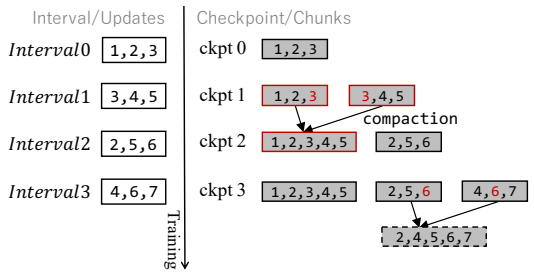


Figure 3: The timeline of how chunks are organized into checkpoints in the LSM-Tree.

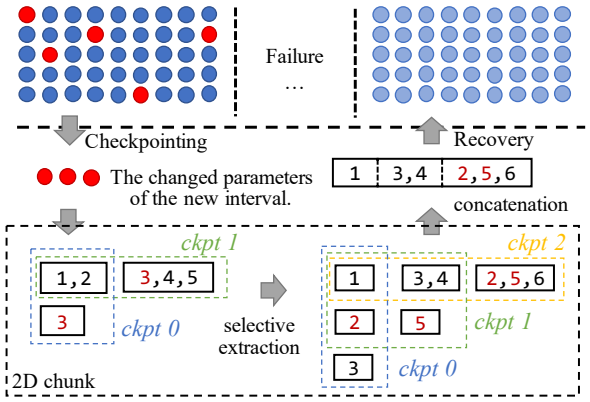


Figure 4: The Overview of IncrCP. The baseline checkpoint is not displayed here.

and an incremental checkpoint of the recommendation model contains the updated embedding vectors since last checkpoint. Therefore, an intuitive way to decompose checkpoint data is leveraging key-value stores, where the indexes of vectors act as keys and the embedding vectors act as values. In this way, the key-value stores are able to arrange checkpoint data at the key-value pair granularity. There are many popular key-value stores [9, 11, 16, 24, 40] serving for different application scenarios, among which checkpointing is write-intensive for checkpoints are created frequently.

The Log-Structured Merge Tree (LSM-tree) [36] is a data structure that is particularly effective for write-intensive tasks. Basically, it provides high write performance by buffering recent updates in memory and flushing them into disk subsequently, which is very similar to performing an incremental checkpoint. To be specific, the LSM-Tree manages data as key-value pairs, appends data into storage to be log-structured chunks, then partially merges data to maintain its orderliness through compaction operations. Moreover, the LSM-Tree serves queries by searching for keys across its memory buffer and potentially multiple chunks in storage. Therefore, the LSM-Tree is able to support checkpoint-recovery and the recovery requires to query for all needed chunks.

In short, we seize the opportunity to decompose incremental checkpoints based on key-value stores, among which the LSM-Tree based key-value store is potential to be applied for managing incremental checkpoints of recommendation model training.

3.2 Chunk-based incremental checkpoints

The LSM-Tree based key-value stores provide a sub-optimal solution to orchestrate incremental checkpoints in chunks. Since the LSM-Tree appends incremental data into storage as chunks, all the added chunks can form a complete checkpoint. Figure 3 shows an example of how chunks are organized to form a checkpoint for recovery. In the *Interval*0, the chunk that records the updated parameters is appended to the tree, forming the first checkpoint(*ckpt*0). Next, the incremental updates of the next interval(*Interval*1) is appended, forming the second checkpoint(*ckpt*1) together with the former added chunk. Then, when the chunk containing the updated parameters of *Interval*2 is appended, a compaction is triggered to merge former chunks into a more sorted chunk. Specifically, the compaction loads key-value pairs in the two chunks into memory, sorts them in key order while discarding duplicated keys, and writes sorted key-value pairs into a new chunk. The new chunk, together with the appended chunk in *Interval*2, form the third checkpoint(*ckpt*2). Subsequently, updates in *Interval*3 is appended and the existing three chunks form the fourth checkpoint(*ckpt*3). At this point, a compaction is triggered and going to generate new chunks for the next checkpoint. In this way, the chunks keeps being appended to form new checkpoints, and the compactions keep merging and deduplicating key-value pairs among existing chunks.

Although the LSM-Tree based key-value stores are able to organize checkpoints in chunks and presents some advantages, it still encounters the following shortcomings. Firstly, the LSM-Tree is unable to maintain multiple checkpoints simultaneously, therefore it lacks the ability of arbitrary checkpoint recovery. Each time a new chunk is appended to the LSM-Tree, only the latest checkpoint is maintained, as the compaction removes necessary key-value pairs of former checkpoints. For example, in Figure 3, the information of *ckpt*0 has lost in *Interval*2 due to the compaction. Secondly, the LSM-Tree is unable to deduplicate keys from a global perspective, missing further opportunities for duplication elimination and reducing I/O burden. It can only deduplicate keys within the most recent chunks, as performing deduplication across all chunks would impose an unacceptable I/O burden.

4 THE DESIGN OF INCRCP

4.1 Overview

IncrCP is a checkpointing system designed for the recommendation model training. It reexamines the overlooked incremental checkpointing strategy, aiming at saving storage space and achieving fast recovery speed. In addition to leveraging the inherent characteristic of saving storage space, IncrCP also incorporates novel approaches to enhance recovery speed, addressing the traditional shortcoming of the incremental strategy. Notably, here we focus exclusively on the dominant embedding tables of recommendation models and disregard the MLPs, which only constitute a very small proportion.

Figure 4 demonstrates an overview of IncrCP, which decomposes and orchestates checkpoints in chunks. Specifically, it is composed of a 2-D chunk approach, a selective extraction approach, and a concatenation approach. The key concept behind IncrCP is the 2-D chunk approach. At first, the 2-D chunk approach of IncrCP

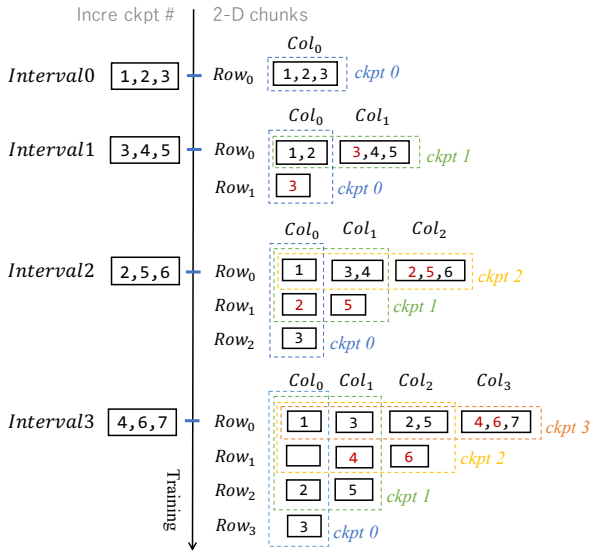


Figure 5: The timeline of how multiple checkpoints are managed by the 2-D chunk approach.

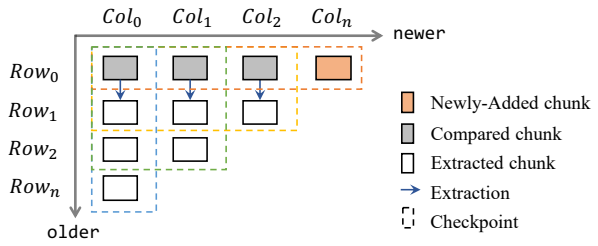


Figure 6: The illustration of how a newly-added chunk is inserted into the 2-dimensional linked list.

records the indexes of parameters that have changed. During the checkpointing process, these indexes, along with the associated parameters, are stored in a chunk file. This chunk is then asynchronously compared with all previous chunks to identify and extract data with duplicated indexes from previous chunk into new chunks. When a checkpoint recovery is required, IncrCP retrieves chunks following a specific route and these chunks are spliced together in advance to accelerate data access.

Besides, in Figure 4, since all the former chunks of previous checkpoints need to be loaded into memory and subsequently written back to storage after retrieving duplicates, introducing excessive overhead and slowing down checkpointing efficiency. The selective extraction approach mitigates this by setting three level of filtering to avoid small extractions where there is minimal duplication between chunks. Frequent extractions can generate many small fragmented files, resulting in large random reads during recovery, which is not disk friendly. The concatenation approach addresses this by concatenating small chunks into large files during extractions, while ensuring the correctness of subsequent extraction.

4.2 2-D Chunk Approach

The 2-D chunk approach enables fast recovery for the incremental checkpointing strategy and acts as the core idea behind IncrCP. Figure 5 demonstrates the timeline of how multiple checkpoints are managed by the 2-D chunk approach and Figure 6 shows how a checkpoint is inserted into and recovered from the 2-dimensional linked list. Here we firstly introduce the 2-D chunk approach in a step-by-step style, following Figure 5, and then introduce it in a more general way, following Figure 6.

The 2-D chunk approach is based on the idea of the incremental checkpointing strategy and supports recovering from an arbitrary checkpoint. It only records and stores these incrementally changed parameters within the training interval. In the beginning, a file that records the baseline checkpoint, or rather the full model states, is stored. Notably, the baseline checkpoint is not displayed in Figure 5 and Figure 6. Next, as the model training progressing, these changed parameters will be recorded in the form of sorted key-value pairs and stored as a chunk.

In the *Interval*0 of Figure 5, the chunk that records the changed parameters is joined to the empty linked list and it is directly added into the first row (Row_0). Next, the chunk that stores incremental updates of the next interval (*Interval*1) is arrived. As is mentioned before, recovery from incremental checkpoints suffers from redundant updates among multiple checkpoints, and we introduce an extraction process to eliminate this. In detail, 3 of *Interval*0 is extracted to form an independent chunk, which is inserted into the next row (Row_1). Until now, if we need to recover from the most recent model state, only loading two chunks in Row_0 is enough. Subsequently, when the chunk containing updates of *Interval*2 is joined, another extraction is triggered. As detailed in Figure 6, the chunk will be compared with all other chunks in Row_0. Then, two new chunks will be inserted into Row_1 and the original chunk of Row_1 will be pushed to Row_2. In *Interval*3, the newly-added chunk does not have duplicates compared to the chunk in the intersection of Row_0 and Col_0, an empty node will still be inserted into Row_1.

Basically, as shown in Figure 6, the 2-D chunk approach constantly extracts duplicates and arranges the extracted data in the 2-dimensional linked list to balance between storage space and recovery speed. The newly-added chunk will be compared with all the other chunks in Row_0. If duplicated keys are found in former Row_0 chunks, the corresponding key-value pairs will be extracted from its original chunk and inserted into Row_1. Thus, an original incremental checkpoint will be extracted multiple times and split into multiple chunks. After comparing and extracting, these extracted key-value pairs will be written to generate a new chunk and always inserted into Row_1 of the 2-dimensional linked list. If there is no duplication, an empty node plays as a placeholder and will be inserted into Row_1. Further, the insertion pushes all rows to their next rows. In this way, all chunks that reside in the same row are the results of the same round of extraction.

Based on the 2-D chunk approach, we can recover from any one of checkpoints without duplicated data through the well organized chunks in the linked list. Thus, to recover from a specific checkpoint, denoted as $ckpt_i$, we get its maximum depth and width, and loads chunks within this rectangle to form the entire checkpoint. Other

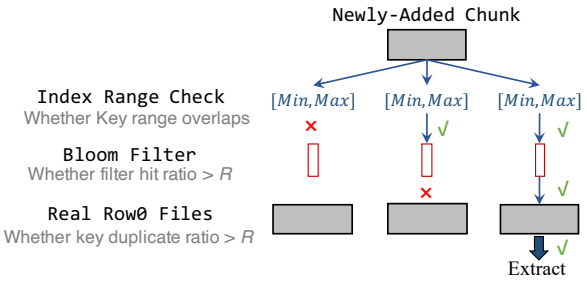


Figure 7: The Selective Extraction Approach.

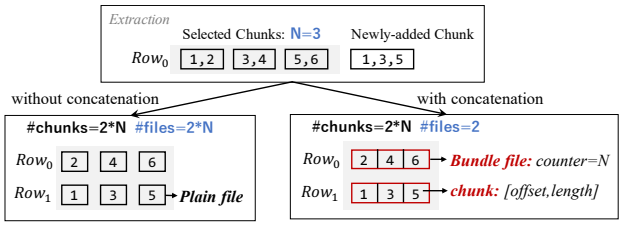


Figure 8: The concatenation approach.

chunks outside this rectangle is ignored. Since there is no duplication between these chunk files, the complex deduplication process is not necessary compared to the naive incremental strategy.

In summary, the 2-D chunk approach targets at improving the inherent shortcoming of the naive incremental checkpointing strategy, i.e. the recovery speed. It decomposes incremental checkpoints into chunks and extracts duplicates between these chunks, so as to avoid loading unnecessary parameters and eliminate the deduplication process in recovery. To better manage these chunks, the 2-D chunk approach arranges chunks in a 2-dimensional linked list.

4.3 Reduce Comparing Overhead by the Selective Extraction Approach

The construction of the 2-dimensional linked list requires the comparing stage that each newly-added chunk will be compared with all previous chunks in Row_0, introducing I/O and CPU processing overhead. If the training executes for an extended period and a single 2-dimensional linked list manages numerous checkpoints, frequent extraction at each checkpoint interval leads to increased overhead, which significantly impairs the performance of the checkpointing system. Specifically, when performing extractions, all chunks in Row_0 should be loaded into memory, do extraction, and then be written back to persistent storage. Although extractions are executed asynchronously, the overhead increases as training progresses, finally blocking the foreground checkpoint writes.

To reduce the extraction overhead, we propose a selective extraction approach. It balances the extraction overhead during checkpointing with the benefits of redundant loading during recovery. Notably, the extraction overhead can be partly hidden with the training process. Since the recommendation model updates only a small proportion of parameters over a limited number of iterations, performing extraction for every chunk results in few duplicates between chunks and generates many extremely small chunks. Therefore, we find it unnecessary to execute thorough extractions.

As shown in Figure 7, the selective extraction is composed of three distinct filter stages to determine whether it is worthwhile to do extraction. Further, it is able to reduce the data loading volume and the number of small files. The first stage is called the index range check. We store the minimum and maximum keys as the metadata of each chunk. At the arrival of the newly-added chunk, we can check the metadata and determine whether the key range of this chunk falls in key ranges of other chunks. If overlaps, it becomes meaningful to do extraction.

The second stage is based on the bloom filter [4], which has been adopted by many key-value stores to accelerate queries[27, 49, 50]. A Bloom filter can indicate whether an element is possibly in a set, with extremely low space overhead and query latency. A bloom filter is a bit-vector, initially set to all zeros, that records the existence of a key in a set according to the hash value of the key. To create a bloom filter, each key in the set is calculated with multiple hash functions, then the corresponding bits are set to 1 with hash values act as indexes. To query for the existence of a key is to calculate the hash values and check the corresponding bits. If all the corresponding bits are 1, then the key is in the set, which is called a hit. However, the answer of the bloom filter may be false-positive due to hash collision. When a chunk of Row_0 is generated, the corresponding bloom filter of the chunk is also generated. In this way, we can get a hit ratio indicating the key duplicated ratio between the target chunk and the newly-added chunk, by querying the bloom filter of the target chunk with keys in the newly-added chunk. Here we set the threshold of the hit ratio as R. If the hit ratio is larger than R, the comparison can enter the next stage. Otherwise, the related extraction process is ignored this time.

The third stage directly checks whether it is worthwhile to do extraction by calculating actual number of duplicated keys during extraction process. Although chunks have been loaded into memory, it is still worthy of terminating the extraction. The frequent extraction can generate extremely small chunks, and presents challenges due to the fragmented data storage, further damaging the overall performance. In this way, we calculate the duplicated ratio by real keys and determine whether to do extraction through comparing the duplicated ratio with R.

4.4 Reduce Random Access Overhead by the Concatenation Approach

Although incremental checkpoints are organized in chunks and able to eliminate most redundant data access, the recovery process cannot fully take advantage of the disk's sequential read capabilities due to the presence of small files. When performing extractions, each chunk is written to its own file, resulting in massive small files in the storage. To address this, we propose the concatenation approach for enhancing the 2-D chunk approach. Basically, we concatenate chunks of the same row into a file during extractions.

As shown in Figure 8, a newly-added chunk triggers an extraction, performing deduplication for N Row_0 chunks, where $N = 3$. When performing without concatenation, this extraction generates total $2 * N$ new files, N files in Row_0 and N files in Row_1, with each file contains only one chunk, called a plain file. Instead, the

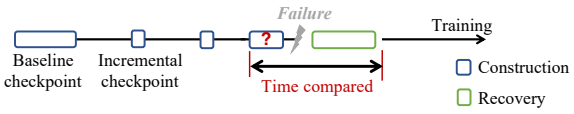


Figure 9: Decide the type of checkpoints by comparing the time consumed by checkpoint construction and recovery.

concatenation approach generates 2 new files, one for Row_0 and one for Row_1, with each file contains N chunks, called a bundle file. A $[offset, length]$ label is assigned to each chunk, indicating its offset in the bundle file and its data length. A counter is initialized as N for the generated Row_0 bundle file, indicating the number of chunks inside the bundle file.

After concatenation, the bundle file of Row_0 may encounter subsequent extractions, resulting in modifications of internal chunks. When a chunk inside the bundle file joins an extraction, it will be divided into two chunks, which are written into new files in Row_0 and Row_1. In this case, the chunk becomes invalid in the original bundle file and the related counter is reduced by 1. When the counter becomes 0, the bundle file will be removed. The concatenation may require additional storage space intermediately, but the overhead is acceptable and can be eliminated after checkpoints accumulate.

To quantify the number of files accessed during recovery, here we assume L checkpoints are generated and related number of chunks are added to Row_0. When we recover from the i_{th} checkpoint ($1 \leq i \leq L$), chunks within a rectangle with depth $d_i = L - i + 1$ and width $w_i = i$ are accessed. Firstly, we compare the number of files in Row_0, denoted as N_0. Without the concatenation approach, $N_0 = i$. With the concatenation approach, $N_0 < i$, thanks to the presence of bundle files. Secondly, we compare the number of files that are extracted to rows deeper than Row_0, denoted as N_+. Without the concatenation approach, $N_+ = (L - i) * i$, while $N_+ = (L - i)$ when adopting the concatenation approach. This is because each row maintains only one bundle file. Consequently, the number of files accessed during recovery is significantly reduced, allowing more sequential reads to be exploited.

4.5 Restrict Recovery Latency by Resetting the Baseline Checkpoint

Due to the inherent nature of the lengthening of the recovery route in the incremental strategy, recovery time increases as training progresses. Specifically, the increase comes from the growing number of updated parameters over the baseline checkpoint. Therefore, it is necessary to store a full model intermittently so that the differential view (i.e. the updated parameters since the baseline checkpoint) can be reset.

Before constructing a checkpoint, we always assume there is a coming failure after current checkpoint. Then, we decide whether to generate a baseline checkpoint by comparing the time of checkpoint construction and recovery, as shown in Figure 9. In other words, the type of a specific checkpoint is decided considering which one can lead to shorter end-to-end training time. The construction time is easy to obtain during training while the recovery time has to be estimated. In our experiments, we observe that the recovery latency of IncrCP shows a near-linear increase as checkpoint accumulates. Thus, the recovery time can be estimated according to history recoveries. The algorithm for the decision is as follows:

Let C_{base} and C_{incre} be the time spent on constructing a baseline checkpoint and an incremental checkpoint, respectively. The time spent on recovering from a baseline checkpoint is denoted as R_{base}. The time consumed by loading the updated parameters since the baseline checkpoint when recovering from $ckpt_i$ is denoted as R_i. When constructing a baseline checkpoint, the construction and recovery time (denoted as T_{base}) can be expressed as

$$T_{base} = C_{base} + R_{base}.$$

When constructing an incremental checkpoint, the construction and recovery time (denoted as T_{incre}) can be expressed as

$$T_{incre} = C_{incre} + R_{base} + R_i,$$

where R_i is inferred from the last two recoveries. If $T_{base} < T_{incre}$, the baseline checkpoint is generated for reset. When the baseline checkpoint is generated, a 2-dimensional linked list will be created relatively. In this way, the growing size of rows and columns in our 2-D chunk approach is limited, and the recovery time is restricted.

5 EVALUATION

This section firstly illustrates the effectiveness of IncrCP by conducting single-node experiments and presenting the overall performance across different storage devices. Next, we conduct a sensitivity study to show the impacts of our optimization solutions on IncrCP. At last, to show the generalizability of IncrCP, we apply IncrCP to two additional recommendation models and evaluate it under larger checkpoint interval and distributed setting.

5.1 Experimental Setup

5.1.1 ***Implementation.*** We develop IncrCP using C++ and provide Python interfaces for integrating with *Pytorch*. We employ MessagePack [13] to perform serialization and deserialization for chunk files. Particularly, to identify which embedding vectors are modified with specific inputs, we track the behavior of the embedding lookup operations and record which embedding vector the input data flows through. Notably, we only take this process in the forward pass. When applying IncrCP to recommendation model training, each embedding table is managed by a separate instance of a 2D linked list. Therefore, it is straightforward to apply IncrCP to distributed training, where model parallelism is applied and embedding tables are distributed to different GPUs.

Table 1: Device Access Bandwidth.

Device	Sequential Wr.(MB/s)	Sequential Rd.(MB/s)	Random Wr.(MB/s)	Random Rd.(MB/s)
HDD	104	163	1.6	2.5
Flash SSD	1405	2463	1042	577
3D X. SSD	2182	2528	1457	2210
Lustre	119	7278	117	6867

5.1.2 ***Testbed.*** We conduct experiments on a server that has 56 CPU cores, 256GB DRAM, and 4 Tesla V100 GPU, each with 16GB of memory. The server runs CentOS 7.2 with CUDA ToolKit 12.0 and Pytorch 1.13.1. For persistent storage, the server is equipped with a 1TB NAND flash-based NVMe SSD , a 726GB HDD, and a 688GB 3D XPoint based NVMe SSD . We also conduct distributed experiments on a cluster where each node is configured with 56 CPU cores, 1 TB of DRAM, and 8 A800 GPUs, each with 80GB of memory. The nodes are connected via infiniband network, and checkpoints are stored using Lustre [5], a parallel distributed file system. Detailed metrics for these storage systems are presented in Table 1, with results obtained using *fio* under 16 concurrent jobs.

Table 2: Model training configurations.

Scale	Model	Dataset	Model size (GB)	Batch size
Single Node	DLRM [33]	Kaggle	25	1024
	DeepFM [17]	Kaggle	21	512
	PNN [39]	Kaggle	21	512
Distributed	DLRM [33]	Terabyte	220	16384

5.1.3 ***Models and training configurations.*** As shown in Table 2, we select and configure three popular recommendation models, DLRM [33], DeepFM [17] and PNN [39], for training on a single node using the Criteo Kaggle dataset [21], with model sizes of 25GB, 21GB and 21GB respectively. Further, we train DLRM on Criteo Terabyte dataset [8] and conduct the distributed training on 2 nodes, with the model size up to 220GB. In the single node training, we set the checkpoint frequency to iteration-level as in CheckFreq [32]. A checkpoint is generated every 10 iterations. Total 150 checkpoints are saved for DLRM [33], and 100 checkpoints are saved for DeepFM [17] and PNN [39]. In the distributed training, a checkpoint is generated every 4000 iterations. To evaluate the recovery performance, we recover from each checkpoint for all the saved checkpoints and record the time consumed. Notably, these recovery time are measured after the all checkpoints are generated.

5.1.4 ***Metrics.*** We evaluate the effectiveness of different checkpointing systems with three metrics: 1)**Recovery time:** The time consumed of recovering from a specific checkpoint. It consists of I/O and deserialization time. 2) **Construction time:** The time consumed for constructing a checkpoint, including the time for copying data from GPU memory to CPU memory (G2C) and the time for copying data from CPU memory to persistent storage (C2S). The C2S construction time can be overlapped with the GPU computation and the G2C construction time blocks the GPU computation. 3) **Storage consumption:** The storage space required for checkpointing, including baseline checkpoints. Note that the baseline checkpoint and dense parameters are ignored when recording checkpoint construction time and recovery time.

5.1.5 ***Baseline Methods.*** There are two baseline approaches:

- **Check-N-Run**: The differential checkpointing strategy introduced in Check-N-Run [10]. This approach is implemented using the same serialization/deserialization library, i.e. MessagePack, for fair comparison. Here we also take its baseline reset strategy.
- **Naive Incre**: The naive incremental checkpointing strategy, which requires to load massive duplicates and execute complex deduplication process in recovery, theoretically good at storage consumption while weak in recovery speed. This approach is also implemented atop MessagePack for fair comparison.

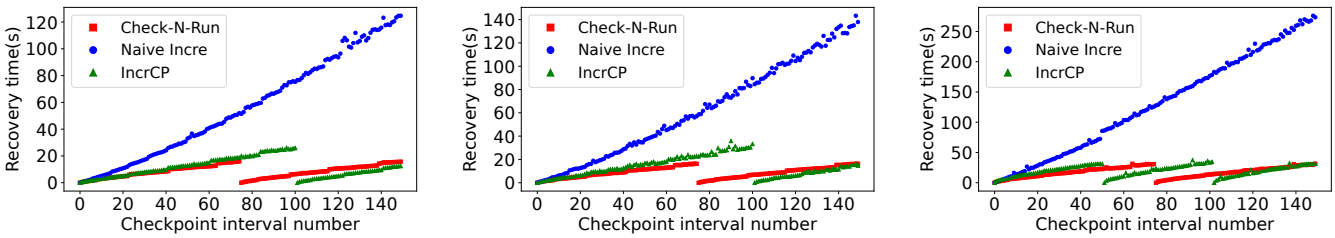
5.2 Overall Performance

This section evaluates the overall performance of IncrCP by conducting the single-node experiment. Since the recommendation model splits embedding tables into different nodes for scaling up and each embedding table is managed by a separate instance of a 2D linked list, the single-node experiment can largely illustrate the effectiveness of IncrCP. The single-node experiment trains DLRM with Kaggle dataset on three different kinds of storage devices, i.e. 3D XPoint SSD, NAND Flash SSD and HDD. We set $R = 0.02$ for the selective extraction approach and turn on the concatenation approach of IncrCP. Notably, here we only report the G2C construction time from the overall training perspective.

5.2.1 Recovery Time. For the recovery time, as demonstrated in Figure 10, IncrCP shows a significant improvement compared to Naive Incre and a comparable performance compared to Check-N-Run. It illustrates that IncrCP successfully overcomes the inherent bottleneck of the incremental checkpointing strategy, where the recovery time is generally unacceptable. In detail, over the 150 checkpoints, IncrCP achieves an average recovery time reduction of 6.6×, 4.7× and 5.2×, compared to that of Naive Incre on HDD, Flash SSD and 3D XPoint SSD, respectively. In other words, IncrCP significantly outperforms Naive Incre on all three types of storage devices. This aligns with our expectations. Through the decomposition and orchestration of incremental checkpoints, IncrCP loads fewer files, processes smaller data volumes, and related less deserialization. Additionally, the recovery time of Naive Incre is consistently and rapidly increasing, which validates its unacceptability.

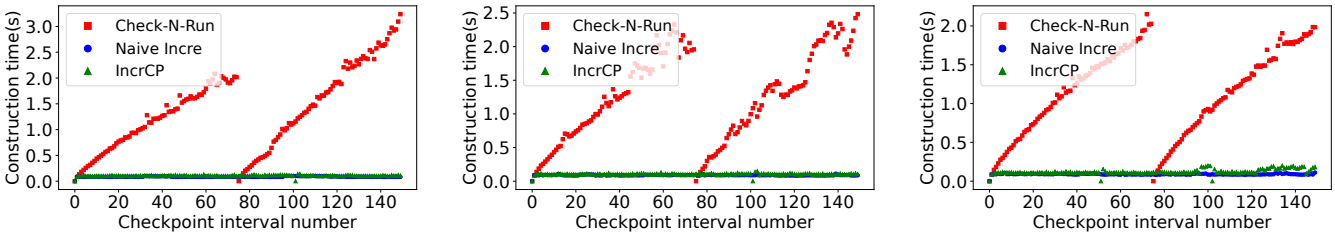
Next, we compare the recovery time of these three storage devices. Although the trend of the absolute recovery time in Figure 10 fits the trend of the access bandwidth illustrated in Table 1, it does not fit the ratio. For instance, the 3D XPoint SSD is more than 10 times faster than the HDD, while the recovery time is only approximately half that of the HDD. This is because, from the time breakdown, the deserialization dominates the overall recovery time. Among these three devices, our IncrCP shows the greatest speed-up when using an HDD. During the 150 checkpoints, training with an HDD requires three times the baseline reset, whereas other configurations only require twice the baseline reset. This is because, in comparison, the ratio of sequential read to random read in an HDD is much larger than that of the other two storage devices. Thus, IncrCP can achieve better speedup when employing the HDD as the storage device.

When compared to Check-N-Run, as shown in Figure 10c, IncrCP achieves a comparable recovery time. In detail, IncrCP incurs



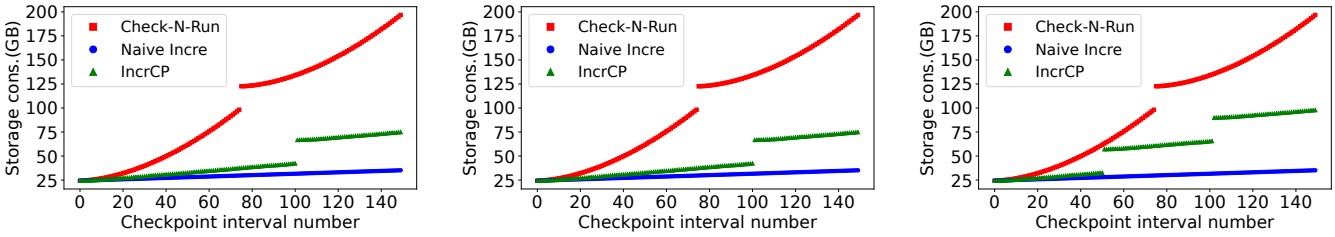
(a) 3D XPoint SSD. (b) NAND Flash SSD. (c) HDD.

Figure 10: The recovery time on different storage devices as checkpoint number increases.



(a) 3D XPoint SSD. (b) NAND Flash SSD. (c) HDD.

Figure 11: The construction time (G2C) on different storage devices as checkpoint number increases.



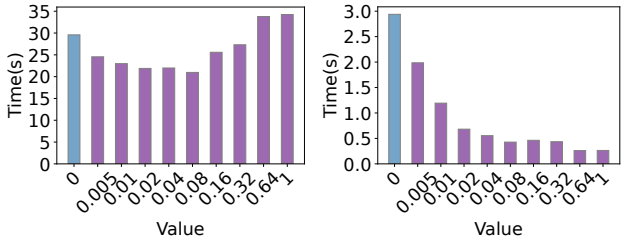
(a) 3D XPoint SSD. (b) NAND Flash SSD. (c) HDD.

Figure 12: The storage consumption on different storage devices as checkpoint number increases.

an average of 1.1×, 1.5× and 1.3× recovery time increase on HDD, NAND Flash SSD and 3D XPoint SSD respectively. Note that IncrCP achieves worse recovery time compared to Check-N-Run within a given interval range, for example, from interval 74 to 100 in Figure 10a. This difference arises from the distinct baseline checkpoint reset policies of Check-N-Run and IncrCP . At interval 74, Check-N-Run resets its baseline checkpoint, causing the recovery time to reset and start from 0 at interval 75. In contrast, IncrCP does not reach its baseline reset threshold at this interval, leading to a continued increase in recovery time. However, resetting the baseline checkpoint introduces significant overhead in storage space and checkpoint construction time, making it a trade-off decision.

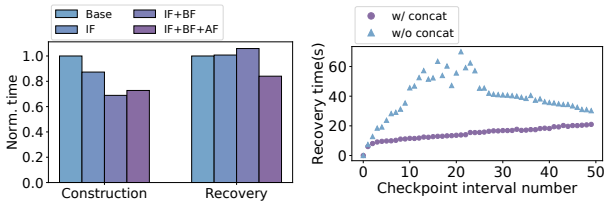
5.2.2 Construction time. As shown in Figure 11, the checkpoint construction time (G2C) for IncrCP is extremely close to that for Naive Incre, while much lower in comparison to Check-N-Run. In detail, Naive Incre is 26% faster than IncrCP using HDD, 13.6% faster using NAND Flash SSD, and 16.4% faster using 3D XPoint SSD when constructing these checkpoints. This also indicates that the background operations of IncrCP have only a slight impact on the foreground G2C data transfer. As for Check-N-Run, its construction time grows linearly when the number of checkpoints grows, largely exceeding that of both incremental approaches. Although it only reaches several seconds in this experiment, significantly less than the recovery time, the construction time is accumulated throughout the training process, while the recovery time may only be triggered once over a long period, or possibly not at all.

5.2.3 Storage consumption. As shown in Figure 12, IncrCP consumes a little more storage than Naive Incre but significantly less storage than Check-N-Run. IncrCP only consumes 44%, 34%, and 34% storage in comparison to Check-N-Run on HDD, NAND Flash SSD and 3D XPoint SSD, after dumping all 150 checkpoints. Compared to Naive Incre, IncrCP requires more storage. The major disadvantage comes from the baseline checkpoint reset, which can be considered a hyperparameter for trade-off.



(a) Average recovery time. (b) Average construction time.

Figure 13: The selective extraction approach of IncrCP with varying R values. Here the construction time includes both G2C and C2S.



(a) Selective Extraction. (b) Concatenation.

Figure 14: Ablation results for the selective extraction approach and the concatenation approach.

5.3 Sensitivity & Ablation Study

This section conducts the sensitivity study of the selective extraction approach , and the ablation study of both the selective extraction and the concatenation approach. They were tested separately, excluding the influence of each other. The selective extraction approach relies on the threshold R of duplicated ratio to determine whether to trigger an extraction. Concatenation is enabled with extraction. Here we select HDD in the following experiments.

5.3.1 Selective Extraction. The selective extraction approach helps reduce the data volume to be loaded during recovery by sacrificing additional disk accesses. Therefore, the selection of the hyperparameter R will have varying effects on recovery time and construction time. Figure 13a shows the average recovery time over 50 checkpoints when using the HDD device with varying R. When $R = 0$, the recovery process makes full extraction and loads the least data volume. As R increases, more extractions are ignored and the data volume also increases. Thus, the number of files decreases because of less extractions. The recovery performance is related to both data volume and number of files accessed, especially on random-access sensitive devices like HDD. Notably, the construction time reported here is from the perspective of the checkpointing subsystem and includes both the transfer time from GPU to CPU and the transfer time from CPU to persistent storage.

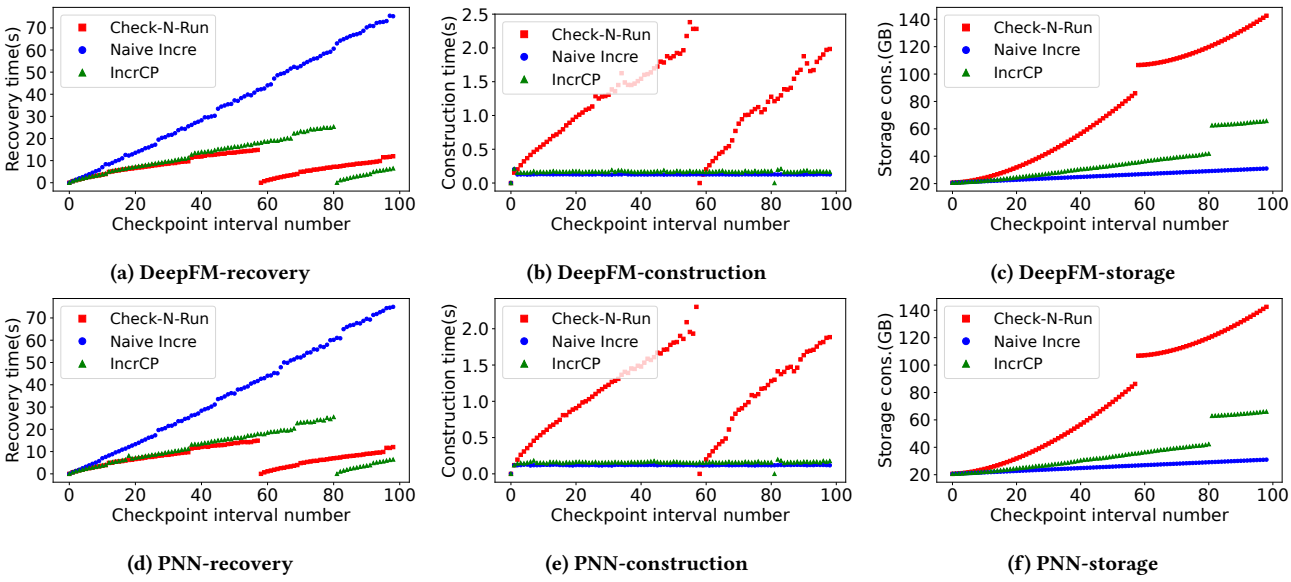
For the checkpoint construction time, it reduces as R increases, shown in Figure 13b. When $R = 0$, the I/O and CPU computational overhead is largest. As R increases, less extractions are performed resulting in shorter construction time. Since each extraction modifies the state of Row_0 chunks, the next extraction has to wait until the current extraction accomplished. Furthermore, since R determines the least number of parameters to be extracted from a chunk, it should be carefully set according to number of parameters updated during one checkpoint interval. A larger interval generally deserves a smaller R value. It is essential to make a trade-off by choosing a suitable R.

Besides the sensitivity study, we also conduct an ablation study to assess the contribution of each selection stages. We incrementally add these three filtering stages, i.e. index filter, bloom filter, and actual filter, to the baseline: 1) Base: none of these stages are applied; 2) IF: applying index range check only; 3) IF+BF: applying index range check and bloom filter hit ratio check; and 4) IF+BF+AF: applying all three stages. We select 0.02 as the R threshold value.

Figure 14a presents the average construction and recovery time normalized to the baseline results. Basically, applying all three stages (IF+BF+AF) results in a significant speedup for both construction and recovery compared to the baseline (Base). Initially, the IF and the IF+BF considerably improves the construction time, but slightly worsens the recovery time, due to the reduction of unnecessary extraction and an increase in small files. However, when the actual filter is added, construction time increases slightly, while recovery time significantly decreases, achieving a more optimal trade-off between the two. This is because the Bloom filter may produce false positives in its predictions and makes it fail to intercept all small extractions. The actual filter fixes this problem by terminating the extraction after loading files, and this can largely reduces the number of small files at the expense of useless I/O. Thus, it slightly worsens the construction time, and largely benefits the recovery time. Notably, selecting an appropriate threshold value for each filter is crucial for achieving an optimal performance trade-off. However, in this case, we use a unified value for all filters.

5.3.2 Concatenation. The concatenation approach concatenates extraction outputs of the same row into the same file. To evaluate this approach, we set $R = 0$ to perform full extraction and compare the recovery performance with and without concatenation. The evaluation is also carried out on HDD. Figure 14b shows that the recovery time significantly increases when the concatenation approach is disabled, due to large amount of small files generated by extraction. Thus, the impact of random accesses can be largely mitigated by the concatenation approach.

Notably, the recovery time without the concatenation approach initially increases and then decreases as the number of checkpoint intervals grows. This is because the recovery time measured is not for the most recent checkpoint; we measure the recovery time of these checkpoints after all checkpoints have already been generated. This fits the design of IncrCP and can be clearly investigated in Figure 6. The number of small files when retrieving the n_{th} checkpoint is $(L-n)*n$. Therefore, the number of random disk accesses attains its maximum when $n = L/2$. Although the data volume grows with the increase of n, the influence of random accesses can exceed that of data volume. In contrast, when applying the concatenation approach, the recovery time exhibits a linear increase corresponding to the linear growth in data volume.



(a) DeepFM-recovery (b) DeepFM-construction (c) DeepFM-storage

(d) PNN-recovery (e) PNN-construction (f) PNN-storage

Figure 15: The overall performance when training DeepFM and PNN.

5.4 Generalizability Study

5.4.1 Different Models. Apart from DLRM, there are many other recommendation models. To verify the generality of IncrCP, we further trained DeepFM [17] and PNN [39] on the 3D XPoint SSD. As depicted in Figure 15, the results for both DeepFM and PNN are quite similar to those for DLRM. Additionally, DeepFM and PNN show almost the same trends in recovery, construction, and storage. This similarity can be explained by the fact that they use the same dimension for the embedding vector. Since the incremental checkpointing strategy only stores and processes these updated embedding vectors. With the same dataset, they only update the same number of embedding vectors under a given batch size.

5.4.2 Distributed training. We conduct distributed training for the 220GB DLRM using the larger Criteo Terabyte dataset, as shown in Figure 16. The embedding tables of DLRM are distributed across 2 nodes and checkpoints are generated every 4000 iterations with a batch size of 16,384. Since the experiment takes extremely long period, we only perform a limited number of intervals. Basically, the distributed experiments show very similar trend as in the single-node experiments. In terms of recovery time, IncrCP is very close to Check-N-Run and averagely 35% faster than Naive Incre. Also, the recovery time gap between the IncrCP and the Naive Incre keeps growing as more checkpoints are generated.

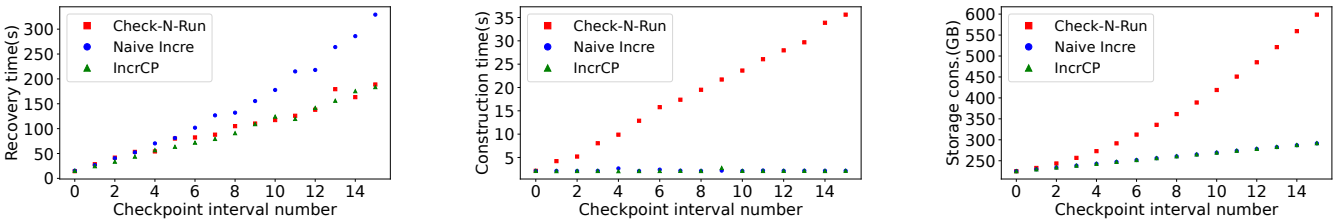
As for construction time, Check-N-Run can spend tens of seconds for transferring data from GPU memory to CPU memory, which is averagely 8× slower than that of IncrCP and Naive Incre. Moreover, the construction time of IncrCP is only 2% higher than that of Naive Incre for the background operations of IncrCP have only a slight impact on the foreground G2C data transfer. For the storage consumption, although IncrCP introduces additional storage usage for storing metadata than Naive Incre, it can be ignored compared to the large size of checkpoint data itself, as in Figure 16c.

5.4.3 Varying checkpoint interval. We verify the effectiveness under different checkpoint intervals for the 220GB DLRM model training. We set checkpoint interval as 2000, 3000, 4000, and 5000 and generate 10 checkpoints in total. Figure 17a demonstrates the experimental results of the 10th checkpoint. Since this cluster uses a shared file system, transferring data from CPU memory to persistent storage can cause contention with other programs. We report both the GPU-to CPU (G2C) and CPU-to-Storage (C2S) construction times from the perspective of checkpointing subsystem.

Figure 17a demonstrates the recovery time, construction time (G2C + C2S), and the checkpoint size of the 10th checkpoint under different interval length. At first, for each interval length, our IncrCP can maintain advantages over Check-N-Run and Naive Incre in terms of all metrics. In details, as the interval length increases, IncrCP maintains a similar recovery time compared to Check-N-Run, while offering more advantages compared to Naive Incre. This is because a larger interval length lead to more duplication between checkpoints and Naive Incre requires to handle more deduplication operations. For both construction time (logarithmic ordinate) and the checkpoint size, the incremental approach goes up slowly, whereas the differential approach grows more rapidly, as the interval length increases. This occurs because a larger interval length is more likely to exist duplication between checkpoints. It indicates that the construction time is largely attributed to the checkpoint size, and they both scale linearly with the multiplication of checkpoint interval length and batch size.

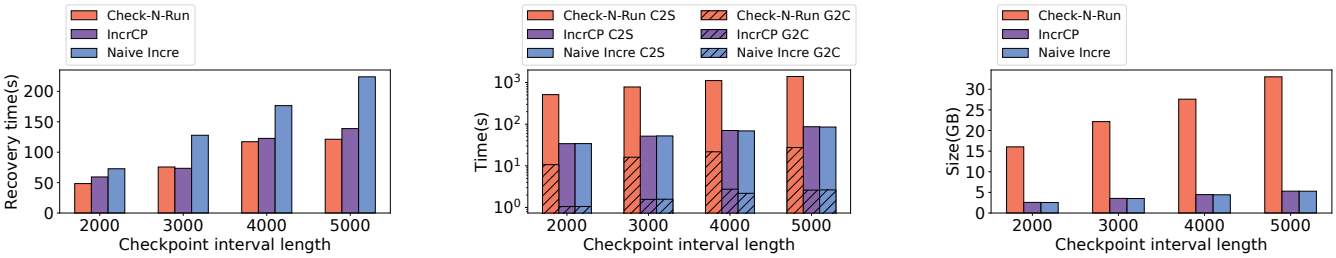
5.5 Summary

IncrCP can significantly eliminate the disadvantage of the incremental checkpointing strategy, achieving a recovery time comparable to Check-N-Run. Simultaneously, IncrCP successfully preserves most of the advantages of the incremental checkpointing strategy, largely outperforming the differential checkpointing strategy in terms of



(a) Recovery time. (b) Construction time (G2C). (c) Storage consumption.

Figure 16: The overall performance for checkpointing a 220GB DLRM model at distributed training.



(a) Recovery time. (b) Construction time (G2C + C2S) (c) Checkpoint size.

Figure 17: The performance at the 10th checkpoint under various checkpoint interval lengths.

construction time and storage consumption. Overall, IncrCP largely improves these three key metrics compared to existing approaches.

6 RELATED WORK

Fault tolerance for deep learning training. To handle unexpected system failures, deep learning training systems usually leverage additional memory resources for parameter backup. ECRec [48] replicates and encodes DLRM parameters with erasure coding among parameter servers. DLRover-RM [45] caches model parameters in its caching system during job migration to reduce remote persistent storage accesses. Bagpipe [2] shards and distributes DLRM parameters with more parameter servers to reduce checkpointing overhead of each server. Although memory-based fault tolerance approaches can improve the robustness of training systems, storing training states as checkpoints in persistent storage remains more reliable, albeit with a higher time consumption. There are two main ways: checkpoint content compression and system-level optimization, to improve performance of checkpointing systems base on persistent storage. For content compression, SCAR [38] and CPR [28] adopt partial recovery, and Check-N-Run[10] leverages quantization. For system-level optimization, DeepFreeze[34], CheckFreq[32] and Gemini[46] use asynchronous checkpointing to overlap I/O with computation, which are orthogonal to IncrCP.

Incremental Checkpointing. Incremental checkpointing has been adopted in many applications such as high performance computing [1, 12, 25, 43] and operating systems[14, 31]. However, these works focus on tracking incremental data because most applications do not give information about data modification. In recommendation model training, tracking model state update is straightforward. Therefore, IncrCP focuses on providing fast recovery, which is a totally different scope.

7 CONCLUSION

This paper presents IncrCP, a checkpointing system designed specifically for recommendation models. Basically, IncrCP innovatively adopts the incremental checkpointing strategy and overcomes its inherent shortcoming of recovery speed, by decomposing checkpoints in chunks and orchestrating these chunks in a 2-dimensional linked list. In this way, IncrCP can avoid the loading of unnecessary parameters and eliminate the need for deduplication during the recovery process. IncrCP also includes a selective extraction approach to reduce I/O by avoiding worthless extractions and a concatenate approach to reduce random disk access when recovery. Experimental results demonstrate that IncrCP successfully eliminate the inherent disadvantage of the naive incremental checkpointing strategy and make it feasible to real use. For instance, it reduces recovery time by 6.6× compared to naive incremental strategy and reduces storage space by 60.4% compared to to differential strategy on HDD.

ACKNOWLEDGMENTS

This work is supported by the National Key R&D Program of China under Grant NO. 2022YFB4500304, the Major Key Project of PCL, the National Natural Science Foundation of China(NSFC): 62272499, 62332021, and 62402534, the GuangDong Basic and Applied Basic Research Foundation: 2023A1515110117, CCF-Tencent Rhin o-Bird Open Research Fund: CCF-TencentRAGR20240102, R&D Program of Shenzhen under Grant No. 202407293000344, and Guangdong Province Special Support Program for Cultivating High-Level Talents: 2021TQ06X160.

REFERENCES

[1] Saurabh Agarwal, Rahul Garg, Meeta S Gupta, and Jose E Moreira. 2004. Adaptive incremental checkpointing for massively parallel systems. In *Proceedings of the 18th annual international conference on Supercomputing*. 277–286.

[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* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 348–363. https://doi.org/10.1145/3600006.3613142

[3] Amey Agrawal, Sameer Reddy, Satwik Bhattamishra, Venkata Prabhakara Sarath Nookala, Vidushi Vashishth, Kexin Rong, and Alexey Tumanov. 2023. DynaQuant: Compressing Deep Learning Training Checkpoints via Dynamic Quantization. *arXiv preprint arXiv:2306.11800* (2023).

[4] Burton H Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.

[5] Peter Braam. 2019. The Lustre storage architecture. *arXiv preprint arXiv:1903.01955* (2019).

[6] Yu Chen, Zhenming Liu, Bin Ren, and Xin Jin. 2020. On efficient constructions of checkpoints. *arXiv preprint arXiv:2009.13003* (2020).

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

[8] CriteoLabs. 2013. Terabyte Click Logs. https://labs.criteo.com/2013/12/download-terabyte-click-logs/

[9] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS operating systems review* 41, 6 (2007), 205–220.

[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] Facebook. [n.d.]. RocksDB. https://rocksdb.org/.

[12] Kurt B Ferreira, Rolf Riesen, Patrick Bridges, Dorian Arnold, and Ron Brightwell. 2014. Accelerating incremental checkpointing for extreme-scale computing. *Future Generation Computer Systems* 30 (2014), 66–77.

[13] Sadayuki Furuhashi. 2013. MessagePack. *URL: https://msgpack. org* (2013).

[14] Roberto Gioiosa, Jose Carlos Sancho, Song Jiang, and Fabrizio Petrini. 2005. Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers. In *SC'05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*. IEEE, 9–9.

[15] Carlos A Gomez-Uribe and Neil Hunt. 2015. The netflix recommender system: Algorithms, business value, and innovation. *ACM Transactions on Management Information Systems (TMIS)* 6, 4 (2015), 1–19.

[16] Google. [n.d.]. LevelDB. https://github.com/google/leveldb.

[17] Huifeng Guo, Ruiming Tang, Yunming Ye, Zhenguo Li, and Xiuqiang He. 2017. DeepFM: a factorization-machine based neural network for CTR prediction. *arXiv preprint arXiv:1703.04247* (2017).

[18] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, David Brooks, Bradford Cottel, Kim Hazelwood, Mark Hempstead, Bill Jia, et al. 2020. The architectural implications of facebook's dnn-based personalized recommendation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 488–501.

[19] Samuel Hsia, Udit Gupta, Mark Wilkening, Carole-Jean Wu, Gu-Yeon Wei, and David Brooks. 2020. Cross-stack workload characterization of deep recommendation systems. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 157–168.

[20] Zhenbo Hu, Xiangyu Zou, Wen Xia, Sian Jin, Dingwen Tao, Yang Liu, Weizhe Zhang, and Zheng Zhang. 2020. Delta-DNN: Efficiently compressing deep neural networks via exploiting floats similarity. In *Proceedings of the 49th International Conference on Parallel Processing*. 1–12.

[21] Olivier Chapelle Jean-Baptiste Tien, joycenv. 2014. Display Advertising Challenge. https://kaggle.com/competitions/criteo-display-ad-challenge

[22] Haoyu Jin, Donglei Wu, Shuyu Zhang, Xiangyu Zou, Sian Jin, Dingwen Tao, Qing Liao, and Wen Xia. 2023. Design of a quantization-based dnn delta compression framework for model snapshots and federated learning. *IEEE Transactions on Parallel and Distributed Systems* 34, 3 (2023), 923–937.

[23] Kai Keller and Leonardo Bautista-Gomez. 2019. Application-level differential checkpointing for HPC applications with dynamic datasets. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 52–61.

[24] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS operating systems review* 44, 2 (2010), 35–40.

[25] Kyushick Lee, Michael B Sullivan, Siva Kumar Sastry Hari, Timothy Tsai, Stephen W Keckler, and Mattan Erez. 2019. Gpu snapshot: checkpoint offloading for gpu-dense systems. In *Proceedings of the ACM International Conference on Supercomputing*. 171–183.

[26] 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).

[27] Guanlin Lu, Young Jin Nam, and David HC Du. 2012. BloomStore: Bloom-filter based memory-efficient key-value store for indexing of data deduplication on flash. In *2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 1–11.

[28] Kiwan Maeng, Shivam Bharuka, Isabel Gao, Mark Jeffrey, Vikram Saraph, Bor-Yiing Su, Caroline Trippel, Jiyan Yang, Mike Rabbat, Brandon Lucia, et al. 2021. Understanding and improving failure tolerant training for deep learning recommendation with partial recovery. *Proceedings of Machine Learning and Systems* 3 (2021), 637–651.

[29] Peter Mattson, Christine Cheng, Gregory Diamos, Cody Coleman, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, et al. 2020. Mlperf training benchmark. *Proceedings of Machine Learning and Systems* 2 (2020), 336–349.

[30] 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.

[31] John Mehnert-Spahn, Eugen Feller, and Michael Schoettner. 2009. Incremental checkpointing for grids. In *Linux Symposium*, Vol. 120. Citeseer.

[32] 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.

[33] 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).

[34] 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.

[35] Bogdan Nicolae, Adam Moody, Gregory Kosinovsky, Kathryn Mohror, and Franck Cappello. 2021. Veloc: Very low overhead checkpointing in the age of exascale. *arXiv preprint arXiv:2103.02131* (2021).

[36] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33 (1996), 351–385.

[37] Konstantinos Parasyris, Kai Keller, Leonardo Bautista-Gomez, and Osman Unsal. 2020. Checkpoint restart support for heterogeneous hpc applications. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. IEEE, 242–251.

[38] Aurick Qiao, Bryon Aragam, Bingjing Zhang, and Eric Xing. 2019. Fault Tolerance in Iterative-Convergent Machine Learning. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.), Vol. 97. PMLR, 5220–5230. https://proceedings.mlr.press/v97/qiao19a.html

[39] Yanru Qu, Han Cai, Kan Ren, Weinan Zhang, Yong Yu, Ying Wen, and Jun Wang. 2016. Product-based Neural Networks for User Response Prediction. arXiv:1611.00144 [cs.LG]

[40] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 497–514.

[41] Elvis Rojas, Albert Njoroge Kahira, Esteban Meneses, Leonardo Bautista Gomez, and Rosa M Badia. 2021. A Study of Checkpointing in Large Scale Training of Deep Neural Networks. arXiv:2012.00825 [cs.DC]

[42] Brent Smith and Greg Linden. 2017. Two decades of recommender systems at Amazon. com. *Ieee internet computing* 21, 3 (2017), 12–18.

[43] Nigel Tan, Jakob Luettgau, Jack Marquez, Keita Teranishi, Nicolas Morales, Sanjukta Bhowmick, Franck Cappello, Michela Taufer, and Bogdan Nicolae. 2023. Scalable Incremental Checkpointing using GPU-Accelerated De-Duplication. In *Proceedings of the 52nd International Conference on Parallel Processing*. 665–674.

[44] Jizhe Wang, Pipei Huang, Huan Zhao, Zhibo Zhang, Binqiang Zhao, and Dik Lun Lee. 2018. Billion-scale commodity embedding for e-commerce recommendation in alibaba. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*. 839–848.

[45] Qinlong Wang, Tingfeng Lan, Yinghao Tang, Bo Sang, Ziling Huang, Yiheng Du, Haitao Zhang, Jian Sha, Hui Lu, Yuanchun Zhou, et al. 2024. DLRover-RM: Resource Optimization for Deep Recommendation Models Training in the Cloud. *Proc. VLDB Endow.* (2024).

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

[47] Zheng Wang, Yuke Wang, Boyuan Feng, Dheevatsa Mudigere, Bharath Muthiah, and Yufei Ding. 2022. EL-Rec: Efficient large-scale recommendation model

training via tensor-train embedding table. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–14.

[48] Tianyu Zhang, Kaige Liu, Jack Kosaian, Juncheng Yang, and Rashmi Vinayak. 2023. Efficient Fault Tolerance for Recommendation Model Training via Erasure Coding. *Proc. VLDB Endow.* 16, 11 (July 2023), 3137–3150. https://doi.org/10.14778/3611479.3611514

[49] Weitao Zhang, Yinlong Xu, Yongkun Li, Yueming Zhang, and Dinglong Li. 2018. FlameDB: A key-value store with grouped level structure and heterogeneous Bloom filter. *IEEE Access* 6 (2018), 24962–24972.

[50] Yueming Zhang, Yongkun Li, Fan Guo, Cheng Li, and Yinlong Xu. 2018. {ElasticBF}: Fine-grained and Elastic Bloom Filter Towards Efficient Read for {LSM-tree-based}{KV} Stores. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*.